International Journal of Reconfigurable Computing

Selected Papers from the International Workshop on Reconfigurable Communication-Centric Systems on Chips (ReCoSoC' 2010)

Guest Editors: Michael Hübner, Jürgen Becker, Loïc Lagadec, and Gilles Sassatelli

Selected Papers from the International Workshop on Reconfigurable Communication-Centric Systems on Chips (ReCoSoC' 2010)

International Journal of Reconfigurable Computing

Selected Papers from the International Workshop on Reconfigurable Communication-Centric Systems on Chips (ReCoSoC' 2010)

Guest Editors: Michael Hübner, Jürgen Becker, Loïc Lagadec, and Gilles Sassatelli

Editorial Board

Contents

Selected Papers from the International Workshop on Reconfigurable Communication-Centric Systems on Chips (ReCoSoC 2010), Michael Hübner, Jürgen Becker, Loïc Lagadec, and Gilles Sassatelli
Volume 2011, Article ID 865402, 1 page

Reconfiguration Techniques for Self-X Power and Performance Management on Xilinx Virtex-II/Virtex-II-Pro FPGAs, Christian Schuck, Bastian Haetzer, and Jürgen Becker
Volume 2011, Article ID 671546, 12 pages

Dynamic Reconfigurable Computing: The Alternative to Homogeneous Multicores under Massive Defect Rates, Monica Magalha es Pereira and Luigi Carro
Volume 2011, Article ID 452589, 17 pages

An NoC Traffic Compiler for Efficient FPGA Implementation of Sparse Graph-Oriented Workloads, Nachiket Kapre and André Dehon
Volume 2011, Article ID 745147, 14 pages

On Self-Timed Circuits in Real-Time Systems, Markus Ferringer
Volume 2011, Article ID 972375, 16 pages

Experiment Centric Teaching for Reconfigurable Processors, Loïc Lagadec, Damien Picard, Youenn Corre, and Pierre-Yves Lucas
Volume 2011, Article ID 952560, 14 pages

Exploration of the Power-Performance Tradeoff through Parameterization of FPGA-Based Multiprocessor Systems, Diana Göhringer, Jonathan Obie, André L. S. Braga, Michael Hübner, Carlos H. Llanos, and Jürgen Becker
Volume 2011, Article ID 985931, 17 pages

A Security Scheme for Dependable Key Insertion in Mobile Embedded Devices, Alexander Klimm, Benjamin Glas, Matthias Wachs, Sebastian Vogel, Klaus D. Müller-Glaser, and Jürgen Becker
Volume 2011, Article ID 820454, 19 pages

Evaluation of the Reconfiguration of the Data Acquisition System for 3D USCT, Matthias Birk, Clemens Hagner, Matthias Balzer, Nicole V. Ruiter, Michael Hübner, and Jürgen Becker
Volume 2011, Article ID 952937, 9 pages

Dynamic Application Model for Scheduling with Uncertainty on Reconfigurable Architectures, Ismail Ktata, Fakhreddine Ghaffari, Bertrand Granado, and Mohamed Abid
Volume 2011, Article ID 156946, 15 pages

Prime Field ECDSA Signature Processing for Reconfigurable Embedded Systems, Benjamin Glas, Oliver Sander, Vitali Stuckert, Klaus D. Müller-Glaser, and Jürgen Becker
Volume 2011, Article ID 836460, 12 pages

Exploration of Heterogeneous FPGA Architectures, Umer Farooq, Husain Parvez, Habib Mehrez, and Zied Marrakchi
Volume 2011, Article ID 121404, 18 pages

A Middleware Approach to Achieving Fault Tolerance of Kahn Process Networks on Networks on Chips, Onur Derin, Erkan Diken, and Leandro Fiorin
Volume 2011, Article ID 295385, 15 pages

Exploring Online Synthesis for CGRAs with Specialized Operator Sets, Stefan Döbrich and Christian Hochberger
Volume 2011, Article ID 601986, 22 pages

A Self-Checking Hardware Journal for a Fault-Tolerant Processor Architecture, Mohsin Amin, Abbas Ramazani, Fabrice Monteiro, Camille Diou, and Abbas Dandache
Volume 2011, Article ID 962062, 15 pages

Static Scheduling of Periodic Hardware Tasks with Precedence and Deadline Constraints on Reconfigurable Hardware Devices, Ikbel Belaid, Fabrice Muller, and Maher Benjemaa
Volume 2011, Article ID 591983, 28 pages

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 865402, 1 page
doi:10.1155/2011/865402

Editorial

Selected Papers from the International Workshop on Reconfigurable Communication-Centric Systems on Chips (ReCoSoC' 2010)

Michael Hübner,[1] Jürgen Becker,[1] Loïc Lagadec,[2] and Gilles Sassatelli[3]

[1] *Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany*
[2] *Lab-STICC MOCS, CNRS, University of Western Brittany (UBO), Brest, France*
[3] *LIRMM, CNRS, University of Montpellier II, Montpellier, France*

Correspondence should be addressed to Michael Hübner, michael.huebner@kit.edu

Received 25 July 2011; Accepted 25 July 2011

The fifth edition of the Reconfigurable Communication-centric Systems-on-Chip workshop (ReCoSoC' 2010) was held in Karlsruhe, Germany, from May 17th to May 19th, 2010.

ReCoSoC is intended to be a periodic annual meeting to expose and discuss gathered expertise as well as state-of-the-art research around SoC-related topics through plenary invited papers and posters. ReCoSoC is a 3-day long event which endeavors to encourage scientific exchanges and collaborations.

ReCoSoC aims to provide a prospective view of tomorrow's challenges in the multibillion transistor era, taking into account the emerging techniques and architectures exploring the synergy between flexible on-chip communication and system reconfigurability.

This special issue includes the following Topics: embedded reconfigurability in all its forms; on-chip communication architectures; multiprocessor systems-on-chips; system and SoC design methods; asynchronous design techniques; low-power design methods; middleware and OS support for reconfiguration and communication; new paradigms of computation including bioinspired approaches.

Michael Hübner
Jürgen Becker
Loïc Lagadec
Gilles Sassatelli

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 671546, 12 pages
doi:10.1155/2011/671546

Research Article

Reconfiguration Techniques for Self-X Power and Performance Management on Xilinx Virtex-II/Virtex-II-Pro FPGAs

Christian Schuck, Bastian Haetzer, and Jürgen Becker

Institut für Technik der Informationsverarbeitung (ITIV), Karlsruher Institut für Technologie (KIT), Vincenz-Prießnitz-Straße 1, 76131 Karlsruhe, Germany

Correspondence should be addressed to Christian Schuck, schuck@itiv.uni-karlsruhe.de

Received 29 August 2010; Accepted 14 December 2010

Academic Editor: Michael Hübner

Xilinx Virtex-II family FPGAs support an advanced low-skew clock distribution network with numerous global clock nets to support high-speed mixed frequency designs. Digital Clock Managers in combination with Global Clock Buffers are already in place to generate the desired frequency and to drive the clock networks with different sources, respectively. Currently, almost all designs run at a fixed clock frequency determined statically during design time. Such systems cannot take the full advantage of partial and dynamic self-reconfiguration. Therefore, we introduce a new methodology that allows the implemented hardware to dynamically self-adopt the clock frequency during runtime by reconfiguring the Digital Clock Managers. We also present a method for online speed monitoring which is based on a two-dimensional online routing. The created speed maps of the FPGA area can be used as an input for the dynamic frequency scaling. Figures for reconfiguration performance and power savings are given. Further, the tradeoffs for reconfiguration effort using this method are evaluated. Results show the high potential and importance of the distributed dynamic frequency scaling method with little additional overhead.

1. Introduction

Xilinx Virtex FPGAs have been designed with high-performance applications in mind. They feature several dedicated Digital Clock Managers (DCMs) and Digital Clock Buffers for solving high-speed clock distribution problems. Multiple clock nets are supported to enable highly heterogeneous mixed frequency designs. Usually all clock frequencies for the single clock nets and the parameters for the DCMs are determined during design time through static timing analysis. Targeting maximum performance these parameters strongly depend on the longest combinatorial path (critical path) between two storage elements. For minimum power the required throughput of the design unit determines the lower boundary of the possible clock frequency. In both cases nonadjusted clock frequencies lead to waste of either processing power or energy [1, 2].

Considering the feature of partial and dynamic self-reconfiguration of Xilinx Virtex FPGAs, during runtime a high dynamic and flexibility arises. Static analysis methods are no longer able to sufficiently determine an adjusted clock frequency during design time. At the same time a new partial module is reconfigured onto the FPGA grid, its critical path changes, and in turn the clock frequency has to be adjusted as well during runtime to fit the new critical path. On the other side the throughput requirement of the application or the environmental conditions may change over time making an adjustment of clock frequency necessary.

Therefore, a new paradigm of system design is necessary to efficiently utilize the available processing power of future chip generations. To address this issue in [3] the Digital on Demand Computing Organism (DodOrg) was proposed, which is derived from a biological organism. Decentralisation of all system instances is the key feature to reach the desired goals of self-organisation, self-adoption, and self-healing, in short the self-x features. The hardware architecture of the DodOrg system consists of many heterogeneous so-called Organic Processing Cells (OPCs) that communicate through the artNoC [4] router network as shown in Figure 1. In general, all OPCs are made of the same

blueprint. On the one side they contain a highly adaptive heterogeneous data path and on the other side several common units are responsible for the organic behaviour of the architecture. Among them a Power Management Unit is able to perform dynamic frequency scaling (DFS) on OPC level. It can control and adjust performance and power consumption of the cell according to the actual computational demands of the application and the critical path of the cell's data path. DFS has a high potential, as it decreases the dynamic power consumption by decreasing the switching activity of flip flops, gates in the fan-out of flip flops, and the clock trees. Hence, the cell's clock domain is decoupled by the network interface and can operate independently from the artNoC and the other OPCs of the organic chip.

In [5], we presented a prototype implementation of the DodOrg architecture on a Virtex FPGA, where it is possible to dynamically change the cells data path through a 2-dimensional partial and dynamic reconfiguration. Therefore, a novel IP core, the Virtual-ICAP-Interface, was developed in order to perform a fast 2-dimensional self-reconfiguration and provide a virtual decentralisation of the internal FPGA configuration access port (ICAP). This paper enhances the methodology by enabling the partial and dynamic self-reconfiguration of the Virtex DCMs, which is inherently not given, through the Virtual-ICAP-Interface. As a result, the desired self-adoption with respect to a fine grained power management could be achieved. Further, an application for the 2-dimensional self-reconfiguration is given by presenting an online speed monitoring method in Section 5.

The rest of the paper is organized as follows. Section 2 reviews several other proposals for DFS on FPGAs while Section 3 summarizes important aspects of the Xilinx Virtex II FPGA clock net infrastructure. Section 4 describes the details of our approach to dynamically reconfigure the DCMs during runtime. Section 6 shows DCM reconfiguration performance and power saving figures. Finally, Section 7 concludes the work and gives an outlook to future work.

2. Related Work

Recently, several works have been published dealing with power management and especially clock management on FPGAs. All authors agree that there is a high potential for using DFS method in both ASIC and FPGA designs [2, 6].

In [1], the authors show that even because of FPGA process variations and because of changing environmental conditions (hot, normal, and cold temperature) dynamically clocking designs can lead to a speed improvement of up to 86% compared to using a fixed, statically estimated clock during design time. The authors use an external programmable clock generator that is controlled by a host PC. However, in order to enable the system to self-adapt its clock frequency on-chip solutions are required.

In [6], the authors proposed an online solution for clock gating. They propose a feedback multiplexer with control logic in front of the registers. So it is possible to keep the register value and to prevent the combinatorial logic behind the register to toggle. But simultaneously they highlight that clock gating on FPGAs could have a much higher power saving efficiency if it would be possible to completely gate the FPGA clock tree. To overcome this drawback in [7] the authors provide an architectural block that is able to perform DFS. However, this approach leads to low-speed designs and clock skew problems as it is necessary to insert user logic into the clock network.

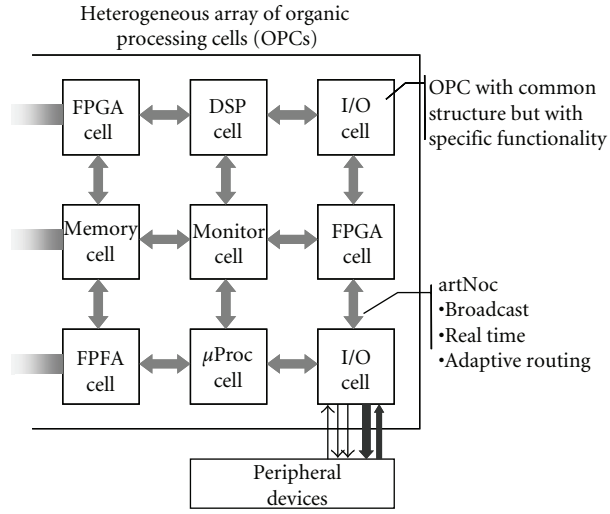


Figure 1: DodOrg organic hardware architecture overview.

We show that on Xilinx Virtex-II no additional user logic is necessary to efficiently and reliably perform a fine grained self-adaptive DFS. All advantages of the high-speed clock distribution network could be maintained.

3. Xilinx Clock Architecture

This section gives a brief overview over the Xilinx Virtex-II clock architecture as our work makes extensive use of the provided features.

3.1. Clock Network Grid. Besides normal routing resources Xilinx Virtex-II FPGAs have a dedicated low-skew high-speed clock distribution network [8, 9]. They feature 16 global clock buffers (BUFGMUX, see Section 3.3) and support up to 16 global clock domains (Figure 2). The FPGA grid is partitioned into 4 quadrants (NW, SE, SW, and NE) with up to 8 clocks per quadrant. Eight clock buffers are in the middle of the top edge and eight are in the middle of the bottom edge. In principle any of these 16 clock buffer outputs can be used in any quadrant as long as opposite clock buffers on the top and on the bottom are not used in the same quadrant, that is, there is no conflict [9]. In addition, up to 12 DCMs are available. They can be used to drive clock buffers with different clock frequencies. In the following important features of the DCMs and clock buffers will be summarized.

3.2. Digital Clock Managers. Besides others, frequency synthesis is an important feature of the DCMs. Therefore, 2

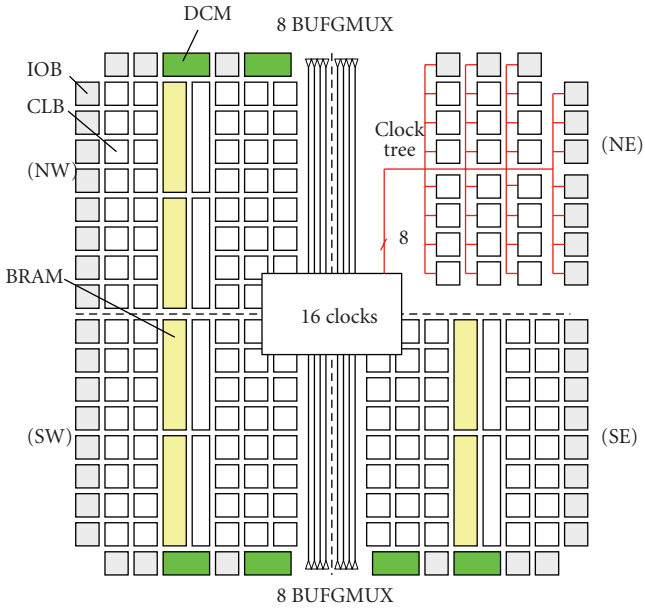


Figure 2: Simplified view of the Xilinx Virtex II clock distribution network.

main different programmable outputs are available. CLKDV provides an output frequency that is a fraction ($\div 1.5$, $\div 2, \div 2.5 \cdots \div 7, \div 7.5, \div 8, \div 9 \cdots \div 16$) of the input frequency CLKIN.

CLKFX is able to produce an output frequency that is synthesised by combination of a specified integer multiplier $M \in \{2 \ldots 32\}$ and a specified integer divisor $D \in \{1 \ldots 32\}$ by calculating CLKFX $= M \div D *$ CLKIN.

3.3. Global Clock Buffer. Global Clock Buffers have three different functionalities. In addition to pure clock distribution, they can also be configured as a global clock buffer with a clock enabler (BUFGCE). Then the clock can be stopped at any time at the clock buffer output.

Further, clock buffers can be configured to act as a "glitch-free" synchronous 2:1 multiplexer (BUFGMUX). These multiplexers are capable of switching between two clock sources at any time, by using the select input that can be driven by user logic. No particular phase relations between the two clocks are needed. For example, as shown in Figure 3 they can be configured to switch between two DCM clock CLKFX outputs. As we will see in the next section, our design makes use of this feature.

4. Organic System Architecture

Compared to μC ASIC solutions, SRAM-based FPGAs like Virtex-II consume a multiple of power. This is due to the fine-grained flexibility and adaptability and the involved overhead. By just using these features during design time to create a static design, most of the potential remains unused. Instead dynamic and partial online self-reconfiguration during runtime is a promising approach to exploit the full potential and even to close the energy gap. Therefore, in

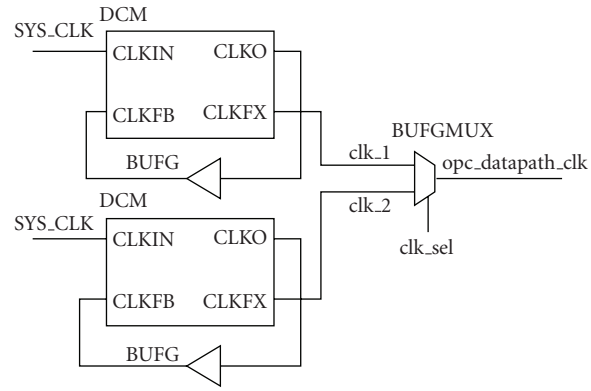


Figure 3: Example BUFGMUX /DCM configuration.

[5], we proposed to implement the OPC-based organic computing organisms on a Virtex-II Pro FPGA as shown in Figure 4.

This paper focuses on the power-related issues of the cell-based DodOrg architecture on the FPGA prototype. Important aspects to reach the desired goal of a fine-grained, decentralized self-adaptive power management will be discussed in the subsequent subsections.

4.1. Clock Partitioning. Depending on the size of the device several OPCs are mapped onto a single FPGA (Figure 4). The clock net of the highly adaptive data path (DP) of every OPC is connected to a BUFGMUX that is driven by a pair of DCMs. There is a power management unit (PMU) inside every OPC, which is connected to the select input of the BUFGMUX. So it can quickly choose between the two DCM clock sources. The DP-clock is decoupled from the artNoC-clock by using a dual-ported dual-clock FIFO buffer. Further, the PMU is connected to the artNoC. Thus, it is able to exchange power-related information with the other PMUs. Beyond that, it has access to the Virtual-ICAP-Interface. Therefore, during runtime every PMU can dynamically adapt the DCM CLKFX output clock frequency through partial self-reconfiguration by using the features of the Virtual-ICAP-Interface.

4.2. Virtual-ICAP-Interface. The Virtual-ICAP-Interface is a small and lightweight IP, which on the one side acts as a wrapper around the native hardware ICAP and on the other side connects to the artNoC network. It provides a virtual decentralisation of the ICAP as well as an abstraction of the physical reconfiguration memory. Its main purpose is to perform the Readback-Modify-Writeback (RMW) method in hardware. Therefore, a fast and true 2-dimensional reconfiguration of all FPGA resources is possible, that is, reconfiguration is no longer restricted to columns. Due to its partitioning into two clock domains, one clock domain for the artNoC controller side and one clock domain for the ICAP side, maximal reconfiguration performance could be achieved [5].

s every bit within the reconfiguration memory can be reconfigured independently the configuration of the DCMs

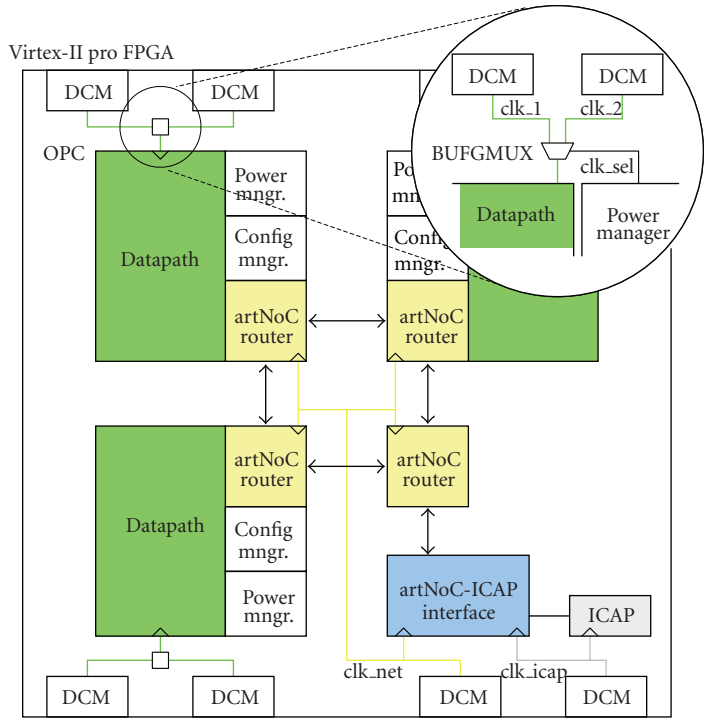


Figure 4: DodOrg FPGA floor plan/clock architecture.

can be altered as well. However, a special procedure is necessary which is described in the next subsection.

4.3. DCM Reconfiguration Details. During reconfiguration of DCMs it is important that a glitchless switching from one clock frequency to another can be guaranteed. In general, after initial setup the CLKDV and CLKFX outputs are only enabled when a stable clock is established. After that, the DCM is locked to the configured frequency, as long as the jitters of the input clock CLKIN stay in a given tolerance range [9]. For our scenario we assume that the input clock is stable.

If we change the DCM configuration (D, M) in configuration memory to switch from one clock frequency to a different frequency while the DCM is locked, it loses the lock and no stable output, that is, no output can be guaranteed. Therefore, to ensure a consistent locking to the new frequency the following steps have to be performed:

(1) stop the DCM by writing a zero configuration ($D = 0, M = 0$),

(2) write the new configuration ($D = D_{\text{new}}, M = M_{\text{new}}$).

To simplify the handling of the DCM reconfiguration this two-step procedure is internally executed by the Virtual-ICAP-Interface. It therefore features a special DCM addressing mode, for an easy access to the DCM configuration bits. Figure 5 shows a plot of a DCM reconfiguration procedure performed by the Virtual-ICAP-Interface. The plots were recorded by a 4-channel digital oscilloscope with all important signals routed to FPGA pins. Figure 5(a) shows the ICAP enable signal that is asserted by the Virtual-ICAP-Interface during ICAP read and write operation. It is an indicator for the overall duration of the reconfiguration procedure. It strongly depends on the device size or rather on the configuration frame length. In this case a Virtex-II Pro XC2VP30 device was used with a frame length of 824 Bytes. For reconfiguration of the DCM just a single configuration frame has to be processed. From the beginning of the ICAP enable low phase to the spike in Figure 5(a) the configuration frame is read back from the configuration memory.

Then, the ICAP is configured to write mode and the zero configuration to shut off the DCM is written followed by a dummy frame to flush the ICAP input register. As soon as the writing of the dummy frame is finished the DCM stops. Figure 5(c) shows a zoom of the DCM CLKFX output (Figure 5(b)) at this point in time. We see that the DCM CLKFX was running at 6.25 MHz and stops without any jitter or spikes. Immediately after the dummy frame, the read back frame which has been merged with the new DCM parameters is written back to the ICAP followed by a second dummy frame. As soon as the dummy frame is processed the DCM CLKFX output runs with the new frequency in this case 8.33 MHz. Figure 5(d) shows a zoom of this point in time. Again no glitches or spikes occur. The overall processing time for a complete DCM reconfiguration in this case is 60.7 μsec. In general, the reconfiguration time for a different Virtex-II family device is given by

$$t_{\text{DCM}} \sim \text{frame length [Byte]} * \frac{2}{67.7}\,[\text{Byte}/\mu\text{s}] + \text{frame\_length [Byte]} * \frac{4}{90.5}\,[\text{Byte}/\mu\text{s}] \quad (1)$$

(@ICAP_CLK = 100 MHz).

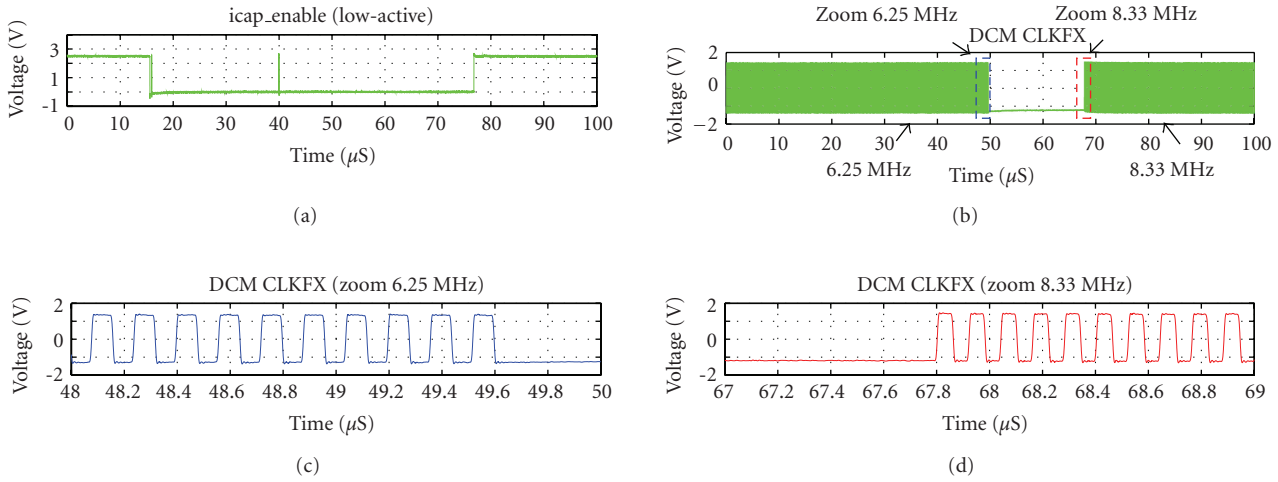


Figure 5: DCM reconfiguration performance: (a) ICAP enable, (b) DCM CLKFX output, (c) zoom of DCM CLKFX output before reconfiguration, and (d) zoom of DCM CLKFX output after reconfiguration.

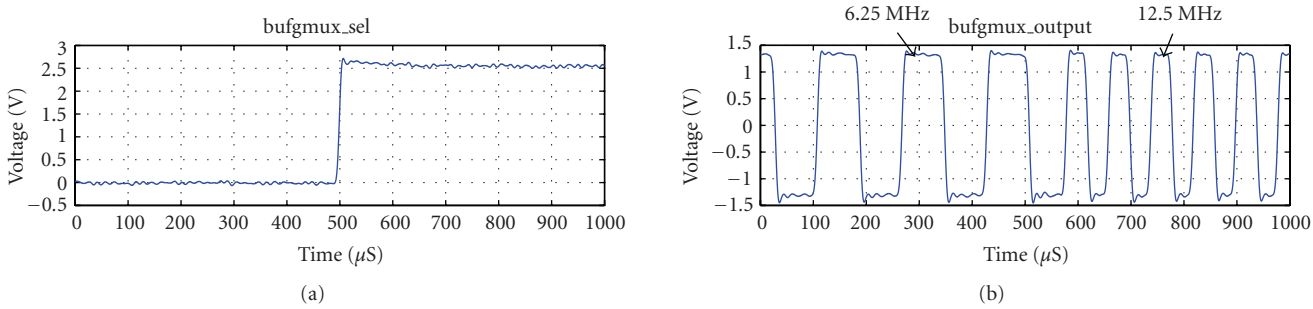


Figure 6: BUFGMUX clock switching: bufgmux_sel signal and bufgmux_output signal.

The two summands in the formula are resulting from the fact that ICAP has different throughputs for reading and writing reconfiguration data [5].

Therefore, this procedure presents a save method to dynamically reconfigure DCMs during runtime. However, even if self-adaptive decentralized DFS can be realized with the presented method two main drawbacks are obvious:

(1) relatively long setup delay until the new frequency is valid (in this example: 60.7 μs),

(2) interruption of clock frequency during reconfiguration (in this example: 18.2 μs).

This means that the method is appropriate for reaching long-term or intermediate-term power management goals, that is, a new data path is configured and the clock frequency is adapted to its critical path and then stays constant until a new data path is required. But if a frequent and immediate switching is necessary, for example, when data arrives in burst and between burst the OPC wants to toggle between shut-off ($f_{DP} = 0$ Hz) and maximal performance ($f_{DP} = f_{max}$), the method needs to be extended.

In this case a setup consisting of two DCMs and a BUFGMUX, as shown in Figure 3, can be chosen. The select input of the BUFGMUX is connected to the PMU of the OPC. Therefore, it is able to toggle between two frequencies immediately without any delay, as shown in Figure 6. Further the interruption of clock frequency during reconfiguration can be hidden. By a combination of both techniques a broad spectrum of different clock frequencies as well as an immediate uninterrupted switching is available.

5. Speed Monitoring

In the preceding section we presented the method to reconfigure the DCMs on Xilinx Virtex-II FPGAs during runtime. If the runtime optimization goal is to run the design at maximum clock speed, in order to achieve the maximum system performance, the question arises: "what is actually the maximum clock frequency that the reconfigurable datapath can run

(1) in the specific situation (temperature, supply voltage)?

(2) in the specific reconfigurable area?

(3) on a specific FPGA?

Table 1: OSC macrofrequency (MHz).

	Macro 1	Macro 2	Macro 3
Real	105	91.2	78.9
STA 20C	88	76	65.3
STA 85C	82.7	71.5	61.1
Min gap	17	15.2	13.6
Min gap %	19.3	20	20.8

Static timing analysis during design time can only determine a worst case scenario, but it is not possible to consider factors like

(1) device-specific local speed variations,

(2) variable or instable device power supply,

(3) different ambient temperatures.

In the following section we present a method of online speed monitoring which takes the device- and environment-specific factors into account. Hence, it is possible to determine the maximum clock frequency of a PRM depending on the current situation. The results can be used as an input for the DCM reconfiguration presented in Section 4.3.

5.1. Measurement Method. The measurement method is based on a ring oscillator monitoring module (OSC). Figure 7(a) shows the basic structure of the used ring oscillator. It is composed of an inverter and a delay line connected as a ring. This circuit oscillates with the frequency $1/(2 * \text{t\_delay})$. To decouple the ring oscillator from the fan-out network a toggle flip flop was added as a second stage to keep the fan-out of the oscillator constant at one. For further calculations it has to be considered that the second stage divides oscillator frequency by two.

The actual layout of the oscillator macro 1 is shown in Figure 7(b). It was built with the Xilinx FPGA Editor, as a so-called hard macro. The goal was to build it very compact. For the delay line and the inverter we used 6 LUTs of the top 3 slices. The fourth slice is used to realize the decoupling toggle flip flop. The delay of the delay line is comprised of the propagation delay of the LUTs and CLB internal wire delays between the LUTs.

For our measurements we build 3 different oscillator macros (macro 1–3) with different delays of the delay line. This was achieved by altering the connection of the 6 delay LUTs and therefore altering the used routing resources.

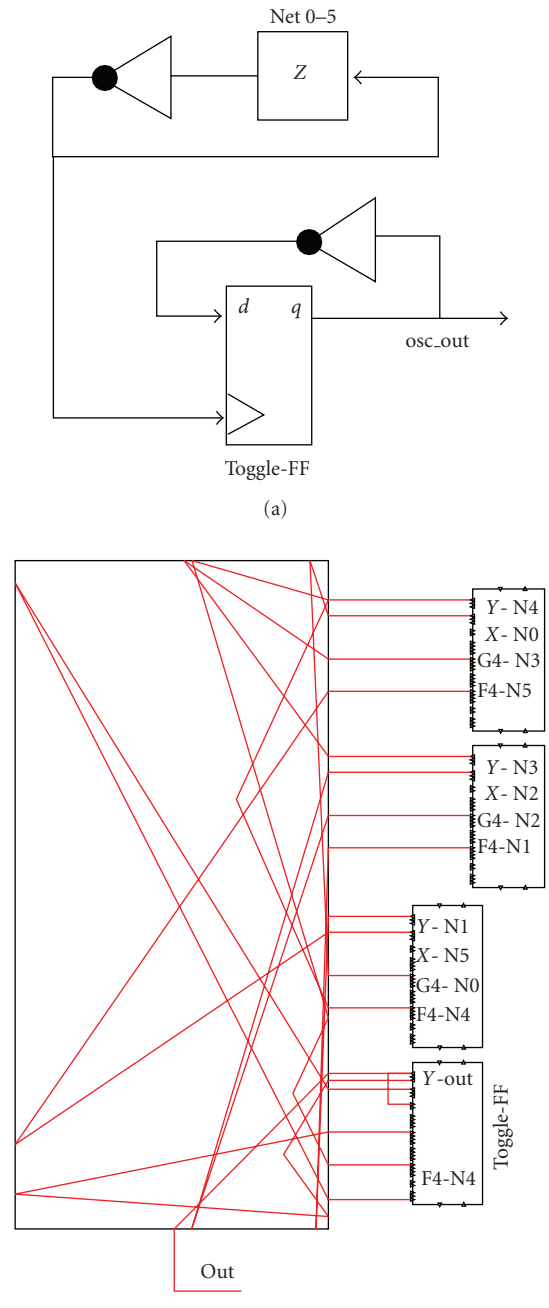


Figure 7: Oscillator macro zoom: (a) schematic and (b) physical FPGA Editor layout.

5.2. Experiment 1: Static Timing Analysis. In our first measurement we performed a static timing analysis (STA) of each of the oscillator macros to determine the theoretical oscillator frequency and compared the results with real measurement on board. The results are shown in Table 1. The STA was performed with the following settings: temperature: 20°C/85°C and voltage: 1.4 V. The real measurements on the board were performed with a room temperature of ca. 20°C and a voltage of 1.497 V.

The results show that at least a speed gap of 19% exists. During the measurements we noticed that the voltage parameter in the Xilinx Timing Analyzer V.9.103i has no effect on the STA results, even if it can be set. In later versions of the Xilinx Timing Analyzer the voltage parameter is taken into account but the Virtex–II series are no longer supported.

5.3. Experiment 2: Supply Voltage Speed Variation. Therefore, in our second experiment we did the same measurement with OSC-macro 2, but this time we altered the supply voltage and we did the measurement with 5 different but identical FPGA boards (board 1–5). The results of the oscillator frequency

Table 2: OSC macro 2 frequency (MHz).

Device id /voltage[V]	1.10	1.20	1.30	1.40	1.50	1.60
Board 1	66.304	72.064	75.904	78.08	78.592	77.696
Board 2	67.456	72.704	76.16	77.824	77.952	77.312
Board 3	65.536	71.04	74.752	76.928	77.824	77.44
Board 4	66.432	72.448	76.672	79.232	80.384	80
Board 5	66.688	72.192	76.416	78.464	79.104	78.208
Mean	66.4832	72.0896	75.9808	78.1056	78.7712	78.1312
Min	65.536	71.04	74.752	76.928	77.824	77.312
Max	67.456	72.704	76.672	79.232	80.384	80
Delta	1.92	1.664	1.92	2.304	2.56	2.688

are shown in Table 2. We altered the supply voltage in discrete steps of 0.1 V from the minimum voltage of 1.1 V to the maximum voltage of 1.6 V. Looking at a single board the biggest frequency delta across the different supply voltages was detected at board 3 with 13.95 MHz or 18.4%. The smallest delta was detected at board 5 with 10.49 MHz or 14% of the mean value.

Looking at different boards at the same supply voltage the biggest delta in frequency between two boards was detected at 1.6 V with a delta of 2.69 MHz or 3.44% of the mean value. The smallest delta was detected at 1.2 V with a delta of 1.66 MHz or 2.31% of the mean value. To summarize this experiment it can be said that

(1) supply voltage has a great impact on the device speed,

(2) even if we just measured 5 different devices this exemplary shows that there is potential for speed improvement if the clock frequency can be tailored device specific.

If we compare these results with the first experiment, we can derive that the Xilinx Timing Analyzer database assumes for the STA a worst case supply voltage of 1.3 V.

5.4. Experiment 3: Local Speed Variations. In Section 5.3 we have seen device-specific speed characteristics, probably based on fabrication variations. The question arises if there are even some regions on the device which are faster than other regions on the same device. In a third experiment we wanted to answer this question. Therefore, we constructed an automatic measurement system with the following main characteristics:

(i) fast measurement of a large number of CLBs,

(ii) on-Chip solution that can also be used for online self-monitoring in our organic system.

The key technique of the measurement system is based on a repeated 2-dimensional partial self-reconfiguration by using the Virtual-ICAP-Interface (see Section 4.2). In a first step the oscillator macro is placed at the CLB that should be monitored. In a second step an XY-online routing based on double lines is performed, in order to connect the frequency output of the oscillator macro with the measurement unit that determines the frequency. In the following, our method

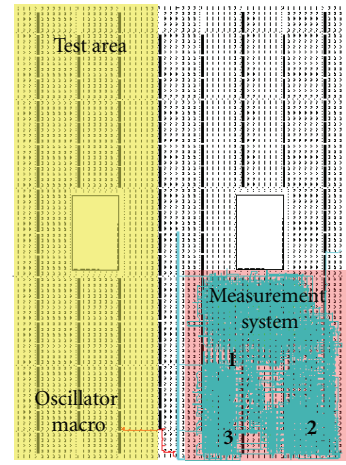


Figure 8: Speed measurement system layout.

to characterize a complete device will be presented in detail followed by the discussion of the results.

Figure 8 shows the basic structure of the example measurement system on an XC2VP30 FPGA. It consists of the static measurement system on the bottom right corner. It contains a MicroBlaze (MB) soft-core processor (1) that controls the reconfiguration and measurement process. It has connections via FSL with the Virtual-ICAP-Interface (VICAP) (2) for reconfiguration, the frequency measurement unit (FMU) and an UART (3) to communicate with a host system (HS). The FMU is connected to the frequency output of the oscillator macro.

The frequency measurement of a single CLB consists of the following steps:

(i) placement of the oscillator macro to target CLB (MB/VICAP),

(ii) online routing to connect oscillator macro with frequency measurement unit(MB/VICAP),

(iii) trigger the start of the measurement via FSL (MB),

(iv) frequency measurement (FMU),

(v) transmit the measurement result via FSL to MB (FMU),

(vi) transmit the result via UART to host system (MB),

(vii) store data into database (HS).

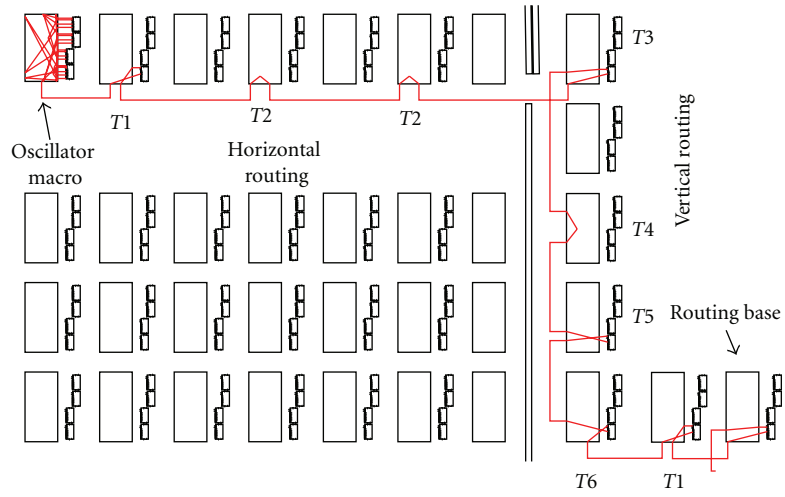


Figure 9: Zoom of online routing macros.

In the following, the basic technique of the used online routing strategy will be presented briefly. Figure 9 shows a zoom of a connection between the oscillator macro and the frequency measurement unit. The online routing is based on double lines only, in order to simplify the routing algorithm. Six different types of routing connections are used as marked in Figure 9:

- T1: connects the middle of a horizontal double line with the start of the next horizontal double line,
- T2: connects the end of a horizontal double line with the start of the next horizontal double line,
- T3: down corner: connects the end of a horizontal double line with the start of a vertical double line,
- T4: connects the end of a vertical double line with the start of the next vertical double line,
- T5: connects the middle of a vertical double line with the start of the next vertical double line,
- T6: right corner: connects the middle of a vertical double line with the start of a horizontal double line.

The output of the oscillator macro is routed to the middle connection of a horizontal double line. At the other end a so called "routing base" macro also connects to the middle connection of a horizontal double line, in order to ensure a fixed connection point for the routing algorithm. The output of the routing base macro is connected to the frequency measurement unit. The path between the output of the oscillator macro and the input of the "routing base" can be established from every CLB north-west of the routing base by reconfiguring the routing connections T1–T6 along the path. Similar routing connections could be created for other directions.

To reconfigure one routing connection just 2 FPGA configuration frames need to be processed by the Virtual-ICAP Interface. Therefore, without optimization (e.g., reconfiguring multiple vertical routing connections simultaneously) each routing connection needs 80 μs to be set. For the oscillator macro all 22 frames of a CLB are reconfigured, which takes 880 μs. For example, to place the oscillator macro and to establish the routing connection as shown in Figure 9 it takes

$$8 * 80\,\mu s + 880\,\mu s = 1520\,\mu s. \quad (2)$$

The configuration data for the routing types T1–T6 and the oscillator macro are stored as constants in the MicroBlaze program memory. All configuration data together require 302 bytes of memory. Because of its short reconfiguration times and little memory requirements the approach is suitable to be used for online speed monitoring of larger areas, for example, the reconfigurable area of an OPC.

In Figures 11, 12, and 13 the results for experiment 3 of boards 1–5 are shown. Boards 2 and 5 as well as boards 1, 3, and 4 show similar speed maps. At all boards the slowest CLBs are located along the edges of the device. Towards the middle of the device the speed gradually increases so that the fastest CLBs are located along the middle axes. The biggest frequency delta between maximum (106 MHz) and minimum frequency (101 MHz) shows board 2 with 5 MHz and a variance of 0.5 MHz (sample size 1440 CLBs). Figure 10 shows the distribution of the OSC speed for all boards. We repeated the measurements with macro 2 and macro 3 (different routing resources) for all boards (not shown here) and got very similar speed maps. That means fast CLBs remained fast and slow CLBs remained slow.

With respect to a real critical path which is mapped either to a region with fast CLBs or a region with slow CLBs the results become more relevant as faster the design runs. For example, on board 2 for a 100 MHz design the frequency delta is 4-5 MHz, whereas for a 200 MHz design the frequency delta grows proportional to ca. 8–10 MHz. This means in turn that for slower designs these effects can be more and more neglected.

To summarize the experiment, it can be said that local and device-specific speed variation could be measured. Their

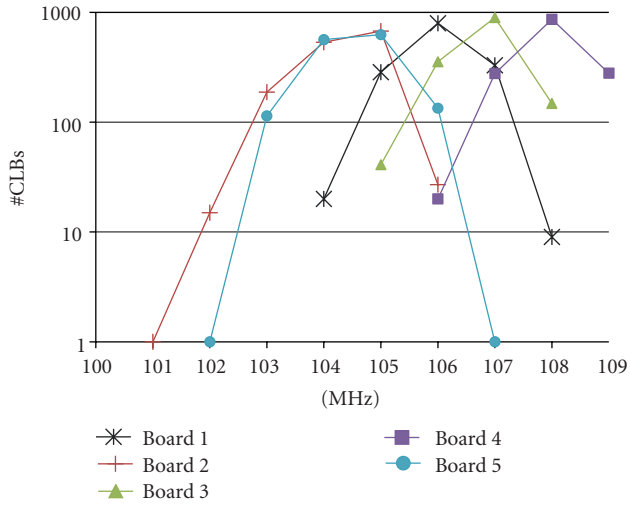


Figure 10: Distribution of OSC speed board 1–5.

Table 3: Component power consumption.

	"Passive" Power [mW]	"Active" Power [mW]
static_offset	—	11
DCM	—	37
PM	<1	<1
artNoC-ICAP-IF	<1	9
ICAP	69	76

Table 4: Resource utilization Virtual-ICAP-Interface.

Resource	Number	Percentage
Slices	364	2%
Slice flip flops	177	0%
4 input LUTs	664	2%
BRAMs	1	0%
MULT18 × 18 s	2	1%

impact compared to speed variations caused by temperature and supply voltage variations is quite low but will become more important as the speed of the designs increases. Especially, if further technology scaling causes bigger process variations, which leads to an increase of local speed variations, the presented online monitoring method becomes more relevant. However, even for Virtex-II FPGAs it can be used for fine tuning the DCM clock speed according to the speed of the region the design should run.

6. Power and Resource Considerations

In Section 4.3 the results for DCM reconfiguration times and tradeoffs have already been presented. This section evaluates the potential of power savings and performance enhancements in the context of module-based partial online reconfiguration. Especially, the overhead in terms of area and power consumption introduced by the DCM reconfiguration approach (PM, Virtual-ICAP-Interface, and DCM) is taken into account.

6.1. Test Setup Power Measurement. We calculated the power consumption by measuring the voltage drop over an external shunt resistor (0.4 Ohm) on the FPGA core voltage (FPGA VINT). As a test system the Xilinx XUP board with a Virtex-II Pro (XC2VP30) device was used. For all measurements the board source clock of 100 MHz was used as an input clock to the design.

To isolate the portions of power consumption, as shown in Table 3, several distinct designs have been synthesised.

For DCM measurement an array of toggle flip flops at 100 MHz with and without a DCM in the clock tree have been recorded and the difference of both values has been taken. For extracting ICAP power consumption a system consisting of PM, Virtual-ICAP-Interface, and ICAP instance and a second identical system but without ICAP instance have been implemented. After activation the PM sends bursts of two complete alternating configuration frames targeting the same frame in configuration memory. The ratio of toggling bits between the two frames is 80% and is considered to be representative for a partial reconfiguration. Therefore, before PM activation the "passive" power and after activation the "active" power could be measured. Again the difference in power consumption of the two systems was taken to extract ICAP portion. The other components were measured with the same methodology. Therefore, for example, all components necessary to implement the approach presented in Section 4.3 with two DCMs+BUFGMUX consume 196 mW when active, that is, 180 mW when passive. But it has to be considered that Virtual-ICAP-Interface as well as ICAP is also used for partial 2D reconfiguration.

6.2. Area and Resource Utilization. The resource requirement for the Virtual-ICAP-Interface with DCM reconfiguration mode is shown in Table 4.

6.3. Power Performance Evaluation. To put the previous power figures into a context we determined the power consumption of a MicroBlaze soft-core processor at different clock frequencies as shown in Figure 14. As we can see there is a high potential for power savings (e.g., the difference in power consumption in idle state between 100 MHz and 50 MHz is 170 mW).

The overhead (ICAP+VICAP) for DCM reconfiguration in a static design is in the range of an MB operating at 20 MHz. As expected, we see that there is a linear dependency between clock frequency and power consumption. Therefore, the energy consumed per clock cycle, $E = P/f_{\text{clk}}$; $f_{\text{clk}} = c * P$, is constant for all clock frequencies. This means, in terms of power savings for a static data path, there is no point for using reconfiguration of DCMs. A setup of DCM$_{f\max}$ and BUFGCE to toggle between $f = f_{\max}$ and $f = 0$ is most appropriate. In terms of performance, DCM reconfiguration can be used to evaluate maximum clock frequency during runtime.

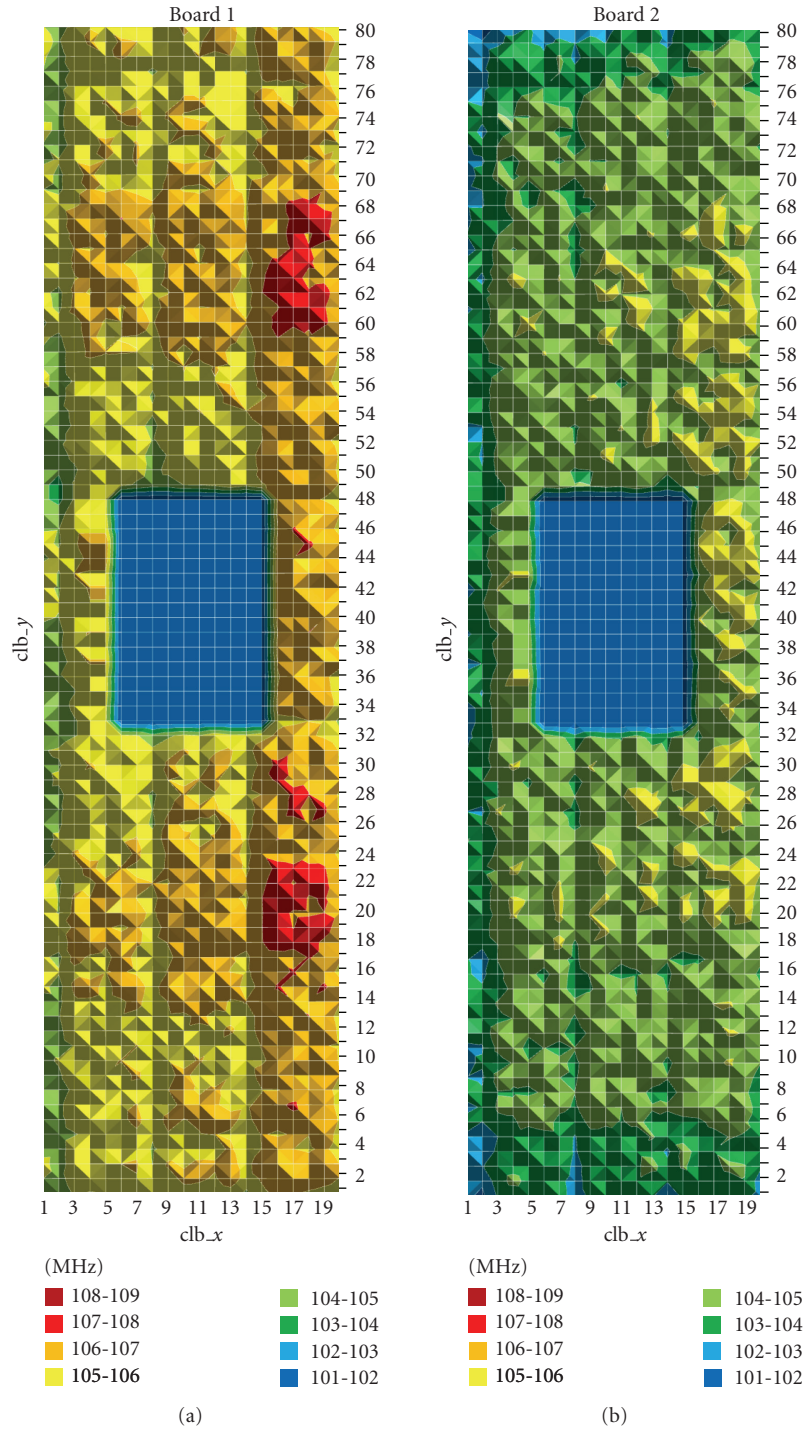


Figure 11: OSC frequency map (MHz) board 1/board 2.

In turn, in a dynamic scenario, where the data path and therefore also the critical path change, DCM reconfiguration is necessary to achieve maximum module performance. It also comes without any additional overhead as ICAP+VICAP +DCM are already needed for reconfiguration. The capability of DCM reconfiguration together with BUFGMUX provides the basis for fine-grained short- or long-term power management strategies.

7. Summary and Future Work

In this paper we have presented a novel methodology to dynamically reconfigure Digital Clock Managers on Xilinx Virtex-II devices through ICAP. On one side optimal performance of partial modules and on the other side the goal of uniform power consumption can be achieved without external hardware. Our measurements show that power

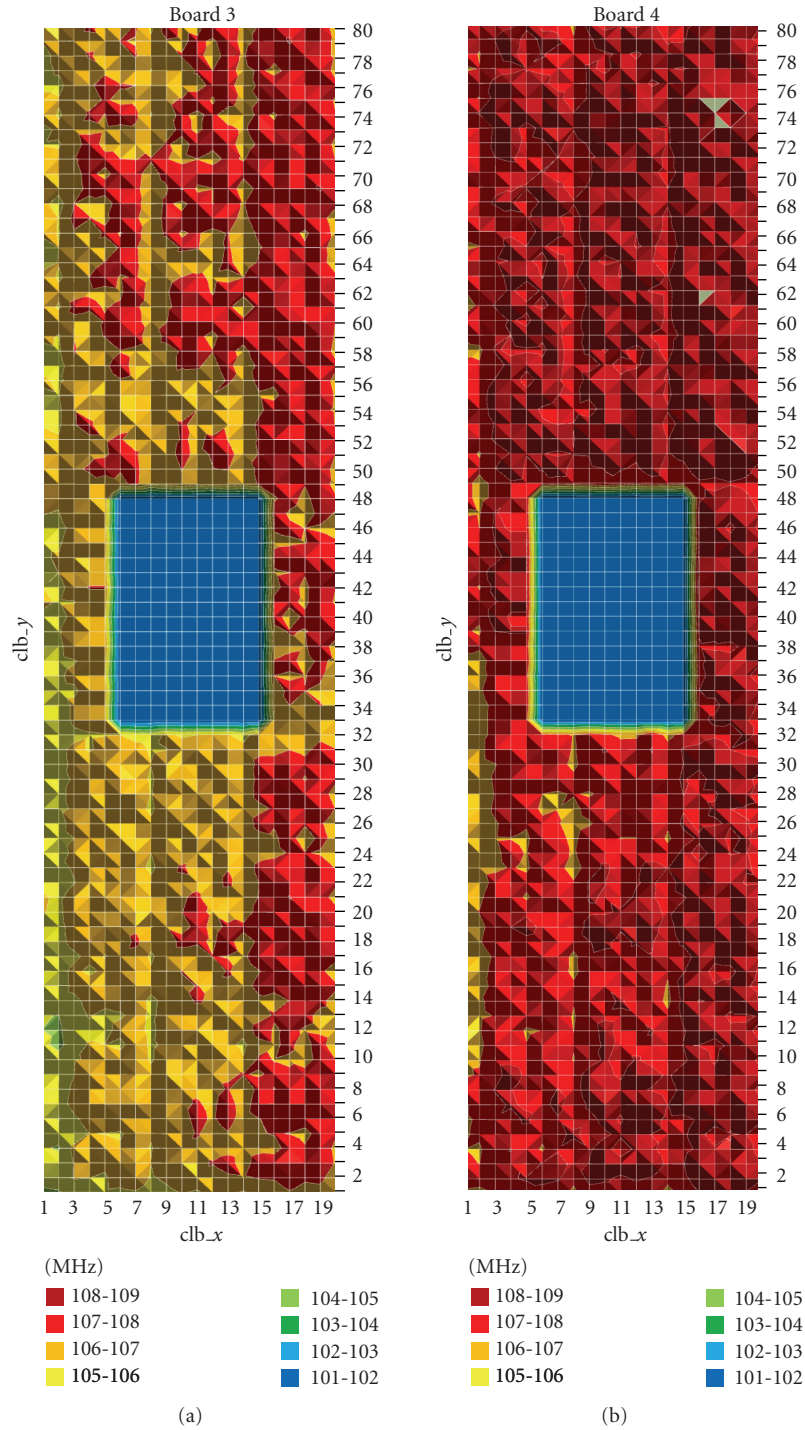


Figure 12: OSC frequency map (MHz) board 3/board 4.

consumed by the components of the proposed hardware framework, especially the DCMs itself, is not negligible and has to be counterweighted. With DCM reconfiguration times in the range of 60 μs long term power management goals can be reached. We also provide figures for reconfiguration times as well resource utilization.

Online on board speed measurements show the potential of device and situation dependent adjusted clock frequencies during runtime. Compared to static timing analysis performed at design time at least 19% of speed improvement can be achieved. A speed analysis of five different boards at different supply voltages shows that a voltage variation of 0.5 V results in a speed variation of 18.4%.

In order to detect FPGA region-specific speed characteristics a new lightweight on-chip online routing method is presented which allows to scan complete FPGA areas. Results

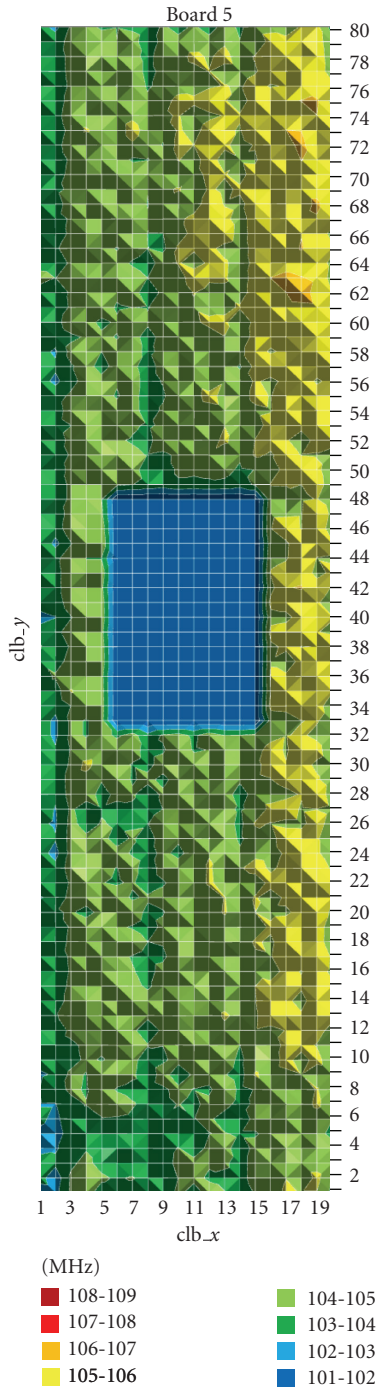


Figure 13: OSC frequency map (MHz) board 5.

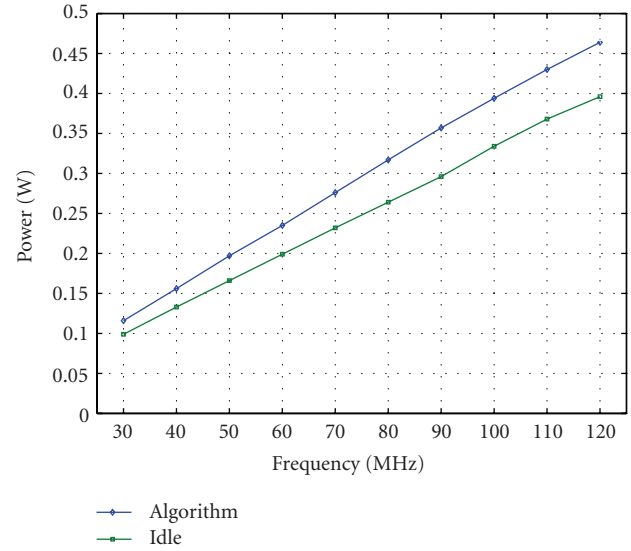


Figure 14: MicroBlaze power measurement.

show that region-specific speed characteristics exist that can be used for fine tuning the design with respect to maximum performance.

Future work is targeting towards the integration of the speed monitoring into the PM and examination of the system level power saving effect resulting from distributed power management with multiple PM and multiple clock domains.

References

[1] J. A. Bower, W. Luk, O. Mencer, M. J. Flynn, and M. Morf, "Dynamic clock-frequencies for FPGAs," *Microprocessors and Microsystems*, vol. 30, no. 6, pp. 388–397, 2006, special issue on FPGA's.

[2] I. Brynjolfson and Z. Zilic, "FPGA clock management for low power," in *Proceedings of International Symposium on Field-Programmable Gate Arrays (FPGA '00)*, 2000.

[3] J. Becker, K. Brändle, U. Brinkschulte et al., "Digital on-demand computing organism for real-time systems," in *Proceedings of the 19th International Conference on Architecture of Computing Systems(ARCS '06)*, W. Karl et al., Ed., 2006.

[4] C. Schuck, S. Lamparth, and J. Becker, "artNoC—a novel multi-functional router architecture for organic computing," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 371–376, August 2007.

[5] C. Schuck, B. Haetzer, and J. Becker, "An interface for a dezentralized 2D-reconfiguration on Xilinx Virtex-FPGAs for organic computing," in *Proceedings of Reconfigurable Communication-Centric SoCs (ReCoSoC '08)*, 2008.

[6] Z. Yan, J. Roivainen, and A. Mämmelä, "Clock-gating in FPGAs: a novel and comparative evaluation," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD '06)*, pp. 584–588, September 2006.

[7] I. Brynjolfson and Z. Zilic, "Dynamic clock management for low power applications in FPGAs," in *Proceedings of the 22nd Annual Custom Integrated Circuits Conference (CICC '00)*, pp. 139–142, May 2000.

[8] "Xilinx Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet," DS083 (v4.7) November 2007.

[9] "Xilinx Virtex-II Pro and Virtex-II Pro X FPGA User Guide," UG012 (V4.2) November 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 452589, 17 pages
doi:10.1155/2011/452589

Research Article

Dynamic Reconfigurable Computing: The Alternative to Homogeneous Multicores under Massive Defect Rates

Monica Magalhães Pereira and Luigi Carro

Instituto de Informática, Universidade Federal do Rio Grande do Sul, 91501-970 Porto Alegre, RS, Brazil

Correspondence should be addressed to Monica Magalhães Pereira, mmpereira@inf.ufrgs.br

Received 25 August 2010; Accepted 14 December 2010

Academic Editor: Michael Hübner

The aggressive scaling of CMOS technology has increased the density and allowed the integration of multiple processors into a single chip. Although solutions based on MPSoC architectures can increase application's speed through TLP exploitation, this speedup is still limited to the amount of parallelism available in the application, as demonstrated by Amdahl's Law. Moreover, with the continuous shrinking of device features, very aggressive defect rates are expected for new technologies. Under high defect rates a large amount of processors of the MPSoC will be susceptible to defects and consequently will fail, not only reducing yield but also severely affecting the expected performance. This paper presents a run-time adaptive architecture that allows software execution even under aggressive defect rates. The proposed architecture can accelerate not only highly parallel applications but also sequential ones, and it is a heterogeneous solution to overcome the performance penalty that is imposed to homogeneous MPSoCs under massive defect rates.

1. Introduction

The scaling of CMOS technology has increased the density and consequently made the integration of several processors in one chip possible. Many architectural solutions with several cores can be found in the literature in the past decade [1]. These solutions are mainly used to accelerate execution through task level parallelism (TLP) exploitation. However, the speedup achieved by these systems is limited to the amount of parallelism available in the applications, as already foreseen by Amdahl [2]. According to Rutzig et al. [3], current embedded system domain applications present a heterogeneous behavior that includes not only highly parallel applications but also general purpose applications that are also migrating to embedded system domain, such as browsers and high definition video processing.

To cope with this heterogeneous behavior an architecture design is necessary to not only accelerate execution through TLP but also find other ways to accelerate software execution, for example, through ILP (instruction level parallelism) exploitation, or accelerate sequential execution. One solution to overcome Amdahl's law and sustain the speedup of MPSoCs is the use of heterogeneous cores, where each core is specialized in different application sets. An example of a heterogeneous architecture is the Samsung S5PC100 [4] used in Apple's iPhone technology [5]. The Samsung S5PC100 is a multimedia-based MPSoC that has an ARM-based central general purpose processor and five multimedia accelerators targeted to DSP processing. Each accelerator is targeted to a different technology such as JPEG codification and NTSC/PAL/HDMI technologies, [4].

Although the use of heterogeneous cores can be an efficient solution to improve the MPSoC's performance, there are other constraints that must be considered in the design of multicore systems, such as reliability. The scaling process shrinks wires' diameter, making them more fragile and susceptible to break. Moreover, it is also harder to keep contact integrity between wires and devices [6]. According to Borkar [7], in a 100 billion transistor device, 20 billion will fail in the manufacture and 10 billion will fail in the first year of operation.

At these high defect rates, it is highly probable that defects will affect most of the processors of the MPSoC (or even all the processors), causing yield reduction and aggressively

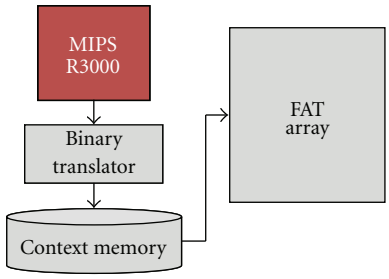


Figure 1: FAT-array system block diagram.

affecting the expected performance. Furthermore, in cases when all the processors are affected, this makes the MPSoC useless. An analysis shown in the next sections demonstrates that under a 20% defect rate a 64-core homogeneous MPSoC would have lost an average of 63 cores, presenting a completely sequential execution. On the other hand, considering a 0.1% defect rate of current technologies, on average only two cores are lost. More details about this analysis are presented in next sections.

To cope with this reliability and consequent loss of performance problem, one solution is to include some fault tolerance approach. Although there exist many solutions proposed to cope with manufacturing defects [8], most of these solutions do not cope with the high defect rates predicted to future technologies. Moreover, the proposed solutions present some additional cost that causes a high impact on area, power, or performance or even in all three [9]. Solutions such as dual and triple modular redundancy (DMR, TMR) presented by HP NonStop architectures [10] and Aggarwal's configurable isolation approach [11] have obvious area and power issues.

In this context, this paper presents a reconfigurable architecture as an alternative to homogeneous multicores that allows software execution even under high defect rates and accelerates execution of parallel and sequential applications. The architecture uses an on-line mechanism to configure itself according to the application, and its design provides acceleration in parallel as well as in sequential portions of the applications. In this way, the proposed architecture can be used to replace homogeneous MPSoCs, since it sustains performance even under high defect rates, and it is a heterogeneous approach to accelerate all kinds of applications. Therefore, its performance is not limited to the parallelism available in the applications.

To validate the architecture, we compare the performance and area of the system to a homogeneous MPSoC with equivalent area. The results indicate that the proposed architecture can sustain execution even under a 20% defect rate, while in the MPSoC all the processors become useless even under a 15% defect rate. Furthermore, with lower defect rates, the proposed architecture presents higher acceleration when compared to the MPSoC under the same defect rates, when the TLP available in the applications is lower than 100%.

The rest of this paper is organized as follows. Section 2 presents the adaptive system. Section 3 details the defect tolerance approach and some experimental results. Section 4 presents a comparison of area and performance between the reconfigurable system and the equivalent homogeneous MPSoC considering different defect rates. Section 5 presents some related work. Finally, Section 6 presents the conclusions and future works.

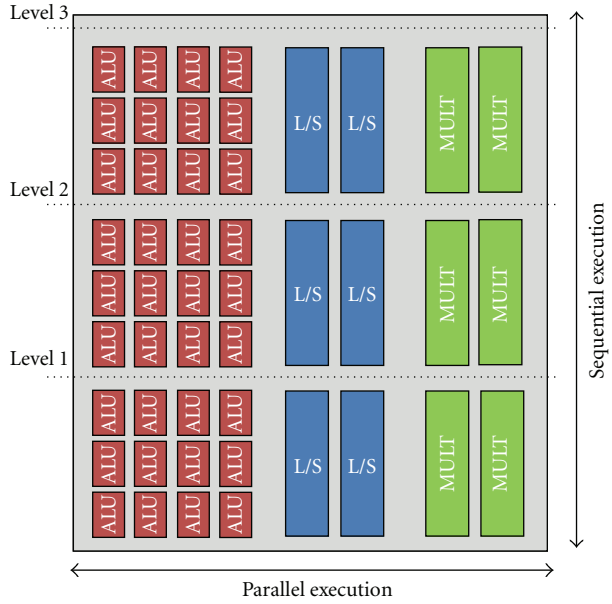


Figure 2: FAT-array block diagram.

2. Proposed Architecture

The FAT-array (fault tolerant array) system consists of a coarse-grained reconfigurable array tightly coupled to an MIPS R3000 processor; a mechanism to generate the configuration, called Binary Translator; and the context memory that stores the configuration [12, 13]. Figure 1 presents the block diagram of the FAT-array system.

The FAT-array consists of a combinational circuit that comprises three groups of functional units: the arithmetic and logic group, the load/store group, and the multiplier group. Figure 2 presents the FAT-array block diagram.

Each group of functional units can have a different execution time, depending on the technology and implementation strategy. Based on this, in this work the ALU group can perform up to three operations in one equivalent processor cycle and the other groups execute in one equivalent processor cycle. The equivalent processor cycle is called level. Figure 2 also demonstrates the parallel and sequential paths of the FAT-array.

The interconnection model, illustrated in Figure 3, is based on multiplexers and buses. The buses are called context lines and receive data from the context registers, which store the data from the processor's register file.

According to Figure 3, the reconfigurable architecture design presents a bottom-up data path. All data that are generated by the functional units flow from the bottom row to the top row through the interconnection model. Moreover,

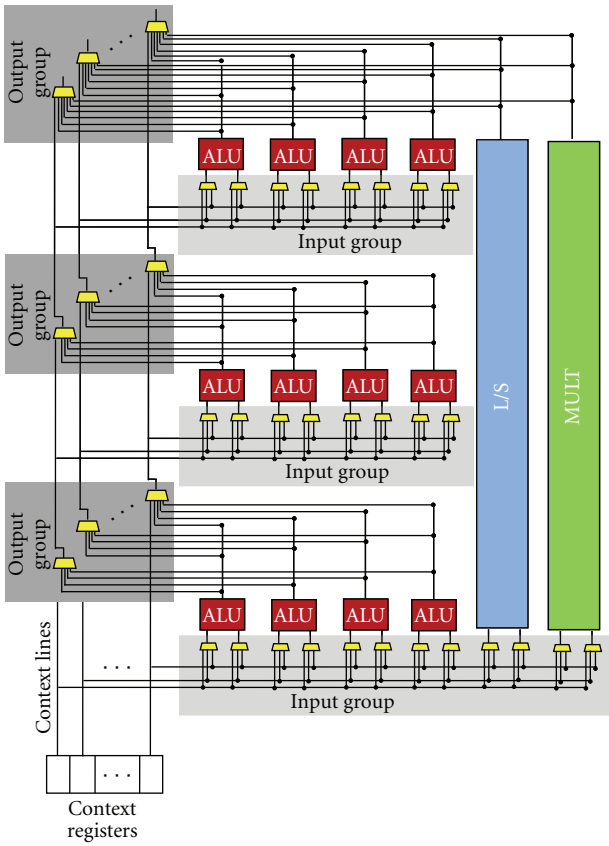
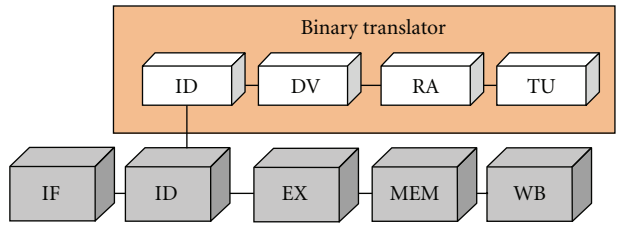


Figure 3: FAT-array interconnection model based on multiplexers and buses.

instructions placed in the same row are executed in parallel, since there are no data flowing from one unit to the other in the same row. Dependent instructions are executed in different rows because data among units flow from bottom to top of the architecture.

There are two groups of multiplexers: the input group that selects data used by each functional unit and the output group that selects the context line that receives the result from the operations performed by each functional unit. Moreover, each output multiplexer also has as input the context register. This input is used when the multiplexer needs only to bypass the previous data without the need of any functional unit.

The different execution times presented by each group of functional units allow the execution of more than one operation per level. Therefore, the FAT-array can perform up to three arithmetic and logic operations that present data dependency among each other in one equivalent processor cycle, consequently accelerating the sequential execution. Moreover, the execution time can be improved through modifications in functional units and with technology evolution, consequently increasing the acceleration of intrinsically sequential parts of a code. Even nonparallel code can have a better performance when executed in the structure illustrated in Figure 2, as shown in [12]. Moreover, the amount of functional units is defined according to area constraints



IF	Instruction fetch	ID	Instruction decode
ID	Instruction decode	DV	Dependency verification
EX	Execution	RA	Resource allocation
MEM	Memory access	TU	Table update
WB	Write back		

Figure 4: Binary translator's pipeline coupled to the processor's pipeline.

and/or an application set demand (given by a certain market, e.g., portable phones).

The Binary Translator (BT) unit implements a mechanism that dynamically transforms sequences of instruction to be executed on the array. The transformation process is transparent, with no need of instruction modification before execution, preserving software compatibility. Furthermore, the BT works in parallel with the processor's pipeline, presenting no extra overhead to the processor. Figure 4 illustrates the Binary Translator's steps attached to the MIPS R3000 pipeline.

2.1. Configuration Generation. To generate the configuration, the first stage of BT's pipeline (ID) consists in searching for sequences of instructions that can be executed in the FAT-array. The second stage (DV) checks data dependency among the current instruction and the previous ones that were already analyzed. Based on data dependency analyzed in previous stage, in the next stage (RA) BT searches for available functional units to perform resource allocation. Both data dependency and resource availability verification are performed through the management of tables that are updated in the last stage (TU). At the end of BT's pipeline a configuration is generated and stored in the context memory indexed by the program counter (PC) value of the first instruction from the sequence.

To check data dependency BT uses two tables to indicate which registers are used in each instruction. One table indexes the input registers and the other indexes the output registers. In this way, BT can verify if the registers used in the current instruction are also used in the previous one, finding RAW (read after write) dependencies.

To perform resource allocation BT uses a resource map that represents the functional units. Considering a defect-free architecture, when BT starts a configuration, all positions of the resource map are set as free. Every time an instruction is placed in a functional unit, its relative position in the resource map is set as busy. Therefore, to select a functional unit, after checking data dependency, BT needs to search in the resource map a free unit and generate the configuration bits for that unit and its multiplexers. If there

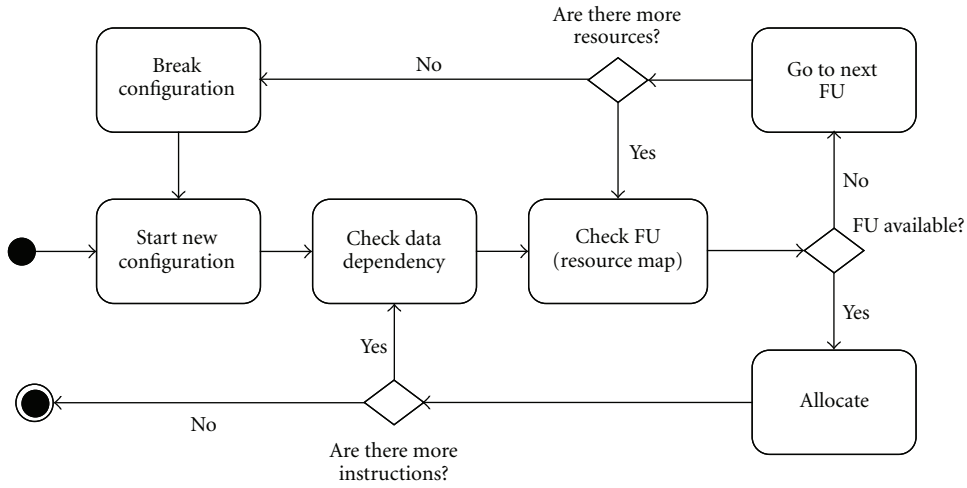


Figure 5: Configuration generation algorithm implemented in Binary Translator (defect-free approach).

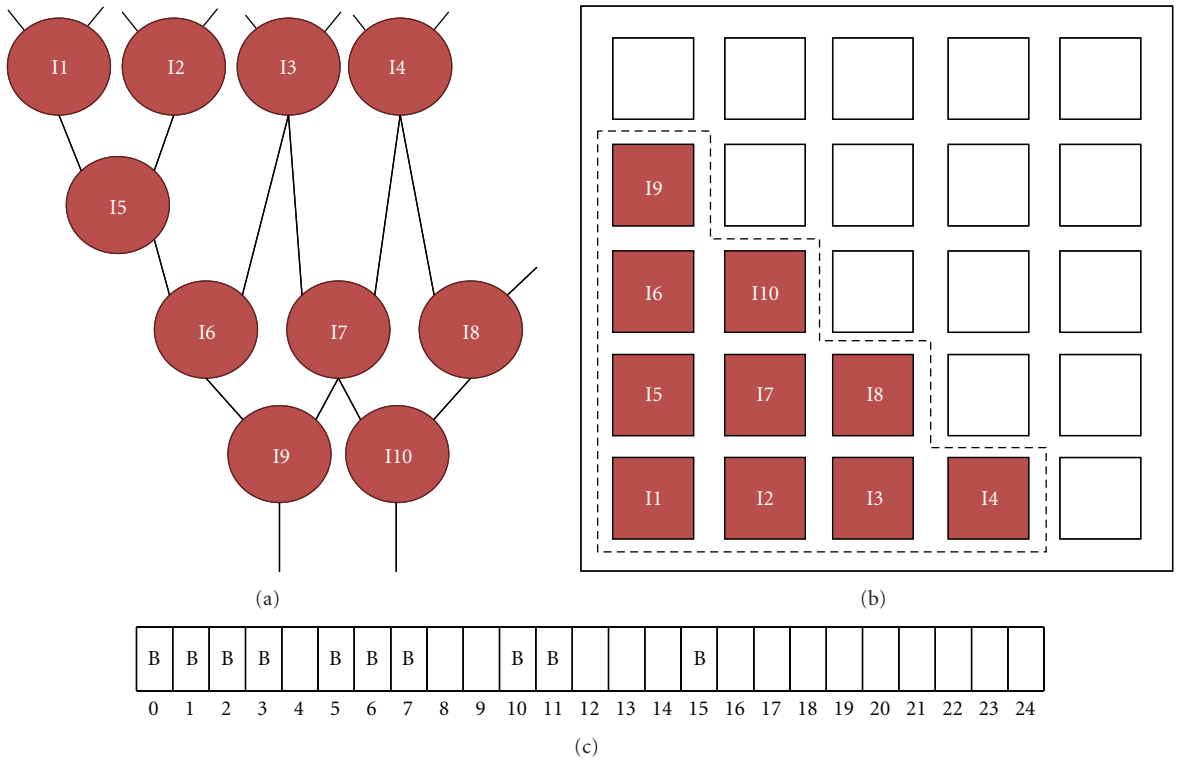


Figure 6: Defect-free resource allocation: (a) data dependency graph, (b) FAT-array resource allocation based on data dependency, and (c) allocated units set as busy in resource map.

are no more available units, then BT breaks the configuration and starts a new one with all units set as free. Figure 5 presents an activity diagram describing the configuration generation algorithm.

Figure 6 presents an example to illustrate how the resources are allocated using the resource map. Figure 6(a) presents the data dependency graph of the instruction sequence mapped to the FAT-array. Figure 6(b) represents the FAT-array with 25 identical functional units, and Figure 6(c) presents the resource map. For clarity's sake, in this example we are considering only arithmetic and logic units. As can be observed, the resource map has 25 positions, each one representing one resource of the array.

As already explained, the parallel instructions are placed in the same row of the FAT-array, and the dependent instructions, which must be executed sequentially, are placed in different rows. To select the functional units to execute the instructions, the mechanism starts by allocating the first available unit (bottom-left). The next instructions are placed according to data dependency and resource availability.

Based on Figure 6(a) the instructions 1, 2, 3, and 4 do not have data dependency among each other; hence they are all placed in the same row of Figure 6(b). On the other hand, instruction 5 is dependent on the result of instructions 1 and 2. Hence, instruction 5 is placed in the first available unit of the second row. Continuing the allocation, the inputs of instruction 6 are the outputs of instructions 3 and 5. Since instruction 5 is in second row, instruction 6 must be placed in third row. Instructions 7 and 8 depend on instructions 3 and 4 that are placed in the first row. Thus, they can be placed in second row. Moreover, since instruction 9 depends on instructions 6 and 7 that are in the third and second rows, respectively, it must be placed in fourth row. Finally, instruction 10 depends on instructions 7 and 8 placed in second row; hence it must be placed in the third row. Therefore, as one can observe in Figure 6(c), the positions of the resource map that correspond to the allocated functional units are set as busy. In this example all the positions are already set as busy but in real execution the positions are filled one per time during run-time.

The approach to avoid and replace defective resources is implemented in the resource allocation step. Section 3 details this approach.

2.2. FAT-Array Reconfiguration and Execution. While BT generates and stores the configuration, the processor continues its execution. Next time a PC from a configuration is found, the processor changes to a halt stage, the respective configuration is loaded from the context memory, and the FAT-array's datapath is reconfigured. Moreover all input data are fetched. Finally, the configuration is executed and the registers and memory positions are written back.

It is important to highlight that the overhead introduced by the FAT-array reconfiguration and data access are amortized by the acceleration achieved by the FAT-array. Moreover, as mentioned before, the configuration generation does not impose any overhead. More details about the reconfiguration process can be found in [12].

3. Defect Tolerance

3.1. Defect Tolerance Approach. Reconfigurable architectures are strong candidates to defect tolerance. Since they consist essentially of identical functional elements, this regularity can be exploited as spare-parts. This is the same approach used in memory devices and has demonstrated to be very efficient [14]. Furthermore, the reconfiguration capability can be exploited to change the resources allocation based on the defective and operational resources.

In addition, dynamic reconfiguration can be used to avoid defective resources and generate new configuration at run-time. Thus, there is no performance penalty caused by the allocation process, nor extra fabrication steps are required to correct each circuit.

Finally, as it will be shown, the capability of adaptation according to the application can be exploited to amortize the performance degradation caused by the replacement of defective resources by working ones.

Since the defect tolerance approach presented in this paper handles only defects generated in the manufacturing process, the information about the defective units is generated before the FAT-array starts its execution, by some classical testing techniques. Therefore, the solution to provide defect tolerance is transparent to the configuration generation.

To demonstrate the approach to tolerate defective functional units Figure 7 presents two examples based on the example of Figure 6.

In a defective FAT-array, the allocation algorithm is exactly like the one described in the example of Figure 6. The only difference is that to allocate only the resources that are effectively able to work, before the FAT-array starts and after the traditional testing steps, all the defective units are set as permanently busy in the resource map, like if they had been previously allocated. With this approach, no modification in the binary translation algorithm itself is necessary.

Figures 7(a) and 7(b) demonstrate the resource allocation considering defective functional units. In Figure 7(a) the configuration mechanism placed the instruction in the first available unit, which in this case corresponds to the second functional unit of the first row. Since the first row still has available resources to place the four instructions, the FAT-array sustains its execution time. In this case the presence of a defective functional unit does not affect the performance.

Figure 7(b) illustrates an example where defective functional units affect the performance of the FAT-array. In this example, the first row has only three available functional units. In this case, when there are not enough resources in one row, the instructions are placed in the next row, and all the data-dependent instructions must be moved upwards. In Figure 7(b), instruction 5 is dependent on instruction 4. Hence, instruction 5 was placed in the next row, and the same happened with other instructions (6 to 10). In this example, because of the defective units it was necessary to use one more row of the FAT-array. Since the sequential path of the FAT-array flows from the bottom row to the top row, the use of one more row leads to the increase of execution time, consequently affecting performance. Figures 7(c) and 7(d) illustrate the resource map of both Figures 7(a) and 7(b) examples, respectively. As one can observe, the defective units have their positions in the resource maps set as busy.

To avoid defective multiplexers from the interconnection model of Figure 3 the strategy can be different depending on which group of multiplexer is affected.

In case of a defective input multiplexer, one of the inputs of the respective functional unit will have invalid data; consequently the result of the operation will be incorrect. Therefore, to keep the defect tolerance approach simple and avoid introduce overhead to the configuration mechanism, the strategy is to consider the multiplexer and its respective functional unit as defectives and place the instruction in the next available functional unit. Figure 8 illustrates the approach to tolerate defective input multiplexers.

As in the case of functional units, it is assumed that all defective multiplexers are detected before BT starts. According to Figure 8, the defective multiplexer invalidates the FU input (dashed line) and this invalidates the functional unit

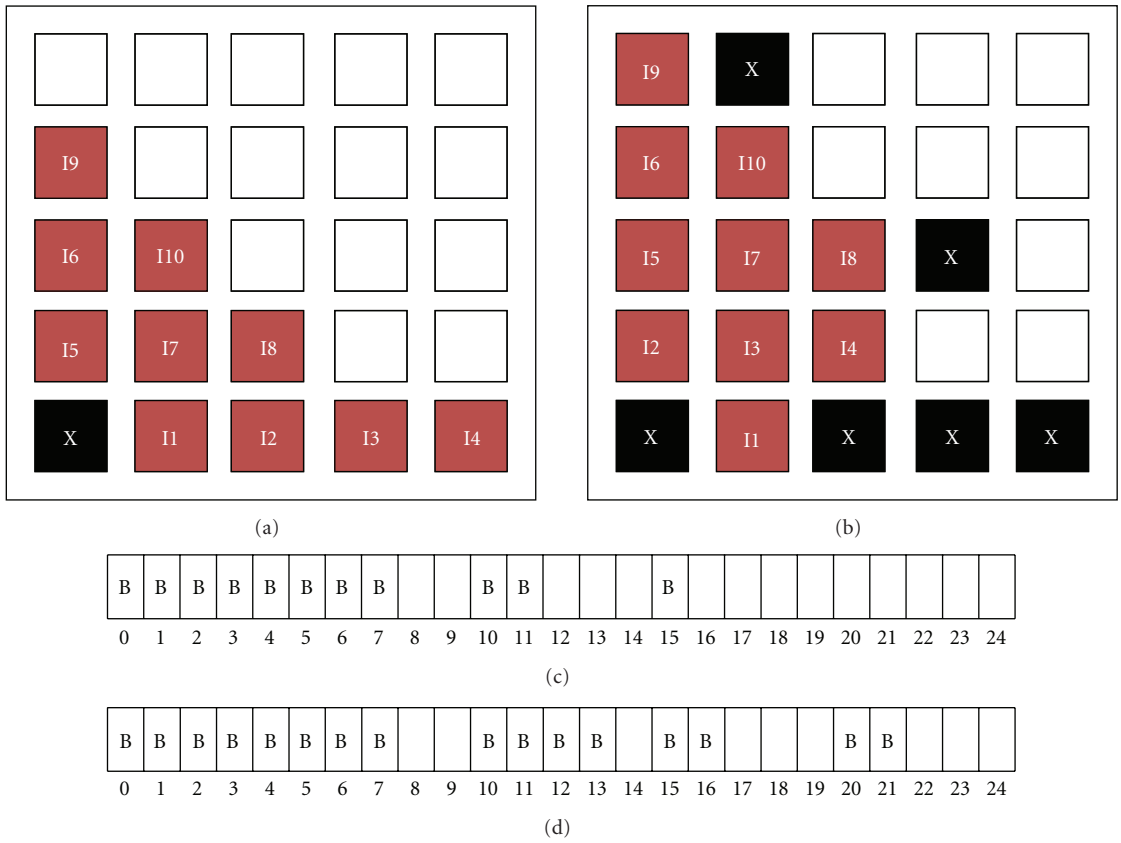


Figure 7: Resource allocation considering defective functional units: (a) not affecting performance, (b) affecting performance, (c) resource map of example (a), and (d) resource map of example (b).

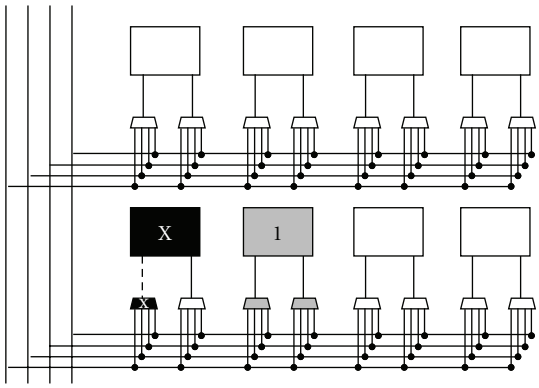


Figure 8: Defective input multiplexer tolerance approach—the defective input multiplexer invalidates the functional unit.

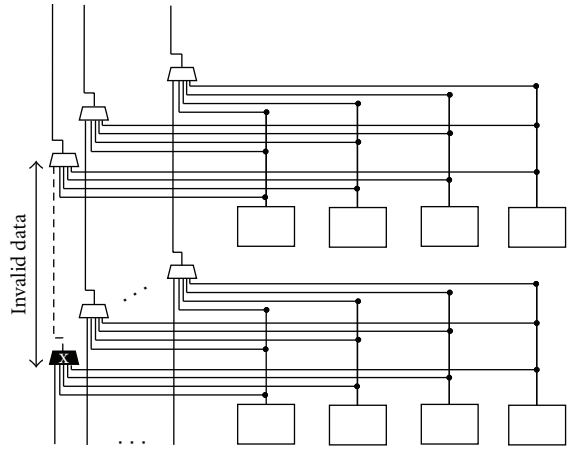


Figure 9: Example of a defective output multiplexer—the defective output multiplexer invalidates the data in the respective context line.

execution. Therefore, the approach is to avoid this FU and to place the instruction in the next available FU (grey modules).

On the other hand, a defective output multiplexer invalidates part of the respective context line. Figure 9 illustrates a defective output multiplexer. For clarity's sake the input multiplexers were omitted. As can be observed in Figure 9, the data remain invalid from defective output multiplexer until the next valid output. The dashed line indicates that this part of the context line has invalid data. The context line will have valid data again when an instruction placed in any row, positioned after the defective multiplexer, writes valid data in this context line.

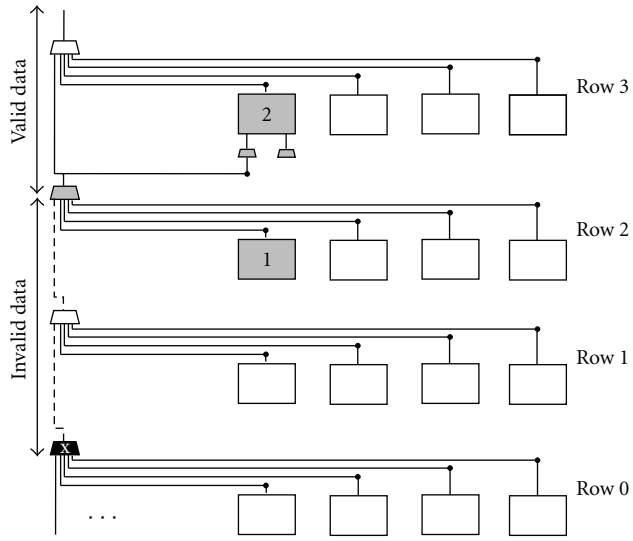


Figure 10: Defective output multiplexer/reading case solution—since instruction 1 has written in context line after defective multiplexer (after row 0) and before instruction 2 reads from this context line (before row 3), then, instruction 2 can read the correct data.

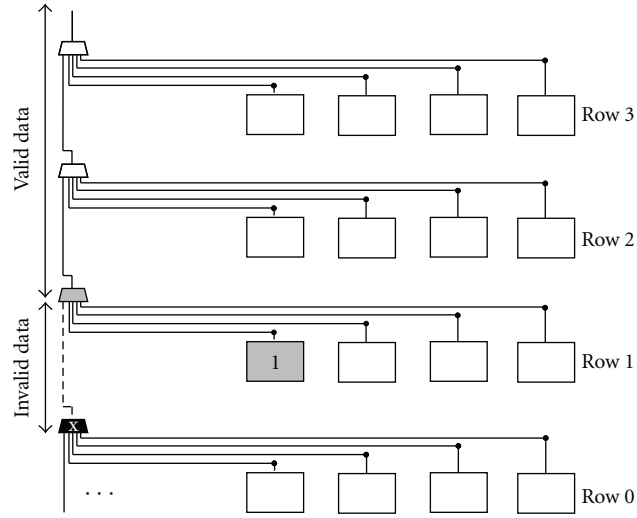


Figure 11: Defective output multiplexer/writing case solution—instruction 1 is placed in the next available row with nondefective output multiplexer (row 1).

Therefore, if an output multiplexer is defective, it is necessary to evaluate if the functional unit needs to read or write in this multiplexer.

When a functional unit needs to read data from a context line, and this same context line has a defective output multiplexer in any previous row, BT must check whether the context line is valid or not. Figure 10 presents an example to illustrate this situation, called reading case.

In the example of Figure 10 instruction 2 that reads from the context line must be placed in row 3. However, the output multiplexer from row 0 is defective. Before placing the instruction in row 3, BT must check if any instruction placed in the rows between row 3 and row 0 wrote in this context line. From Figure 10 it can be observed that instruction 1 placed in row 2 wrote in the context line making this line valid again. Thus, the instruction 2 can be placed in row 3 because it will use the output data from row 2.

In case there is no FU that writes in the context line, this context line remains invalid from the defective output multiplexer until the last row of the array. Thus, the instruction cannot be placed in any functional unit and the configuration must be broken in two configurations. One configuration includes all the instructions that can be placed before the defective multiplexer position and the other configuration will have the other instructions placed in a completely different manner, avoiding the defective multiplexer. Since resource allocation is performed dynamically, there is no predefined order to store data in the context registers. This is the reason that the second configuration can be generated avoiding the defective multiplexer.

The other defect tolerance solution to cope with output defective multiplexers is used when a functional unit needs to write in a context line and the output multiplexer is defective. This case is called writing case and it is illustrated in Figure 11.

The solution to tolerate defective output multiplexers in the writing case consists in placing the instruction in the next available FU. However, since each row has only one output multiplexer for each context line, the strategy in this case is to place the instruction in the next available FU of the next available row. Furthermore, it is also necessary to check if the output multiplexer from the next row is operational. Thus, the instruction can be placed in the next row (as demonstrated in Figure 11) or in any other row with defect-free multiplexer. For example, if the multiplexer from row 1 was also defective, the instruction would be placed in the row 2 and all the instructions dependent on this instruction would be placed from row 3 upwards.

Figure 12 presents the activity diagram summarizing the configuration generation algorithm including the defective unit management. The algorithm is the same presented in Figure 5 with additional steps (grey boxes).

As can be observed in the diagram, the additional steps are responsible for managing only the defective output multiplexers. This is the only case that requires modifications in the algorithm. Moreover, the information about defective functional units and input multiplexers is transparent to BT. This information is translated into the resources map that indicates which functional unit is available and which one is busy. Therefore, the FU that cannot be allocated due to defects is permanently set as busy.

Although some modifications are required to the defective output multiplexer approach, the new steps included in BT are part of the same algorithm already implemented and do not increase the number of cycles required to generate the configuration. Therefore, BT continues working

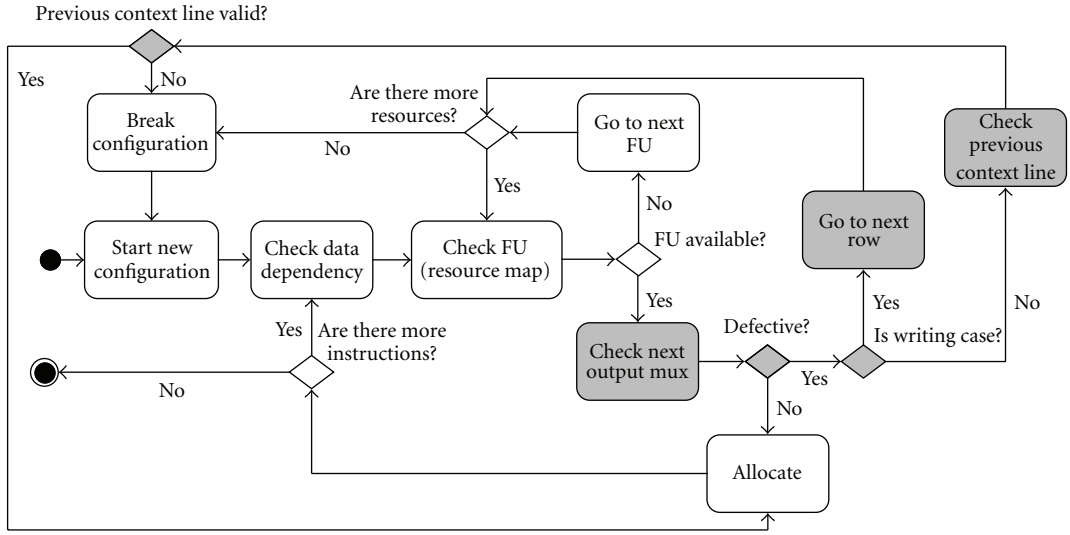


Figure 12: Configuration generation algorithm implemented in BT considering defective units—same steps of defect-free resource allocation (Figure 5) with additional steps for managing defective output multiplexers.

in parallel with the processor's pipeline even with the defect tolerance approach. More important, since BT works in parallel to the processor's pipeline, no performance overhead is incurred while making the binary translation, since it takes fewer cycles than the processor's pipeline, even when defect tolerance is in place.

The defect tolerance approach for functional units and interconnection model was already proposed in [15], where more details about the defect tolerance of interconnection model and experimental results can be found. These details are not in the scope of this paper, since the main focus of the current paper is to demonstrate how the FAT-array with the defect tolerance approach presented in [15] can be an efficient alternative to homogeneous multicores under high defect rates.

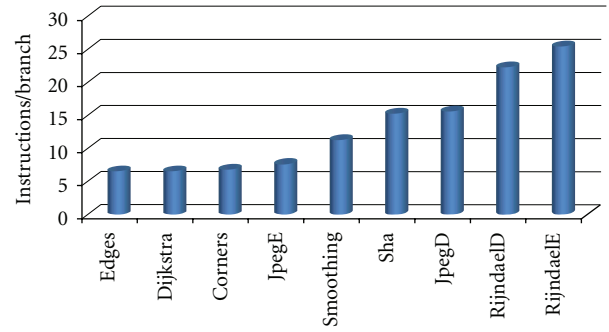


Figure 13: Average of number of instructions executed per branch—*edges* is the most control-flow application and *rijndaelE* is the most data-flow application.

3.2. Experimental Results. To evaluate the proposed approach and its impact on performance, we have implemented the FAT-array in an architectural simulator that provides the MIPS R3000 execution trace. Furthermore, we have chosen a software workload that reflects an embedded system environment. The selected workload is a subset of MiBench suite [16] and covers a wide range of application behaviors. Figure 13 presents the average number of instructions executed between two branch instructions, demonstrating the heterogeneous behavior of the applications. This metric is correlated to the amount of instruction level parallelism available. Therefore, according to Figure 13, *edges* is the most control-flow application, presenting, on average, five operations per branch, and *rijndaelE* is the most data-flow one, with almost 25 instructions per branch.

To include defects in the FAT-array a tool was implemented to randomly select functional and interconnection units as defective, based on several different defect rates. The tool's input is the information about the amount of resources available in the array and its sequential and parallel distribution, as well as the defect rate. Based on this, the tool's output has the same resources information, but now with the randomly selected functional units set as busy and some multiplexers set as defective. This information was used as input to the architectural simulator. In this study we used five different defect rates (0.01%, 0.1%, 1%, 10%, and 20%), and the reference design was the defect-free FAT-array.

Since the functional unit selected as defective must be eliminated from the resource pool, this can affect the performance achieved by the execution of each application. To ensure that the achieved performance was not influenced by one specific defect, we performed several simulations using the same array structure and defect rates, but changing the random defects selection. Thus, the performance results of each set of application, array, and defect rate are actually an average of the performance results of all the simulations that used the same set.

The size of the FAT-array was based on several studies varying the amount of functional units and their parallel and sequential distribution. The studies considered large

arrays with thousands of functional units and also small ones with only dozens functional units. The chosen FAT-array is a middle-term of the studied architectures. It contains 512 functional units (384 ALUS, 96 load/stores, and 32 multipliers). The area of this architecture is equivalent to 10 MIPS R3000.

It is important to highlight that despite the performance degradation presented by the FAT-array system under a 20% defect rate, the performance was still superior to the standalone processor's performance. Hence, Figure 14 presents the speedup degradation of the FAT-array system, instead of the performance degradation.

According to Figure 14, the highest acceleration penalty was presented in the execution of *jpegE*, with 6.5% of speedup reduction under a 20% defect rate. Nevertheless, the FAT-array system is still 2.4 times faster than the standalone processor.

The mean speedup achieved by the defect-free FAT-array in the execution of MiBench applications was 2.6 times. Under a 20% defect rate the mean speedup degraded to 2.5 times. This is less than 4% of speedup degradation.

These results demonstrate that even under a 20% defect rate, the FAT-array combined with the on-line reconfiguration mechanism is capable of not only ensuring that the system remains working but also accelerating the execution when compared to the original MIPS R3000 processor.

3.3. Considerations about Power and Energy Consumption of FAT-Array System. Although technology scaling increases integration capability and consequently performance, another consequence of miniaturization process is the increase of leakage power [17]. According to Sery et al. [18], the leakage power is around 40% of the total power in today's high-performance microprocessors.

Many solutions to increase power efficiency of reduced feature size devices consist in scaling down the supply voltage. However, in order to maintain performance, it is also necessary to reduce the transistor threshold voltage, which in turn increases the subthreshold leakage current. Due to the exponential relationship between the transistor threshold voltage and the subthreshold leakage current, the consequence is the increase of leakage power [19].

To control leakage power the solutions rely on static and dynamic techniques. Static techniques work during the circuit design phase and do not change during operation. On the other hand, dynamic techniques work when the circuit is in idle or in the standby state. One of the most effective dynamic techniques is power-gating. This technique uses sleep transistors to shut down the power supply of parts of the design that are in the idle or standby mode [20].

To cope with power dissipation in the FAT-array architecture, Pereira and Carro [21] propose the inclusion of sleep transistors in each functional unit of the reconfigurable fabric. The approach consists in using the Binary Translator to control the sleep transistors and, at run-time, shut down the idle and defective functional units.

To validate this approach, Pereira and Carro [21] present simulation results evaluating energy consumption of the reconfigurable system with and without sleep transistors. Therefore, a comparison among the MIPS R3000, the full reconfigurable system (without sleep transistors), and the reconfigurable system with sleep transistors in each functional unit is presented. The results demonstrate that the execution of MiBench benchmarks on the reconfigurable system without sleep transistor resulted in an average of 59% of energy saving compared to the execution on MIPS R3000. Moreover, an average of 32% additional energy saving is obtained by implementing sleep transistors in each functional unit.

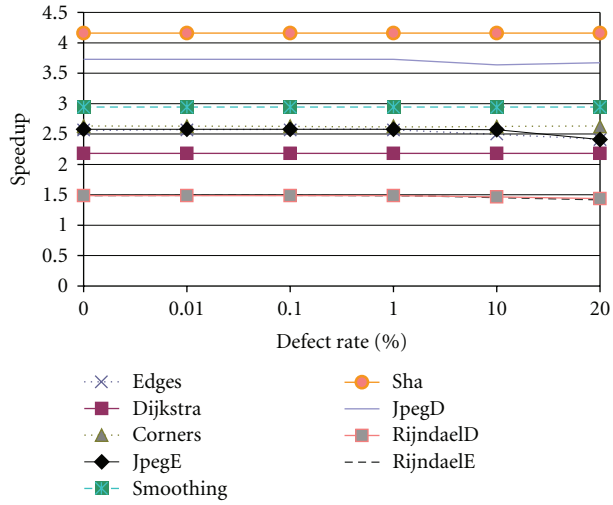


Figure 14: Speedup degradation of the FAT-array system under different defect rates—under 20% defect rate *jpegE* presents the highest speedup degradation with 6.5% of speedup reduction.

It is important to highlight that to control the sleep transistors the Binary Translator has to generate a few more bits to the configuration. Therefore, no significant performance overhead is introduced in the Binary Translator to control the transistors. Moreover, the total area overhead corresponds to two transistors for each functional unit.

4. Adaptive System X Homogeneous MPSoC

4.1. Area and Performance. To demonstrate the efficiency of the proposed architecture this section presents a comparison between the adaptive system and a homogeneous MPSoC with the same area.

As mentioned before, the area of the FAT-array system is equivalent to 10 MIPS R3000 processors, including data and instruction caches. Furthermore, the mean speedup achieved by the reconfigurable system is 2.6 times for the MiBench suite.

The homogeneous MPSoC used to compare area and performance consists of ten MIPS R3000. In this analysis the communication and memory access overheads are not considered. Although the interprocessors communication is not considered, its impact would certainly be higher in the case of the MPSoC; hence all presented results are somewhat favoring the MPSoC.

Table 1: Acceleration as a function of f, $n = 10$.

f	Speedup
0.10	1.099
0.15	1.156
0.20	1.220
0.25	1.290
0.30	1.370
0.35	1.460
0.40	1.563
0.45	1.681
0.50	1.818
0.55	1.980
0.60	2.174
0.65	2.410
0.70	**2.703**
0.75	3.077
0.80	3.571
0.85	4.255
0.90	5.263
0.95	6.897
0.99	9.174
1.00	10.000

As mentioned before, according to Amdahl's law, the speedup achieved by the MPSoC is limited to the execution time of the sequential portion of the application. Equation (1) repeats Amdahl's law for parallel systems:

$$\text{Speedup}(f, n) = \frac{1}{(1 - f) + (f/n)}, \quad (1)$$

where f is the fraction of the application that can be parallelized and n is the number of cores.

Since the MPSoC has ten cores, by varying f and fixing n in 10 (to have the same area of the FAT-array and hence normalize results by area), from Amdahl's law we obtain the results presented in Table 1, where one can see the speedup as a function of f in (1), the part that can be parallelized.

Since communication and memory accesses overheads are not considered, with 10 cores it is possible to achieve a speedup of 10 times if 100% of the application is parallelized.

According to Table 1 it is necessary that 70% of the application be parallelized to achieve a speedup of 2.7 times, which is around the acceleration obtained by the FAT-array system.

Now, one can fix the speedup and vary f to find the number of processors to achieve the required speedup. From (1), varying f from 0.1 to 1 we have that when $f \geq 0.65$, we can achieve the speedup of 2.6. When $f < 0.65$, it is not possible to find a number of cores to achieve an acceleration of 2.6 times. Thus, even with hundreds or thousands of cores, if the application has less than 65% of parallelism, it will never achieve the speedup of 2.6, the same of the FAT-array. Nevertheless, with 65% there would be necessary 19 cores to achieve speedup of 2.6 times, as shown in Figure 15.

Table 2: Sequential acceleration as a function of f.

f	Speedup	AP	AS
0.1	2.60	10	2.402
0.2	2.60	10	2.194
0.3	2.60	10	1.974
0.4	2.60	10	1.741
0.5	2.60	10	1.494
0.6	2.60	10	1.232
0.7	2.60	10	0.954
0.9	2.60	10	0.339
0.99	2.60	10	0.035
1	**2.60**	**10**	**0.000**

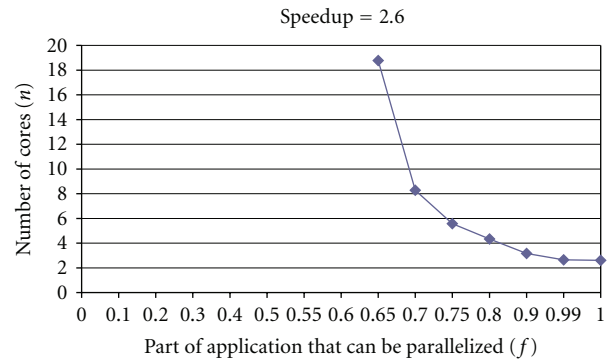


Figure 15: Number of cores as a function of f—according to Amdahl's law, to achieve speedup of 2.6 it is required at least 65% of application parallelism and 19 cores.

One solution to cope with this is to improve the homogeneous core's performance to increase the speedup of the sequential execution. Therefore, one can rewrite Amdahl's law to take this into account, as it is demonstrated in (2). This solution was discussed in [2], where the authors presented the possible solutions to increase performance of a homogeneous MPSoC. They conclude that more investment should be done to increase the individual core performance even at high cost:

$$\text{Speedup}(f, \text{AP}, \text{AS}) = \frac{1}{((1 - f)/\text{AS}) + (f/\text{AP})}. \quad (2)$$

Equation (2) is an extension of Amdahl's law and reflects the idea of improving the MPSoC's overall performance by increasing core performance through acceleration of sequential portions. In (2), AS is the speedup of the sequential portion and AP is the speedup of the parallel portion. Table 2 presents values for AS, fixing the speedup in 2.6 (acceleration given by the FAT-array) and AP in 10 (homogeneous multicore and FAT-array have the same area), while varying f. As one can see in Table 2, only when $f = 100\%$ that AS $= 0$, which means that this is the only case that does not require sequential acceleration. This acceleration cannot be achieved by the homogeneous MPSoC; however as explained in Section 2, the FAT-array can accelerate sequential portions of the application.

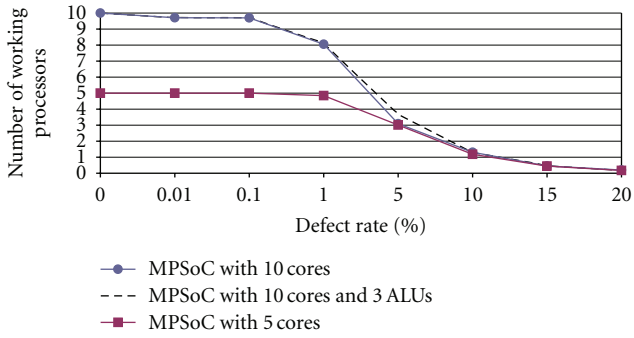


Figure 16: Defects simulation in the MPSoC—simulations performed on a 10-core MPSoC and no fault tolerance solution, on a 10-core MPSoC and 3 ALUs per core, and on a 5-core MPSoC with two processors per core. All cores fail under 15% defect rate.

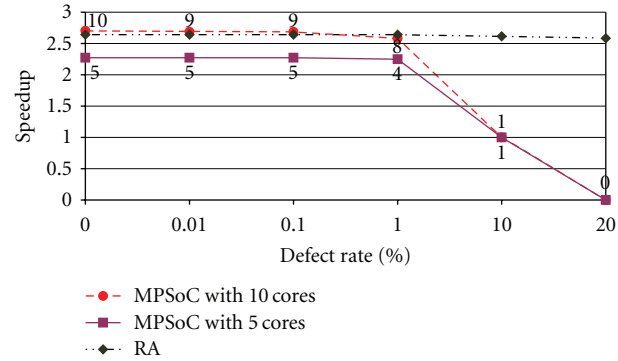


Figure 17: Performance degradation of the MPSoC—considering an application with 70% of available parallelism. Both MPSoC solutions, without fault tolerance approach and with two processors per core, present sequential execution under 10% defect rate and completely fail under a 20% defect rate.

The next section presents a comparison between the MPSoC and the FAT-array considering the fault tolerance capability. The results are normalized by area and speedup.

4.2. Defect Tolerance. This section presents an analysis of performance degradation of both, FAT-array and MPSoC, caused by the presence of defects. This analysis was done through a performance simulation varying the defect rate.

To simulate the defects in both, MPSoC and FAT-array, a tool was implemented to randomly insert defects in both architectures. To ensure that the defect position was not affecting the results, thousands of simulations were performed and in each simulation a new random set of defects was generated. Moreover, the defects generated had the same size (granularity) to both MPSoC and reconfigurable architecture.

In the first analysis we normalized FAT-array and MPSoC by area. In the second analysis we increased the numbers of cores of the MPSoC to evaluate the tradeoff between area and fault tolerance capability.

4.2.1. MPSoC and FAT-Array with Same Area. Figure 16 illustrates the number of cores that remain operating in function of the defect rate in three different studies. The first analysis was performed in a homogeneous MPSoC with 10 MIPS R3000 processors without any fault tolerance approach. According to the results, when the defect rate is 15% or higher, less than 1 core is operating; that is, the whole MPSoC system fails under a 15% defect rate.

The second and third analyses were performed considering that the MPSoC has some kind of fault tolerance solution implemented. In the second analysis, the fault tolerance solution consists in replicating the processor in each core. In this case, instead of having 10 cores with 10 processors, the MPSoC has 5 cores with 2 processors in each core. The second processor works as spare that is used only when the first processor fails. This solution was proposed for two main reasons. First there is no increase in area. Thus, the MPSoC still has the same area of the FAT-array. Second, even with half of the number of cores, the MPSoC still presents higher speedup than the array when the application presents 100% of parallelism.

The solution proposed in the third analysis is also based on hardware redundancy. However, in this case instead of replicating the whole processor, only critical components of the processor are replicated, for example, the arithmetic and logic unit. Therefore, this solution considers that each processor has 3 arithmetic and logic units (ALUs), where 2 ALUs are used as spare. This solution presents lower area cost compared to the solution of the second analysis. However, it can be more complex to implement, since each processor must have an extra unit to implement the fault tolerance approach.

As can be observed in Figure 16, in both second and third analyses, under a 15% defect rate all cores failed. This means that even with fault tolerance solutions, the MPSoC tends to fail completely under high defect rates.

Figure 17 presents the performance degradation of the MPSoC when the number of cores is reduced due to the presence of defects. To obtain the speedup there was used Amdahl's law represented in (1), the MPSoC with ten cores (without fault tolerance), and the one with 5 cores and 2 processors per core. Again, we considered no communication costs, and hence real results tend to be worse. The chart also presents the mean performance degradation of the FAT-array system in the execution of MiBench applications.

This analysis was performed using $f = 0.70$ (the portion of the application that can be parallelized). This number was used because according to Table 1, the speedup achieved by the MPSoC when 70% of the application can be parallelized approaches the speedup achieved by the FAT-array system. The numbers next to the dots in the chart represent the amount of cores that are still working in the MPSoC in function of the defect rate.

As can be observed in Figure 17, the performance of the MPSoC degraded faster than the degradation presented by the FAT-array system, even when the MPSoC presented

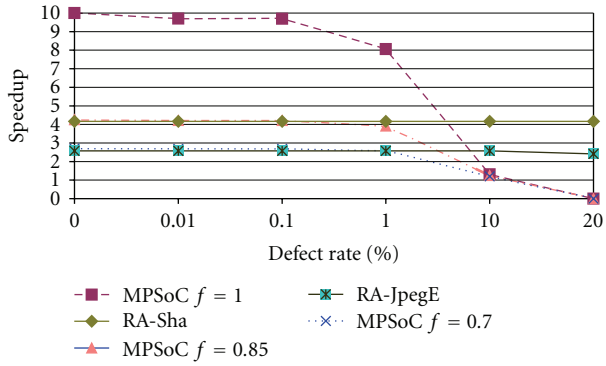


Figure 18: Graceful degradation of *sha* and *jpegE* applications—comparison performed considering applications with different degrees of parallelism running on the 10-core MPSoC without fault tolerance approach.

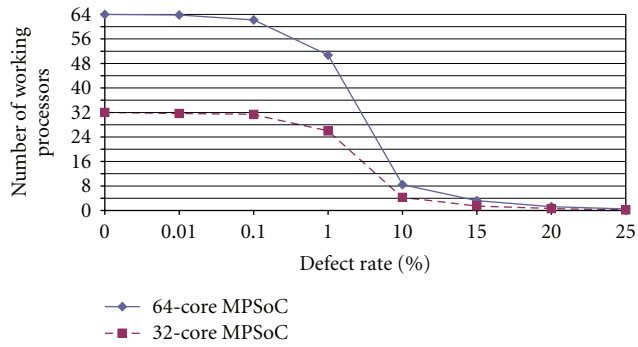


Figure 19: Defect simulation in 32-core and 64-core MPSoC—under 20% defect rate all cores fail in the 32-core MPSoC and only one core remains in the 64-core MPSoC.

higher speedup in a defect-free situation (10-core MPSoC). This happens because when a defect is placed in any part of the processor, the affected processor cannot execute properly. Moreover, even in the MPSoC with five cores and one spare processor in each core the system still degraded faster than the FAT-array. This means that a much more sophisticated approach is necessary to increase fault tolerance of the MPSoC. Nevertheless, this does not guarantee that the system works under high fault rates expected to future technologies. Besides, the more sophisticated is the fault tolerance approach, the more complex it will be, possibly introducing a high area, power, and/or performance overheads.

On the other hand, when a defect is placed in any functional unit or interconnection element of the FAT-array, the run-time mechanism selects another unit (or element) to replace the defective one. According to Figure 17 the execution of MiBench applications by the FAT-array system presented less than 4% of speedup degradation even under a 20% defect rate.

It is important to highlight that in these analyses there was not considered the impact on area and performance that the implementation of the fault tolerance strategies in the MPSoC should introduce. Again the presented results are somewhat favoring the MPSoC. Moreover, the choice of these fault tolerance solutions was based on the idea of causing the minimal impact on the area of the system to maintain both FAT-array and MPSoC equivalent in area.

Figure 18 presents the graceful degradation of applications *sha* and *jpegE*. These applications were selected because the first one achieved the highest speedup by the FAT-array system among all the applications from the MiBench suite and the second presented the highest speedup degradation.

According to Figure 18 the execution of application *sha* by the FAT-array system presented less than 1% of speedup degradation even under a 20% defect rate. On the other hand, even considering that the application was 100% parallelized ($f = 1$), with an initial acceleration higher than the one achieved by the FAT-array system, the 10-core MPSoC stopped working under a 20% defect rate. This same behavior was observed when the amount of parallelism available was reduced ($f = 0.85$ and $f = 0.70$). In these cases, not only did the whole system stop working under a 20% defect rate but also the initial acceleration was equal and lower, respectively, than the one achieved by the FAT-array system.

Moreover, as can be observed in Figure 18, the same behavior was detected in *jpegE* results. However, in this application execution the MPSoC presented higher speedup with $f = 0.85$ than the FAT-array, which rapidly decreased to 0 with the defect rate higher than 1%. On the other hand, the FAT-array sustained acceleration even under a 20% defect rate that presented a speedup degradation of 6.5%.

4.2.2. Increase MPSoC Core Number. Since the FAT-array consists in a large amount of identical functional units that can be easily replaced, the same idea was proposed to the MPSoC: increase the number of cores to increase reliability. Thus, this solution consists in adding more cores to the MPSoC to allow software execution under higher defect rates.

As one can observe in Figure 19, the MPSoCs with 32 and 64 cores still execute under a 15% defect rate. However, in case of 32 cores the execution is completely sequential (one core left under 15% defect rate), and in case of 64 cores, the MPSoC has only 3 cores left under a 15% defect rate. Moreover, under a 20% defect rate the 32-core MPSoC completely fails and the 64-core MPSoC presents a completely sequential execution with only one core left.

The speedup results presented in Figure 20 also demonstrate the rapid decrease in the MPSoC speedup with 32 and 64 cores while the FAT-array sustains acceleration in both *sha* and *jpegE* even under a 20% defect rate.

Based on this result, one can conclude that simply replicating the cores it is not enough to increase the defect tolerance of the system to tolerate high defect rates that new technologies should introduce. Moreover, adding a defect tolerance approach can be costly in area and performance.

To evaluate the area of the system with the reconfigurable architecture used in this work we estimated the size of

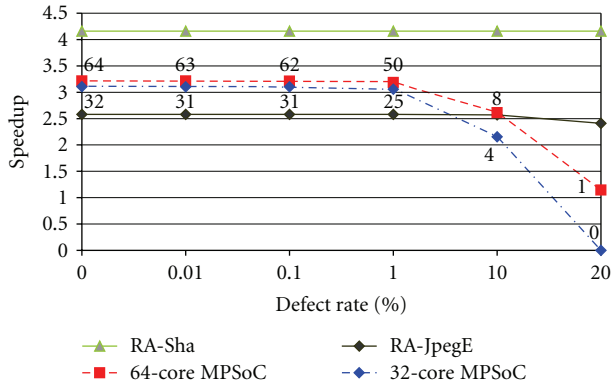


Figure 20: Graceful degradation of 32-core and 64-core MPSoC—under different defect rates and comparing with FAT-array graceful degradation when executing *sha* and *jpegE* applications.

the chip based on the gates number presented by the LeonardoSpectrum from Mentor Graphics [22]. Table 3 presents the chip size (in mm^2) in the 90 nm technology.

Since the reconfigurable fabric is highly tolerant to defects, as demonstrated in the analysis presented above, we also estimated the area of the FAT-array system in scaled technologies that present high defect rates, using the transistor density presented in ITRS roadmap [23]. In this case, since the processor is not tolerant to defects, shrinking its area would increase the probability of this unit presenting defects, and consequently, precluding the operation of the system. Thus, the processor remained the same size in all technologies. The same approach was used with BT, once it is a critical part of the system and it is not tolerant to defects. Therefore, only the reconfigurable fabric was scaled to reduced feature size technologies. Table 3 presents the area estimated according to the technology (in mm^2). It is clear that with technology evolution the area overhead of the reconfigurable fabric gets much smaller, and by using the proposed approach, its reliability does not get compromised by the scaling process.

According to Table 3, the 90 nm system is around 7 times larger than the 11 nm system. This means that if one uses the area occupied by the 90 nm system, it is possible to have 7 cores each one consisting of an MIPS R3000, a BT mechanism and a FAT-array. The combination of MPSoC and reconfigurable architecture allows task level parallelism exploitation through the use of several cores and instruction level parallelism by the reconfigurable architecture. Moreover, sequential arithmetic and logic units can also be accelerated by the FAT-array. This is a very recent research topic. Rutzig et al. [3] proposes an MPSoC architecture that combines general-purpose processors and reconfigurable architectures. However, the study of fault tolerance in this system was not addressed yet.

The analyses presented in this section demonstrate that to future technologies with high defect rates, homogeneous MPSoCs may not be the most appropriate solution. The main reasons are the fact that a defect in any part of the processor invalidates this processor. Thus, the higher

Table 3: Estimated chip size (mm^2).

	90 nm	32 nm	22 nm	11 nm
MIPS R3000	0.2162	0.2162	0.2162	0.2162
BT	0.1216	0.1216	0.1216	0.1216
FAT-array	2.1619	0.0269	0.0135	0.0034
TOTAL	2.4997	0.3647	0.3513	0.3412

is the defect rate, more aggressive is the performance degradation, leading to a completely fail of the system under defect rates already predicted to futures technologies, such as nanowires [6]. Furthermore, solutions to provide fault tolerance in homogeneous MPSoCs under high defect rates can be costly, both in area and performance.

Another disadvantage of homogeneous MPSoCs is the fact that they can only exploit task level parallelism, depending on the parallelism available in each application. Therefore, only a specific application set that is highly parallelized can benefit from the high integration density and consequently the integration of several cores in one chip [24].

There are two main solutions to cope with this. The first one is to use heterogeneous MPSoCs, where each core can accelerate a specific application set [4]. The main problem of this solution is that like the homogeneous multicore, the heterogeneous one must also have some fault tolerance approach to cope with high fault rates, and this can increase area and performance costs.

The other possible solution is to increase the speedup of each core individually. With the improvement of each core it is possible to accelerate sequential portions of code and consequently increase the overall performance of the system. An example of this approach is to change the MIPS R3000 cores for superscalar MIPS R10000 [25]. However, this strategy can result in a significant area increase. According to [3], an MIPS R10000 is almost 29 times larger than the MIPS R3000.

The analyses also demonstrate that the proposed reconfigurable architecture ensures software execution and also accelerates the execution of several applications even under a 20% defect rate. Moreover, the FAT-array system is a heterogeneous solution that accelerates parallel and sequential code. Thanks to this approach, the proposed architecture even exposed to high defect rates predicted to future technologies can still accelerate code, since the parallelism exploitation is not the only way to accelerate execution.

5. Related Work

This section presents some works related to the two main topics of this paper. The first one is fault tolerance approach in reconfigurable architectures. In this context, this section presents some relevant works found in the literature that use reconfigurable architecture as a solution to increase fault tolerance of the systems.

The second topic presented in this section is fault tolerance in MPSoCs. Since this paper proposes to replace a homogeneous multicore for a reconfigurable architecture

as a solution to increase fault tolerance of the system, it is relevant to show that there are many solutions to increase reliability of MPSoCs. However, as mentioned before, the efficient solutions present high overheads, such as increase in area and power and decrease in performance due to the fault tolerance approach; for example, DMR and TMR approaches add at least 2 and 3 times more area and power just to duplicate or triplicate the system, respectively. The area and power consumption are even higher when considering the voters. Moreover, the solutions that focus on a low-cost approach are not as efficient as the more complex ones.

5.1. Fault Tolerant Reconfigurable Architectures. Most works that try to connect reconfiguration with fault tolerance have been developed targeting commercial and fine-grained FPGA devices.

Redundancy exploitation, graceful degradation, and yield enhancement are examples of techniques used in FPGAs, such as the one proposed by Hatori et al. [26]. This work introduced the first fault tolerance solution using redundancy specifically to FPGAs. The authors propose the inclusion of extra rows of logic blocks and extra wiring to be used as spare-parts. Moreover, in this approach reconfiguration is performed during manufacture. Therefore, there is no time overhead due to reconfiguration. However, it tolerates only one fault per row and the strategy consists in eliminating an entire column for one fault. As a consequence, the fault tolerance is low.

Hanchek and Dutt [27] adopt a node-covering method to replace faulty elements. In this approach chains of PLBs (nodes) are created and the last PLB from the chain is used as spare (cover node). When a fault occurs, the node is replaced by shifting all nodes along the row, one position at a time, from the faulty node until the end of the chain. Additional segments (cover segments) are also included to ensure that the moved local routing will be reconnected. Each chain corresponds to a row in an FPGA. Therefore, the approach can tolerate one fault per row with the overhead of one additional column of PLBs.

Lach et al. [28] present an offline approach that partitions the physical design into tiles and replaces the affected tile with a functionally equivalent one. Besides part of the system, each tile contains spare logic and interconnection elements. To replace the tile several precomputed configurations are stored in memory, each one with the spare logic placed in a different position. If a fault occurs, the current configuration from this tile is replaced by a configuration that does not use the faulty resource.

Abramovici et al. [29] propose an online testing, diagnosis, and fault tolerance approach that uses self-testing areas (STARs) of the FPGA while the rest of the device continues its normal operation. In this work the STARs are gradually roved across the FPGA until the entire chip is tested. Although this work proposes a solution that does not require stopping the system, the reconfiguration is performed while the STAR area is nonoperational. Therefore, there is still a performance overhead to reconfigure the parts of the FPGA [30].

Gebelein et al. [31] propose the combination of hardware and software approaches to mitigate radiation effects in FPGAs. At the bottommost FPGA configuration bit layer, the proposed solution consists in combining scrubbing with application of DMR in the CPU. The CPU consists in a soft-core compatible with MIPS-R3000 architecture and instruction set. This solution was implemented in VHDL and tested through live beam tests. Considerations about the use of ECC BRAM blocks and memory refresh scrubber as well as configuration of software layers like operating system and software application to mitigate radiation effects are also presented. Moreover, the reconfiguration capability and area increase due to DMR are used specifically to mitigate radiation effects. They do not allow acceleration of software execution.

Besides the overhead introduced to reconfigure fine-grained FPGAs, some of the presented works involve some form of redesign, which forces the application to stop while reconfiguration is taking place, even if the application is being configured in some other part of the FPGA. Regarding the solutions that use spare-parts, they require the addition of extra hardware or require the user to perform an alternative placement and routing scheme to reconfigure the FPGA. Moreover, solutions that use precomputed configurations are usually limited to a small amount of faults that the approach can cover or require a large amount of different configurations to cope with all the possible faults combination.

The solution presented in this paper manages all these issues during the execution phase, hence it does not require any redesign, and the application is not required to stop. First, the system uses a coarse-grained reconfigurable fabric, which reduces the overhead introduced by the amount of information needed to reconfigure the system. Moreover, to replace defective functional units the same mechanism implemented to dynamically configure the reconfigurable fabric is used; that is, the system is self-adaptive. Thus, it is not necessary to implement several backup algorithms at design time or add extra hardware to specifically perform fault tolerance. Since the reconfiguration is performed at run-time, it is also not necessary to use precomputed configurations as a solution to avoid fault elements. Finally, the fault tolerance approach does not require the addition of spare-parts, since the system will continue its execution, only with less acceleration. The redundancy characteristic of the reconfigurable fabric is used to replace defective elements for operational elements with the cost of a slightly lower acceleration penalty, as demonstrated in last sections.

5.2. Fault Tolerant MPSoCs. Redundancy is the main solution to increase reliability of processors. There are four main redundancy strategies: hardware, software, information, and time redundancy [8]. Since this work proposes a fault tolerance approach through hardware redundancy, the related works presented in this section also use hardware redundancy to increase reliability. To see more details about the other types of redundancy and related work on this topic please refer to [8].

One of the most relevant works in hardware redundancy is DIVA architecture [32]. DIVA uses two processors in the same die to perform fault detection and fault tolerance. The processors are an out-of-order superscalar that executes the leading thread and a simple in-order processor that works as a checker to verify the correctness of all the computations performed by the superscalar. The checker receives all data through the reorder buffer (ROB) of the processor's pipeline. For this reason, the checker does not need to calculate addresses and perform branch prediction, among other tasks, becoming simpler than a processor. When the checker finds inconsistency, it triggers a recovery action. The problem of this approach is the assumption that the checker is error-free, which is acceptable for current technologies but in future technologies this assumption probably cannot be taken. Moreover, the static nature of its hardware prevents it from gaining additional performance when reliability is not warranted.

HP NonStop [10] system was the first commercial system designed to achieve high availability. The main applications of NonStop systems are as follows: credit card authorization, emergency calls, electronic mail, among other applications that require high level of availability, data protection, or scalability [10]. The HP NonStop system was first introduced by former Tandem Computers, Inc in 1976 before it became an HP product [10]. The current system from HP is the NonStop Advanced Architecture (NSAA) that consists in a massively parallel cluster where each independent processor, a 4-way or 8-way SMP Itanium2, runs a copy of the operating system. The system relies on dual or triple modular redundancy (DMR or TMR). According to the authors, the main advantage of the NSAA over previous versions is that the NSAA implements the loose lockstep approach, where each processor can run the same application instruction stream in different clock rates. This allows each processor to do error retries or fix up routine, and to hit or miss caches at different points in time [10]. The NSAA also relies in checksums and error correction codes in network messages and disk data. All implemented techniques cope with transient and permanent faults to provide a correct result. Otherwise the faulty component will stop working and remove itself from the system.

Another fault tolerant approach in multicore systems to cope with permanent faults is the configurable isolation proposed by Aggarwal [11]. This approach consists in splitting the resources in groups, represented by colors, and any fault resource in one group will affect only the cores from that group (with the same color). To avoid that one failure in one group affects resources in the other group, an isolation mechanism is implemented for interconnection, caches, and memory controllers. According to the authors the architecture can be used in different configurations, such as DMR, TMR, or even N-MR configuration.

Many other fault tolerance strategies in processor and multicores architectures can be found in the literature [33–37]. Although some present efficient strategies, such as NonStop [10], and others present low-cost approaches such as DIVA [32], all works that rely on hardware redundancy present area and power overhead. Moreover, many works also present a performance penalty due to the fault tolerance approach. On the other hand, the FAT-array presented in this paper does not present any overhead due to the defect tolerance approach. Moreover, all the redundant hardware included in the system is used to accelerate software execution, and only in case of defects the system can suffer a performance penalty. Nevertheless, as already demonstrated, even under a 20% defect rate the system still remains accelerating.

6. Conclusions

Advances in CMOS technology scaling have increased the integration density, and consequently allowing the inclusion of several cores in one single chip (multicore solutions).

The MPSoC (Multiprocessor System on Chip) architectures allow the acceleration of application execution through task level parallelism exploitation. However, the drawback of this approach is the fact that it is limited to the amount of parallelism available in each application, as demonstrated by Amdahl's law.

One of the solutions to overcome this limit is using heterogeneous MPSoCs, where each core is specialized in different applications set. Another possible solution consists in increasing the speedup of each core individually. These approaches can improve performance but cannot handle high defect rates presented in future technologies.

As a solution to cope with high defect rates and sustain performance this paper presented a run-time reconfigurable architecture to replace homogeneous MPSoCs. The system consists of a coarse-grained reconfigurable array and an on-line mechanism that perform defective functional unit replacement at run-time without the need of extra tools or hardware.

The Fault-Tolerant array (FAT-array) design allows the acceleration of parallel and sequential portions of applications and can be used as a heterogeneous solution to replace the homogeneous MPSoCs and ensure reliability in a highly defective environment.

To validate the proposed approach several simulations were performed to compare the performance degradation of the FAT-array system and the MPSoC using the same defect rates, normalizing the architectures by area and speedup. According to the results, the FAT-array system sustains execution even under a 20% defect rate, while the MPSoC with equivalent area (10-core MPSoC) has all the cores affected under a 15% defect rate. Moreover, analyses considering 32-core and 64-core MPSoCs demonstrated that under a 20% defect rate the former completely fails and the latter presents a sequential execution with only one core left.

Future works include the possibility of coping with transient faults in the FAT-array system.

References

[1] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (MPSoC) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[2] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[3] M. B. Rutzig, F. Madruga, M. A. Alves et al., "TLP and ILP exploitation through a reconfigurable multiprocessor system," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, April 2010.

[4] Samsung Electronics Co., Ltd, "Samsung S5PC100 ARM Cortex A8 based Mobile Application Processor," 2009.

[5] iPhone, http://www.apple.com/iphone/.

[6] A. DeHon and H. Naeimi, "Seven strategies for tolerating highly defective fabrication," *IEEE Design and Test of Computers*, vol. 22, no. 4, pp. 306–315, 2005.

[7] S. Borkar, "Microarchitecture and design challenges for gigascale integration," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO '04)*, p. 3, December 2004.

[8] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*, Morgan Kaufmann, San Francisco, Calif, USA, 2007.

[9] S. K. Shukla and R. I. Bahar, *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[10] D. Bernick, B. Bruckert, P. D. Vigna et al., "NonStop advanced architecture," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 12–21, July 2005.

[11] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*, pp. 470–481, June 2007.

[12] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1208–1213, March 2008.

[13] A. C. S. Beck and L. Carro, "Transparent acceleration of data dependent instructions for general purpose processors," in *Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI-SoC '07)*, pp. 66–71, October 2007.

[14] E. Stott, P. Sedcole, and P. Y. K. Cheung, "Fault tolerant methods for reliability in FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 415–420, September 2008.

[15] M. M. Pereira and L. Carro, "A dynamic reconfiguration approach for accelerating highly defective processors," in *Proceedings of the 17th IFIP/IEEE International Conference on Very Large Scale Integration*, October 2009.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst et al., "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the 4th IEEE International Workshop on Workload Characterization*, pp. 3–14, IEEE Press, December 2001.

[17] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305–327, 2003.

[18] G. Sery, S. Borkar, and V. De, "Life is CMOS: why chase the life after?" in *Proceedings of the 39th Design Automation Conference*, pp. 78–83, June 2002.

[19] J. P. Halter and F. N. Najm, "Gate-level leakage power reduction method for ultra-low-power CMOS circuits," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 475–478, May 1997.

[20] K. Shi and D. Howard, "Challenges in sleep transistor design and implementation in low-power designs," in *Proceedings of the 43rd Annual Conference on Design Automation*, pp. 113–116, ACM Press.

[21] M. M. Pereira and L. Carro, "Dynamically adapted low-energy fault tolerant processors," in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 91–97, IEEE Press, August 2009.

[22] MENTOR GRAPHICS, "LeonardoSpectrum™," http://www.mentor.com/products/fpga/synthesis/leonardo_spectrum/.

[23] ITRS, "International Technology Roadmap for Semiconductors," http://www.itrs.net/reports.html.

[24] K. Olukotun, L. Hammond, and J. Laudon, *Chip Multiprocessor Architecture*, Mark D. Hill, 2006.

[25] K. C. Yeager, "Mips R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, 1996.

[26] F. Hatori, T. Sakurai, K. Nogami et al., "Introducing redundancy in field programmable gate arrays," in *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 711–714, May 1993.

[27] F. Hanchek and S. Dutt, "Node-covering based defect and fault tolerance methods for increased yield in FPGAs," in *Proceedings of the 9th International Conference on VLSI Design*, pp. 225–229, Bangalore, India, January 1996.

[28] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Efficiently supporting fault-tolerance in FPGAs," in *Proceedings of the ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 105–115, Monterey, Calif, USA, February 1998.

[29] M. Abramovici, C. Stroud, C. Hamilton, S. Wijesuriya, and V. Verma, "Using roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications," in *Proceedings of the ITC International Test Conference (ITC'99)*, pp. 973–982, Atlantic City, NJ, USA, September 1999.

[30] J. M. Emmert, C. E. Stroud, J. A. Cheatham et al., "Performance penalty for fault tolerance in roving STARs," in *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL '00)*, pp. 545–554, Villach, Austria, August 2000.

[31] J. Gebelein, H. Engel, and U. Kebschull, "FPGA fault tolerance in radiation susceptible environments," in *Radiation Effects on Components and Systems Conference (RADECS '10)*, Längenfeld, Austria, September 2010.

[32] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '99)*, pp. 196–207, November 1999.

[33] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger, "Exploiting microarchitectural redundancy for defect tolerance," in *Proceedings of the 21st International Conference on Computer Design (ICCD '03)*, pp. 481–488, October 2003.

[34] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin, "Tolerating hard faults in microprocessor array structures," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 51–60, July 2004.

[35] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Exploiting structural duplication for lifetime reliability enhancement," in *Proceedings of the 32nd Interntional Symposium on Computer Architecture (ISCA '05)*, pp. 520–531, June 2005.

[36] D. Sylvester, D. Blaauw, and E. Karl, "ElastIC: an adaptive self-healing architecture for unpredictable silicon," *IEEE Design and Test of Computers*, vol. 23, no. 6, pp. 484–490, 2006.

[37] K. Constantinides, S. Plaza, J. Blome et al., "BulletProof: a defect-tolerant CMP switch architecture," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pp. 3–14, February 2006.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 745147, 14 pages
doi:10.1155/2011/745147

Research Article

An NoC Traffic Compiler for Efficient FPGA Implementation of Sparse Graph-Oriented Workloads

Nachiket Kapre[1] and André Dehon[2]

[1] *Department of Electrical and Electronic Engineering, Imperial College London, London SW7 2AZ, UK*
[2] *Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA 19104, USA*

Correspondence should be addressed to Nachiket Kapre, nachiket@ieee.org

Received 28 August 2010; Accepted 14 December 2010

Academic Editor: Michael Hübner

Parallel graph-oriented applications expressed in the Bulk-Synchronous Parallel (BSP) and Token Dataflow compute models generate highly-structured communication workloads from messages propagating along graph edges. We can statially expose this structure to traffic compilers and optimization tools to reshape and reduce traffic for higher performance (or lower area, lower energy, lower cost). Such offline traffic optimization eliminates the need for complex, runtime NoC hardware and enables lightweight, scalable NoCs. We perform load balancing, placement, fanout routing, and fine-grained synchronization to optimize our workloads for large networks up to 2025 parallel elements for BSP model and 25 parallel elements for Token Dataflow. This allows us to demonstrate speedups between 1.2× and 22× (3.5× mean), area reductions (number of Processing Elements) between 3× and 15× (9× mean) and dynamic energy savings between 2× and 3.5× (2.7× mean) over a range of real-world graph applications in the BSP compute model. We deliver speedups of 0.5–13× (geomean 3.6×) for Sparse Direct Matrix Solve (Token Dataflow compute model) applied to a range of sparse matrices when using a high-quality placement algorithm. We expect such traffic optimization tools and techniques to become an essential part of the NoC application-mapping flow.

1. Introduction

Real-world communication workloads exhibit structure in the form of locality, sparsity, fanout distribution, and other properties. If this structure can be exposed to automation tools, we can reshape and optimize the workload to improve performance, lower area, and reduce energy. In this paper, we develop a traffic compiler that exploits structural properties of Bulk-Synchronous Parallel communication workloads. This compiler provides insight into performance tuning of communication-intensive parallel applications. The performance and energy improvements made possible by the compiler allow us to build the NoC from simple hardware elements that consume less area and eliminate the need for using complex, area-hungry, adaptive hardware. We now introduce key structural properties exploited by our traffic compiler.

(i) When the natural communicating components of the traffic do not match the granularity of the NoC architecture, applications may end up being poorly load balanced. We discuss *Decomposition* and *Clustering* as techniques to improve load balance.

(ii) Most applications exhibit sparsity and locality; an object often interacts regularly with only a few other objects in its neighborhood. We exploit these properties by *Placing* communicating objects close to each other.

(iii) Data updates from an object should often be seen by multiple neighbors, meaning the network must route the same message to multiple destinations. We consider *Fanout Routing* to avoid redundantly routing data.

(iv) Applications that use barrier synchronization can minimize node idle time induced by global synchronization between the parallel regions of the program by using *Fine-Grained Synchronization*.

We show the compilation flow for the NoC in Figure 1 and illustrate the relative benefits using an example application graph. We also show representative speedups for the ConceptNet `cnet-default` workload after each step of the traffic compilation flow when compared to an unoptimized mapping. The unoptimized graph has imbalanced, performance-limiting communication characteristics which get reshaped, reduced, and redistributed for an efficient execution by the traffic compiler. While these optimizations have been discussed independently in the literature extensively (e.g., [1, 2, 17, 19, 21]), we develop a toolflow that autotunes the control parameters of these optimizations per workload for maximum benefit and provide a quantification of the cumulative benefit of applying these optimizations to various applications in fine-grained, on-chip network settings. This quantification further illustrates how the performance impact of each optimization changes with NoC size. The key contributions of this paper include:

(i) development of a traffic compiler for fine-grained applications described using the BSP, Token Dataflow, and Static SIMD compute models,

(ii) use of communication workloads extracted from ConceptNet (BSP), Sparse Matrix-Vector Multiply (BSP), Bellman-Ford (BSP), and Sparse Direct Matrix Solve (Token Dataflow) running on range of real-world circuits and graphs,

(iii) quantification of cumulative benefits of each stage of the compilation flow (performance, area, energy) compared to the unoptimized case.

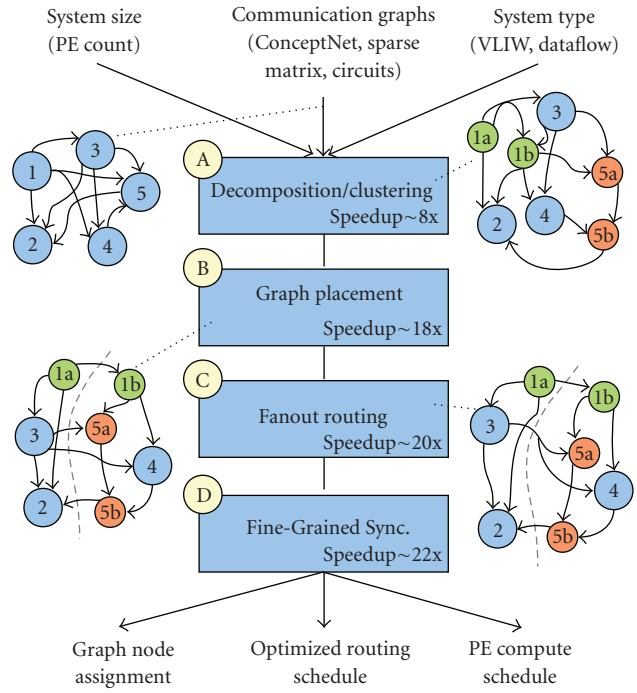


Figure 1: NoC traffic compilation flow (annotated with `cnet-default` speedups at 2025 PEs).

2. Background

2.1. Applications and Compute-Models. We consider two compute models and associated applications: (1) Graphstep [3], and (2) Token Dataflow [4].

2.1.1. Graphstep. Parallel graph algorithms are well suited for concurrent processing on FPGAs. We describe graph algorithms in a Bulk-Synchronous Parallel (BSP) compute model [5] and develop an FPGA system architecture [3] for accelerating such algorithms. The compute model defines the intended semantics of the algorithm, so we know which optimizations preserve the desired meaning while reducing NoC traffic. The graph algorithms are a sequence of steps where each step is separated by a global barrier. In each step, we perform parallel, concurrent operations on nodes of a graph data-structure where all nodes send messages to their neighbors while also receiving messages. The graphs in these algorithms are known when the algorithm starts and do not change during the algorithm. Our communication workload consists of routing a set of messages between graph nodes. We route the same set of messages, corresponding to the graph edges, in each epoch.

2.1.2. Token Dataflow. Lightweight processing of sparse dataflow graphs can be efficiently accelerated using FPGAs. In [4], we show how to accelerate Sparse Matrix-Solve computation generated by the KLU [6] solver using this approach. The dataflow compute model enables parallel distributed operations in the sparse dataflow graph by implementing the *dataflow firing rule*. According to this rule, each node in the graph will *fire* when it receives all its inputs. Instead of a global barrier, each node implements its own local synchronization operation. Each token is a message that is routed along an edge of the dataflow graph. We evaluate the complete graph by propagating tokens from the graph inputs all the way to the outputs in dataflow fashion. Our approach is similar to the implementation in [7]. Token propagating along parallel paths allows this token-passing architecture to exploit parallelism in the graph. Our workload is the ordered set of messages that are activated and routed according to the dependencies in the graph.

Applications in the compute models we consider generate traffic with a variety of communication characteristics (e.g., locality, sparsity, multicast) which also occur in other applications and compute models as well. Our traffic compiler exploits the *a priori* knowledge of structure-rich communication workloads (see Section 5.1) to provide performance benefits. Our approach differs from some recent NoC studies that use statistical traffic models (e.g., [8–11]) and random workloads (e.g., [12–14]) for analysis and experiments. Statistical and random workloads may exaggerate traffic requirements and ignore application structure leading to overprovisioned NoC resources and missed opportunities for workload optimization. We use real workloads generated from three different compute models to demonstrate the value and generality of our parallel approach.

2.1.3. Other Studies. In [8], the authors demonstrate a 60% area reduction along with an 18% performance improvement for well-behaved workloads. In [10], the authors show a 20% reduction in buffer sizes and a 20% frequency reduction for an MPEG-2 workload. In [12], the authors deliver a 23% reduction in time and a 23% reduction in area as well as a 38% reduction in energy for their design. We demonstrate better performance (95% reduction in runtime), lower area requirements (90% area savings), and lower energy consumption (90% less energy). Our approach is designed to deliver the larger, order-of-magnitude improvement because our systems (1) route fine-grained message-passing workloads, (2) utilize a high-throughput design of the Processing Elements, and (3) support scalability to larger system sizes. In contrast, the other studies attempt to optimize NoCs running restrictive, coarse-grained application workloads which reduce the impact of traffic characteristics on overall system behavior.

2.2. Architecture. We organize our FPGA NoC as a bidirectional 2D mesh [15] with a packet-switched routing network as shown in Figure 2(a). The application graph is distributed across the Processing Elements (PEs) which are specialized to process graph nodes for the different compute models. Portions of the application graph are stored in local on-chip memories in each PE. The PE is internally pipelined and capable of injecting and receiving a new packet in each cycle. The PE can simultaneously handle incoming and outgoing messages.

2.2.1. BSP PE. For the BSP PE shown in Figure 2(b), each PE performs accumulate and update computations on each node as defined by the BSP graph algorithm. The internal PE pipelines are managed by the GraphStep Logic controller. During execution, the controller iterates through all local nodes and generates outbound traffic that is routed over the packet-switched network. Inbound traffic is stored in the incoming message buffers of each PE. Once all messages have been received, a barrier is detected using a global reduce tree (a bit-level AND-reduce tree). The graph application proceeds through multiple global barriers until the algorithm terminates.

2.2.2. Token Dataflow PE. For the Token Dataflow PE shown in Figure 2(c), each PE implements the *dataflow firing rule* by keeping track of the number of messages received on each graph node. In each execution of the graph, the PE starts processing the graph from the inputs nodes and successively propagates the computation through the levels of the graph to the outputs. We do not use global barriers and instead allow distributed local barriers at each node. After a node is fired, it is inserted into the result queue that injects message into the network for each recipient of the node. This propagates the computation through the levels of the graph. We declare termination when all network traffic and PE activity has quiesced using a global reduce tree.

2.2.3. Network Switch. Each switch in the bidirectional 2D mesh supports fully-pipelined operation using composable *Split* and *Merge* units as shown in Figure 2(d). The switches in the bidirectional mesh network implement the Dimension-Ordered Routing (DOR) algorithm [16] that is simplest to realize in hardware and widely used in NoC designs. We discuss additional implementation parameters in Section 4.2. Prior to execution, the traffic compiler is responsible for allocating graph nodes to PEs.

We measure network performance as the number of clock cycles (*Barrier_Cycles*) required for one epoch between barriers, including both computation and all messages routing. We report speedups as Speedup = $Barrier\_Cycles_{unoptimized}/Barrier\_Cycles_{optimized}$.

3. Optimizations

In this section, we describe a set of optimizations performed by our traffic compiler. The compiler accepts the graph structure from the application and maps it to the NoC architecture. It suitably modifies the graph structure (replacing nodes and edges) and generates an assignment of graph nodes to the PEs of the NoC. The traffic compiler also selects the type of synchronization implemented in the PEs. It is a fully automated flow that sequences the different graph optimizations to generate an optimized mapping.

3.1. Decomposition. Ideally for a given application, as the PE count increases, each PE holds smaller and smaller portions of the workload. For graph-oriented workloads, unusually large nodes with a large number of edges (i.e., nodes that send and receive many messages) can prevent the smooth distribution of the workload across the PEs. As a result, performance is limited by the time spent sending and receiving messages at the largest node (streamlined message processing in the PEs implies work $\propto$ number of messages per node). For example, the ConceptNet `cnet-default` workload has a communicating node with 16 K input edges and 36 K outgoing edges (about 10% of total edges, see Table 1). For a fully-streamlined, high-throughput PE operation (1 cycle/network operation), we will need to spend at least 16 K cycles receiving messages and 36 K cycles sending messages irrespective of the number of PEs in the parallel NoC. *Decomposition* is a strategy where we break down large nodes into smaller nodes (either inputs, outputs or both can be decomposed) and distribute the work of sending and receiving messages at the large node over multiple PEs. The idea is similar to that used in synthesis and technology mapping of logic circuits [1]. Figure 3 illustrates the effect of decomposing a node. Node 5 with 3 inputs gets *fanin-decomposed* into Node 5a and 5b with 2 inputs, each thereby reducing the serialization at the node from 3 cycles to 2. Similarly, Node 1 with 4 outputs is *fanout-decomposed* into Node 1a and 1b with 3 outputs and 2 outputs each. Greater benefits can be achieved with higher-fanin/fanout nodes as is the case with `cnet-default` workload (see Table 1).

In general, when the output from the graph node is a result which must be multicast to multiple outputs, we can easily build an output fanout tree to decompose output routing. However, input edges to a graph node can only

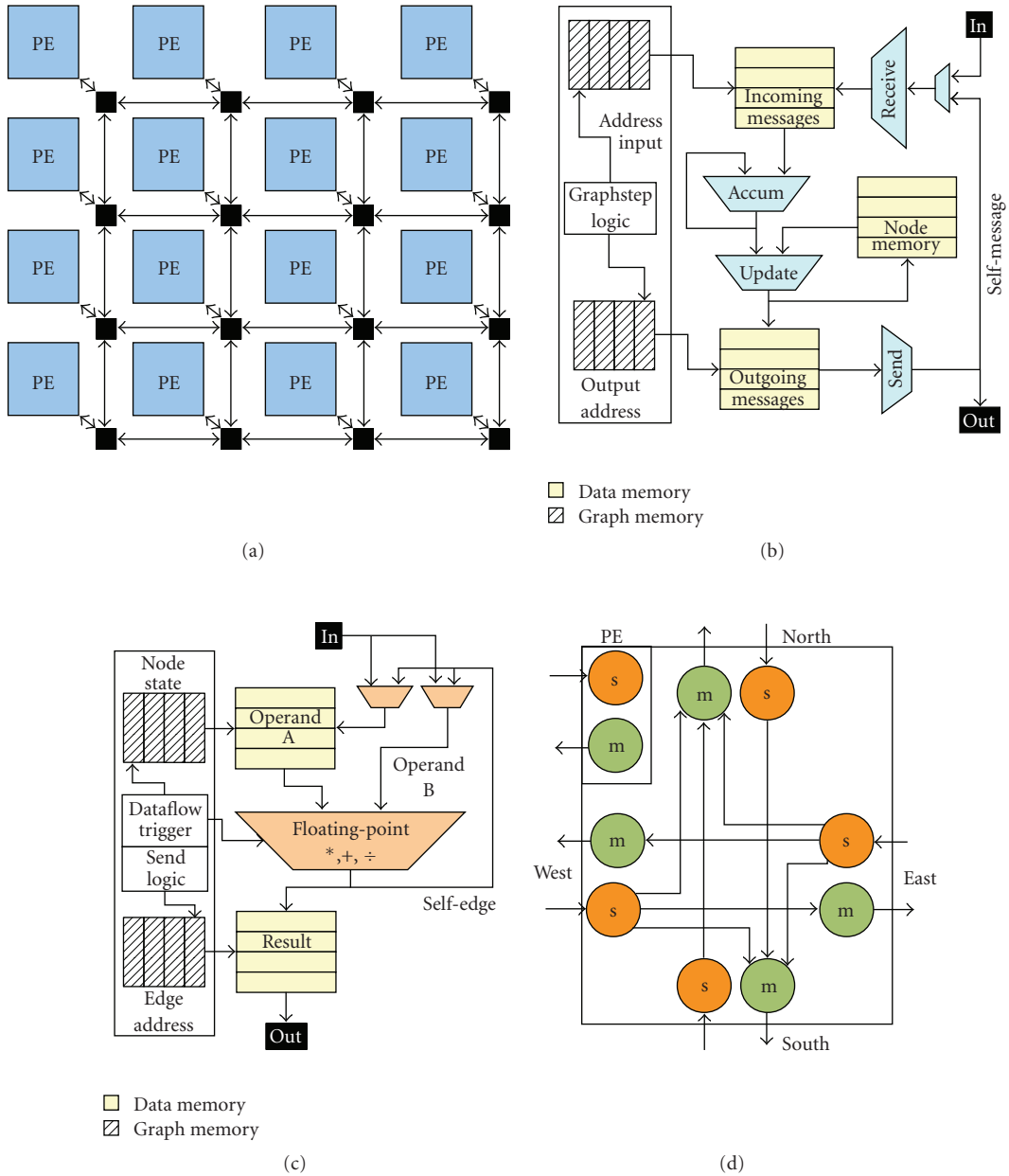


Figure 2: NoC architecture and organization.

be decomposed when the operation combining inputs is associative. ConceptNet and Bellman-Ford (discussed later in Section 4.1) permit input decomposition since nodes perform simple integer sum and max operations which are associative and can be decomposed. However, Matrix Multiply nodes perform nonassociative *floating-point accumulation* over incoming values which cannot be broken up and distributed.

We implement the decomposition phase by constructing an n-ary tree that replaces the high-fanin or high-fanout node. As as example, consider node 6 in Figure 4 with 5 inputs. We construct a suitable 2-ary tree rooted at node 6 and assign the fanin nodes to the leaves of this tree. When the number of inputs are not a power-of-2, we generate an appropriate imbalanced tree to accommodate all inputs. Presently, our algorithm orders the inputs or outputs to the leaves arbitrarily while showing performance and scalability improvements. In the future, we can also consider sophisticated tree construction algorithms that attempt to carefully minimize the critical path or post-placement locality during leaf assignment.

3.2. Clustering. While *Decomposition* is necessary to break up large nodes, we may still have an imbalanced system if we randomly place nodes on PEs. Random placement fails to account for the varying amount of work performed per node. Lightweight *Clustering* is a common technique used to quickly distribute nodes over PEs to achieve better load

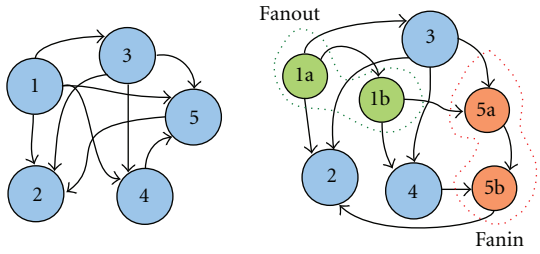


Figure 3: *Decomposition.*

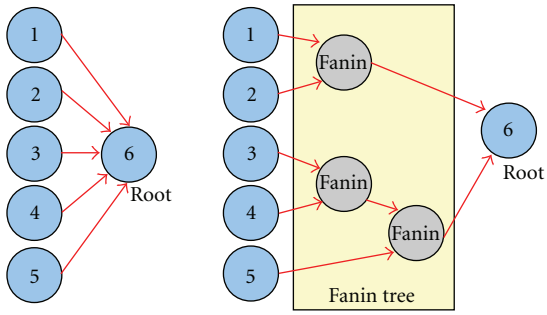


Figure 4: Fanin *Decomposition* tree example.

$$\text{Inputs} = \sum_{\text{node}\in\text{cluster}} (\text{Input\_Edges})\,(\text{node}) \quad (1)$$

$$\text{Outputs} = \sum_{\text{node}\in\text{cluster}} (\text{Output\_Edges})\,(\text{node}) \quad (2)$$

$$\text{MaxIn} = \max_{\text{node}\in\text{cluster}} (\text{Input\_Edges})\,(\text{node}) \quad (3)$$

$$\text{NetworkIO} = \max(\text{Inputs}, \text{Outputs}) \quad (4)$$

$$\text{Compute} = \max(\text{Inputs}, \text{MaxIn} * \text{Cycles}) \quad (5)$$

$$\text{Cost} = \text{NetworkIO} + \text{Compute} \quad (6)$$

Figure 5: Clustering cost function.

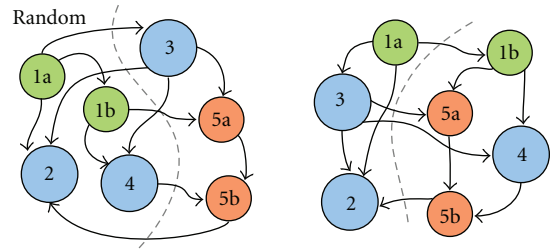


Figure 6: *Placement* (random placement versus good placement).

balance (e.g., [17]). We use a greedy, linear-time *Clustering* algorithm similar to the *Cluster Growth* technique from [17]. We start by creating as many "clusters" as PEs and randomly assign a seed node to each cluster. We then pick nodes from the graph and greedily assign them to the PE that least increases cost. The cost function ("Closeness metric" in [17]) is chosen to capture the amount of work done in each PE including sending messages, receiving messages, or computing on a node. This is expressed as shown in Figure 5. In the equation, we compute total cost as a sum of communication and computation costs. The clustering algorithm then picks the cluster which has the smallest Cost. Communication cost is simply the larger of the send and receive cycles. Compute cost is a function of the number of cycles in the pipelined compute datapath (for nonassociative computation, we must add the total pipeline cycles per edge, *Cycles*). We can explore other greedy clustering packages (e.g., T-Vpack [18]) as part of future enhancements to the compiler.

3.3. Placement. Object communication typically exhibits locality. A random placement ignores this locality resulting in more traffic on the network. Consequently, random placement imposes a greater traffic requirement which can lead to poor performance, higher energy consumption, and inefficient use of network resources. We can *Place* communicating nodes close to each other to minimize traffic requirements and get better performance than random placement. The benefit of performing placement for NoCs has been discussed in [19]. Good placement reduces both the number of messages that must be routed on the network and the distance which each message must travel. This decreases competition for network bandwidth and lowers the average latency required by the messages. Figure 6 shows a simple example of good *Placement*. A random partitioning of the application graph may bisect the graph with a cut size of 6 edges (i.e., 6 messages must cross the chip bisection). Instead, a high-quality partitioning of the graph will find a lower cut width size of 4. The load on the network will be reduced since 2 fewer messages must cross the bisection. In general, *Placement* is an NP-complete problem, and finding an optimal solution is computationally intensive. We use a fast multilevel partitioning heuristic MLPART [20] that iteratively clusters nodes and moves the clustered nodes around partitions to search for a better quality solution. In future, we can improve placement quality with a slower simulated-annealing heuristic.

3.4. Fanout Routing. Some applications may require multicast messages (i.e., single source, multiple destinations). Our application graphs contain nodes that send the exact same message to their destinations. Routing redundant messages is a waste of network resources. We can use the network more efficiently with *Fanout Routing* which avoids routing redundant messages. This has been studied extensively by Duato et al. [21]. If many destination nodes reside in the same physical PE, it is possible to send only one message instead of many, duplicate messages to the PE. For this to be beneficial, there needs to be at least two sink nodes in some destination PE. The PE will then internally distribute the message to the intended recipients. This is shown in Figure 7. The fanout edge from Node 3 to Node 5a and Node 4 can be replaced with a shared edge as shown. This reduces the number of messages crossing the bisection by 1. This

optimization works best at reducing traffic and message-injection costs at low PE counts. As PE counts increase, we have more possible destinations for the outputs and fewer shareable nodes in the PEs resulting in decreasing benefits. We realize this optimization with no hardware overheads. We simply configure the data-structures with appropriate addresses for indexing into the message memory.

3.5. Fine-Grained Synchronization. In parallel programs with multiple threads, synchronization between the threads is sometimes implemented with a global barrier for simplicity. However, the global barrier may artificially serialize computation. Alternately, the global barrier can be replaced with local synchronization conditions that avoid unnecessary sequentialization. Techniques for eliminating such barriers have been previously studied [2, 22]. In the BSP compute model discussed in Section 2, execution is organized as a series of parallel operations separated by barriers. We use one barrier to signify the end of the communicate phase and another to signify the end of the compute phase. If it is known prior to execution that the entire graph will be processed, the first barrier can be eliminated by using local synchronization operations. A node can be permitted to start the compute phase as soon as it receives all its incoming messages without waiting for the rest of the nodes to have received their messages. This prevents performance from being limited by the sum of worst-case compute and communicate latencies when they are not necessarily coupled. This is implemented by adding extra logic and state to store and update messages received per node as well as the total message count per node. We show the potential benefit of *Fine-Grained Synchronization* in Figure 8. Node 2 and Node 3 can start their *Compute* phases after they have received all their inputs messages. They do not need to wait for all other nodes to receive all their messages. This optimization enables the *Communicate* phase and the *Compute* phase to be overlapped.

4. Experimental Setup

4.1. Workloads. We generate workloads from a range of applications mapped to the BSP compute model and the Token Dataflow model. We choose applications that cover different domains including AI, Scientific Computing, and CAD optimization that exhibit important structural properties.

4.1.1. ConceptNet (BSP). ConceptNet [23] is a commonsense reasoning knowledge base described as a graph, where nodes represent concepts and edges represent semantic relationships. Queries to this knowledge base start a *spreading-activation* algorithm from an initial set of nodes. The computation spreads over larger portions of the graph through a sequence of steps by passing messages from activated nodes to their neighbors. In the case of complex queries or multiple simultaneous queries, the entire graph may become activated after a small number of steps. We route all the edges in the graph representing this worst-case step. In [3], we show a

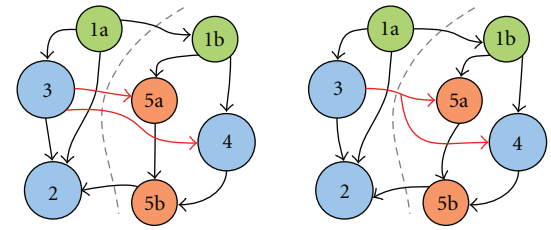


Figure 7: *Fanout-Routing.*

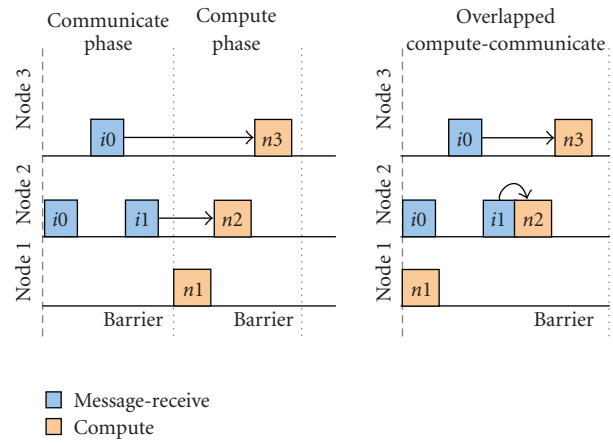


Figure 8: *Fine-Grained Synchronization.*

per-FPGA speedup of 20× when comparing a Xilinx Virtex-2V6000 FPGA to an optimized sequential implementation on a 3.4 GHz Pentium-4Xeon. We use the BSP compute model to capture the semantics of the computation and implement the application on a Graphstep PE (Figure 2(b)) with an associative datapath (input edges can be summarized associatively as they arrive without being sequentialized with a particular ordering).

4.1.2. Sparse Matrix-Vector Multiply (BSP). Iterative Sparse Matrix-Vector Multiply (SMVM) is the dominant computational kernel in several numerical routines (e.g., Conjugate Gradient, GMRES). In each iteration, a set of dot products between the vector and matrix rows is performed to calculate new values for the vector to be used in the next iteration. We can represent this computation as a graph where nodes represent matrix rows and edges represent the communication of the new vector values. The graph captures the sparse communication structure inherent in the dot-product expression. In each iteration, messages must be sent along all edges; these edges are multicast as each vector entry must be sent to each row graph node with a nonzero coefficient associated with the vector position. We use sample matrices from the Matrix Market benchmark [24]. Our traffic optimized implementation in this paper improves over the ≈2× speedups demonstrated previously in [25] when comparing a Xilinx Virtex-2V6000 with an Itanium 2 processor. We use the BSP compute model to express the sparse matrix computation and realize the processing on a Graphstep PE (Figure 2(b)) with a nonassociative datapath.

4.1.3. Bellman-Ford (BSP). The Bellman-Ford algorithm solves the single-source shortest-path problem, identifying any negative edge weight cycles, if they exist. It finds application in CAD optimizations like Retiming, Static Timing Analysis, and FPGA Routing where the graph structure is a representation of the physical circuit. Nodes represent gates in the circuit while edges represent wires between the gates. The algorithm simply relaxes all edges in each step until quiescence. A relaxation consists of computing the minimum at each node over all weighted incoming message values. Each node then communicates the result of the minimum to all its neighbors to prepare for the next relaxation. Again, we capture this computation in the BSP compute model and implement it on a Graphstep PE (Figure 2(b)) with an associative datapath.

4.1.4. Sparse Direct Matrix Solve in SPICE (Token Dataflow). Matrix Solve computation on sparse matrices is a key repetitive component of many applications like the SPICE circuit simulator. For SPICE, we prefer to use sparse direct solver techniques than SMVM-based (see Section 4.1.2) iterative techniques for reasons of robustness at the expense of parallelism. We integrate the KLU solver with SPICE to expose parallelism in the computation with a one-time symbolic analysis at the start of the simulation. We extract sparse dataflow graphs for the LU factorization and Front-Solve and Back-Solve phases which we distribute across our parallel NoC architecture. Each node in the graph represents a floating-point operation (add, multiply, or divide) while each edge represents a dependency in the calculation. The original nonzero values in the sparse matrix represent graph inputs, and the factored L and U nonzero values represent the graph outputs. We evaluate the graph by propagating tokens from the inputs of the graph to its outputs. This computation is processed on the NoC architecture using a Dataflow PE (Figure 2(c)). In [26], we show how to construct the dataflow graph and accelerate the Sparse Matrix Solve computation by 0.6–7.1× when comparing a Xilinx Virtex-6 LX760 with an optimized sequential implementation on an Intel Core i7 965.

4.2. NoC Timing and Power Model. All our experiments use a single-lane, bidirectional-mesh topology that implements a Dimension-Ordered Routing function. The network for Matrix-Vector Multiply and Sparse Direct Matrix Solve experiments is 84-bit wide (64-bit double-precision data, 20-bit header/address) while the network for ConceptNet and Bellman-Ford experiments is 52-bit wide (32-bit integer data, 20-bit header/address). The switch is internally pipelined to accept a new packet on each cycle (see Figure 2(a)). Different routing paths take different latencies inside the switch (see Table 2). We pipeline the wires between the switches for high performance (counted in terms of cycles required as T_{wire}). The PEs are also pipelined to start processing a new edge every cycle. ConceptNet and Bellman-Ford compute simple sum and max operations while Matrix-Vector Multiply performs floating-point *accumulation* on the incoming messages. The Sparse Direct Matrix Solve algorithm performs floating-point *accumulation* and divide operations on the incoming messages as appropriate. For our floating-point benchmarks, we consider double-precision implementations of all operations. Each computation on the edge then takes 1–57 cycles of latency to complete (see Table 2). We estimate dynamic power consumption in the switches using XPower [27]. Dynamic power consumption

Table 1: Application graphs.

Graph	Nodes	Edges	Max Fanin	Max Fanout
BSP Compute Model [3]				
ConceptNet				
`cnet-small`	14556	27275	226	2538
`cnet-default`	224876	553837	16176	36562
Matrix-Multiply				
`add20`	2395	17319	124	124
`bcsstk11`	1473	17857	27	30
`fidap035`	19716	218308	18	18
`fidapm37`	9152	765944	255	255
`gemat11`	4929	33185	27	28
`memplus`	17758	126150	574	574
`rdb32001`	3200	18880	6	6
`utm5940`	5940	83842	30	20
Bellman-Ford				
`ibm01`	12752	36455	33	93
`ibm05`	29347	97862	9	109
`ibm10`	69429	222371	137	170
`ibm15`	161570	529215	267	196
`ibm16`	183484	588775	163	257
`ibm18`	210613	617777	85	209
Token Dataflow Compute Model [4]				
`bcspwr01`	753	985	3	6
`mux8`	1037	1395	3	8
`ringosc`	2883	3866	3	4
`psadmit1`	9814	13356	3	10
`dac`	43000	67265	3	10
`psadmit2`	22259	30108	3	11
`sandia01`	40400	55765	3	8
`sandia02`	40400	55765	3	8
`s208`	43055	62067	3	11
`bcspwr09`	221807	391654	3	53
`s298`	70928	106247	3	13
`s344`	70666	103314	3	12
`s349`	73914	108888	3	14
`s382`	81060	119475	3	16
`s444`	90288	133901	3	16
`s386`	100637	151868	3	20
`s510`	220092	380930	3	54
`s526`	146442	228017	3	26
`s641`	212474	348453	3	39
`10stages`	124720	178396	3	8
`circuit2`	416454	747587	3	172

Table 2: NoC timing model.

Mesh Switch	Latency
T_{through} (X-X, Y-Y)	2
T_{turn} (X-Y, X-Y)	4
$T_{\text{interface}}$ (PE-NoC, NoC-PE)	6
T_{wire} (GraphStep NoC)	2
T_{wire} (Token Dataflow NoC)	5
Processing Element	**Latency**
T_{send}	1
T_{receive} (ConceptNet, Bellman-Ford)	1
T_{receive} (Matrix-Vector Multiply)	9
T_{add} (Sparse Matrix Solve)	8
T_{multiply} (Sparse Matrix Solve)	10
T_{divide} (Sparse Matrix Solve)	57

at different switching activity factors is shown in Table 3. We extract switching activity factor in each Split and Merge unit from our packet-switched simulator. When comparing dynamic energy, we multiply dynamic power with simulated cycles to get energy. We generate bitstreams for the switch and PE on a Xilinx Virtex-5 LX110T FPGA [27] to derive our timing and power models shown in Tables 2 and 3.

4.3. Packet-Switched Simulator. We use a Java-based cycle-accurate simulator that implements the timing model described in Section 4.2 for our evaluation. The simulator models both computation and communication delays, simultaneously routing messages on the NoC and performing computation in the PEs. Our results in Section 5 report performance observed on cycle-accurate simulations of different circuits and graphs. The application graph is first transformed by a, possibly empty, set of optimizations from Section 3 before being presented to the simulator.

5. Evaluation

We now examine the impact of the different optimizations on various workloads to quantify the cumulative benefit of our traffic compiler. Our performance baseline is an unoptimized, unprocessed, barrier-synchronized graph workload which is randomly distributed across the NoC PEs. We order the optimization appropriately to analyze their additive impacts. We first show relative scaling trends for the total routing time for the `bcsstk11` benchmark to identify potential performance bottlenecks. We then quantify the impact of each optimization in systematically eliminating these bottlenecks. Initially, we load balance our workloads by performing *Decomposition*. We then determine how the workload gets distributed across PEs using *Clustering* or *Placement*. Finally, we perform *Fanout Routing* and *Fine-Grained Synchronization* optimizations. We illustrate scaling trends of individual optimizations using a single illustrative workload for greater clarity. At the end, we show cumulative data for all benchmarks together.

5.1. Performance Scaling Trends. Ideally, as PE counts increase, the application performance scales accordingly, ($T_{\text{parallel}} = T_{\text{sequential}}/\text{PEs}$) where $T_{\text{sequential}}$ = sequential time. However, applications rarely exhibit such ideal behavior. We recognize three potential bottlenecks that can prevent ideal scaling which we illustrate in Figure 9 for the `bcsstk11` benchmark.

5.1.1. Serialization. This metric measures the number of cycles spent injecting or receiving messages at the NoC-PE interface. We measure this as follows:

$$\begin{aligned} T_{\text{input}} &= \left(N_{\text{input}} + N_{\text{self}}\right) \times T_{\text{send}}, \\ T_{\text{output}} &= \left(N_{\text{output}} + N_{\text{self}}\right) \times T_{\text{receive}}. \end{aligned} \tag{1}$$

In Equation (1) and Equation (8), N_{input} = number of messages entering the PE, N_{output} = number of messages leaving the PE, N_{self} = number of self-messages in the PE, and T_{send} and T_{receive} = number of cycles between successive sends and receives, respectively. We engineer our PEs to handle an external input, output, or self message in one cycle ($T_{\text{send}} = T_{\text{receive}} = 1$). The internal memory architecture of our PE requires N_{self} to be counted on both ports. Since our input and output interfaces work independently and simultaneously, we define the total serialization cost as follows:

$$T_{\text{serialization}} = \max\left(T_{\text{input}}, T_{\text{output}}\right). \tag{2}$$

We expect that, for ideal scaling, the number of serialization cycles decrease with increasing PE counts. We distribute both the computation and the communication over more PEs. However, communication from very large graph nodes (i.e., large number of edges) will cause a serial bottleneck at the PE-NoC interface. In Section 3, we discussed *Decomposition* (Section 3.1), *Clustering* (Section 3.2), and *Fanout Routing* (Section 3.3) as strategies to avoid this bottleneck.

5.1.2. Bisection. This metric measures the number of cycles required for messages to cross the chip bisection. If the volume of NoC traffic crossing the chip bisection is larger than the number of physical wires (NoC channels in the bisection × Channel Width), then the bisection must be reused

$$T_{\text{cut}} = \left\lceil \frac{N_{\text{messages}}}{N_{\text{wires}}} \right\rceil. \tag{3}$$

The top-level bisection may not be the largest bottleneck in the network. Hence, we consider several hierarchical cuts (horizontal and vertical cuts for a mesh topology) and identify the most limiting of cuts ($T_{\text{cut}i}$)

$$T_{\text{bisection}} = \max_{\text{cuts}i}(T_{\text{cut}i}). \tag{4}$$

For applications with high locality, the amount of traffic crossing the bisection is low (when placed properly) and bandwidth does not become a bottleneck. Conversely, for

Table 3: NoC dynamic power model.

Datawidth (Application)	Block	dynamic power at diff. activity (mW)				
		0%	25%	50%	75%	100%
52 (ConceptNet, Bellman-Ford)	Split	0.26	1.07	1.45	1.65	1.84
	Merge	0.72	1.58	2.1	2.49	2.82
84 (Matrix-Vector Multiply, Sparse Marix Solve)	Split	0.32	1.35	1.78	2.02	2.26
	Merge	0.9	1.87	2.45	2.88	3.25

application with low locality, a larger number of messages need to cross the bisection and bisection bandwidth can become a bottleneck. In the Section 3, we considered *Placement* (Section 3.3) and to a lesser extent *Clustering* (Section 3.2) to reduce bisection bandwidth requirements.

5.1.3. Latency. This metric measures the sum of switch latencies and wire latencies along the worst-case message path in the NoC assuming no congestion

$$T_{\text{path}} = \sum_{\text{switches}\,j} T_{\text{switch}_j} + \sum_{\text{wires}\,k} T_{\text{wire}_k},$$
$$T_{\text{latency}} = \max_{\text{messages}\,i} \left(T_{\text{path}_i}\right). \tag{5}$$

For barrier-synchronized workloads, all data is routed from sources to sinks in an epoch. At small PE counts, the number of cycles required to cross the network will be small compared to serialization or bisection cycles. However, as the PE counts increase, the latency in the network will also increase and eventually dominate both serialization and bisection. In the high latency regime, latency hiding techniques like *Fine-Grained Synchronization* (Section 3.4) that overlap compute and communicate phases become necessary.

In Figure 9, we observe that at low PE counts (PE < 25), most cycles are dedicated towards serialized processing at the NoC-PE interfaces. As we increase the number of PEs (25 < PE < 128), the number of messages in the network increases causing the network to become bandwidth bottlenecked. Eventually, at high PE counts (PE > 128), performance is dominated by latency.

5.2. Impact of Individual Optimizations.

5.2.1. Decomposition. In Figure 10, we show how the Concept-Net `cnet-default` workload scales with increasing PE counts under *Decomposition*. We observe that *Decomposition* allows the application to continue to scale up to 2025 PEs and possibly beyond. Without *Decomposition*, performance quickly runs into a serialization bottleneck due to large nodes as early as 100 PEs. The decomposed NoC workload manages to outperform the undecomposed case by 6.8× in performance. However, the benefit is lower at low PE counts, since the maximum logical node size becomes small compared to the average work per PE. Additionally, decomposition is only useful for graphs with high degree

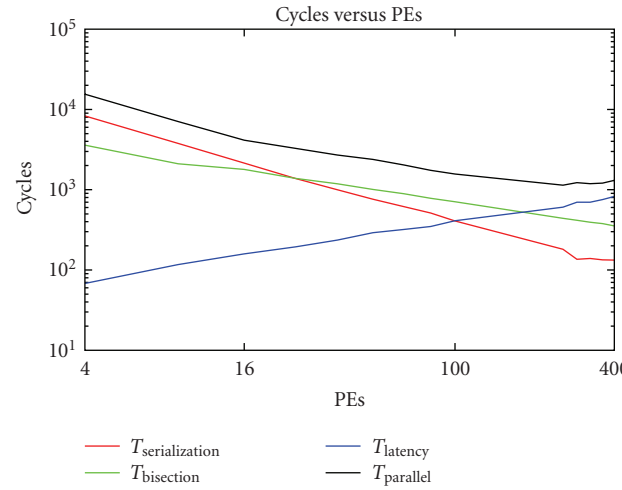


Figure 9: Explaining performance scaling of total cycles for `bcsstk11` benchmark.

(see Table 1). In Figure 11, we show how the *decomposition limit* control parameter impacts the scaling of the workload. As expected, without decomposition, performance of the workload saturates beyond 32 PEs. Decomposition with a limit of 16 or 32 allows the workload to scale up to 400 PEs and provides a speedup of 3.2× at these system sizes. However, if we attempt an aggressive decomposition with a limit of 2 (all decomposed nodes allowed to have a fanin and fanout of 2), performance is actually worse than undecomposed case between 16 and 100 PEs and barely better at larger system sizes. At such small decomposition limits, performance gets worse due to an excessive increase in the workload size (i.e., number of edges in the graph). Our traffic compiler sweeps the design space and automatically selects the best decomposition limit.

5.2.2. Clustering. In Figure 12, we show the effect of *Clustering* on performance with increasing PE counts. *Clustering* provides an improvement over *Decomposition* since it accounts for compute and message injection costs accurately, but that improvement is small (1–18%). Remember from Section 3 that *Clustering* is a lightweight, inexpensive optimization that attempts to improve load balance and as a result, we expect limited benefits.

5.2.3. Placement. In Figure 13, we observe that *Placement* provides as much as 2.5× performance improvement over

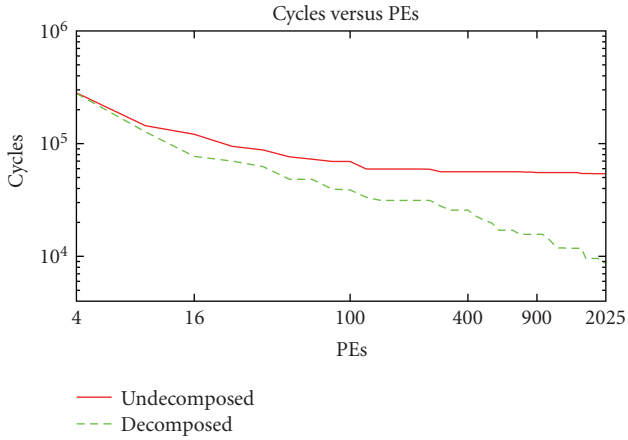


FIGURE 10: *Decomposition* (`cnet-default`).

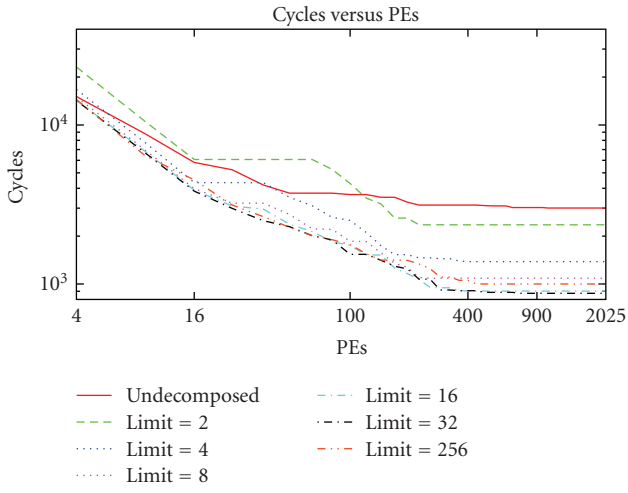


FIGURE 11: *Decomposition Limits* (`cnet-small`).

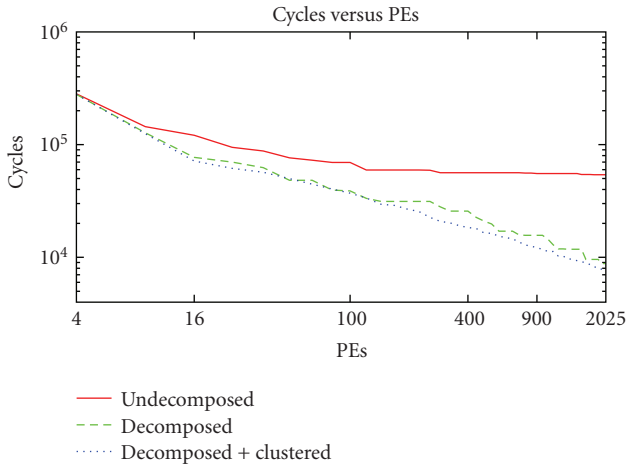


FIGURE 12: *Decomposition* and *Clustering* (`cnet-default`).

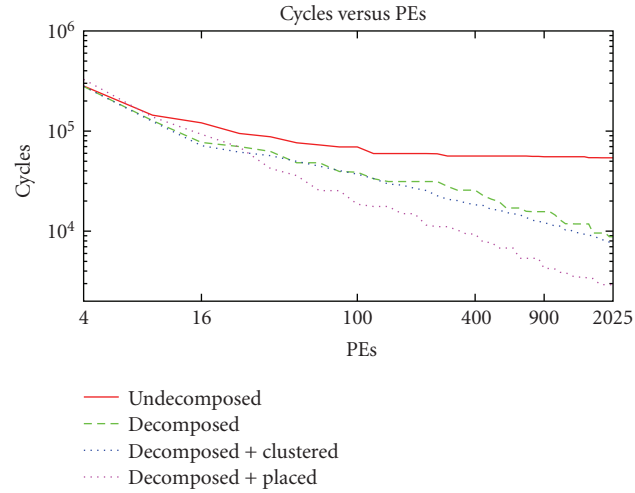


FIGURE 13: *Decomposition*, *Clustering*, and *Placement* (`cnet-default`).

a randomly placed workload as PE counts increase. At high PE counts, localized traffic reduces bisection bottlenecks and communication latencies. However, *Placement* is less effective at low PE counts since the NoC is primarily busy injecting and receiving traffic and NoC latencies are small and insignificant. Moreover, good load balancing is crucial for harnessing the benefits of a high-quality placement.

As we can see in Figure 14, we get speedups of 0.5–13× (geomean 3.6×) for Sparse Matrix Solve benchmarks when using a high-quality placement for distributing the graph operations when compared to a random placement. Random placement performs poorly for the Token Dataflow compute model as it is unable to minimize network bandwidth. For the `s208` benchmark, random placement does provide a 2× benefit over good placement. This is because our placement algorithm currently does not specifically attempt to constrain long critical dependency chains within the PE. However, for all other benchmarks, the high-quality placer is able to outperform random placement by optimizing bandwidth. In the future, with a different placement strategy that targets latency of the computation, we expect to be able to outperform random in all cases by an even greater amount.

5.2.4. Fanout-Routing. We show performance scaling with increasing PEs for the Bellman-Ford `ibm01` workload using *Fanout Routing* in Figure 15. The greatest performance benefit (1.5×) from *Fanout Routing* comes when redundant messages distributed over few PEs can be eliminated effectively. The absence of benefit at larger PE counts is due to negligible shareable edges as we suggested in Section 3.

5.2.5. Fine-Grained Synchronization. In Figure 16, we find that the benefit of *Fine-Grained Synchronization* is the greatest (1.6×) at large PE counts when latency dominates performance. At low PE counts, although NoC latency is small, elimination of the global barrier enables greater freedom in

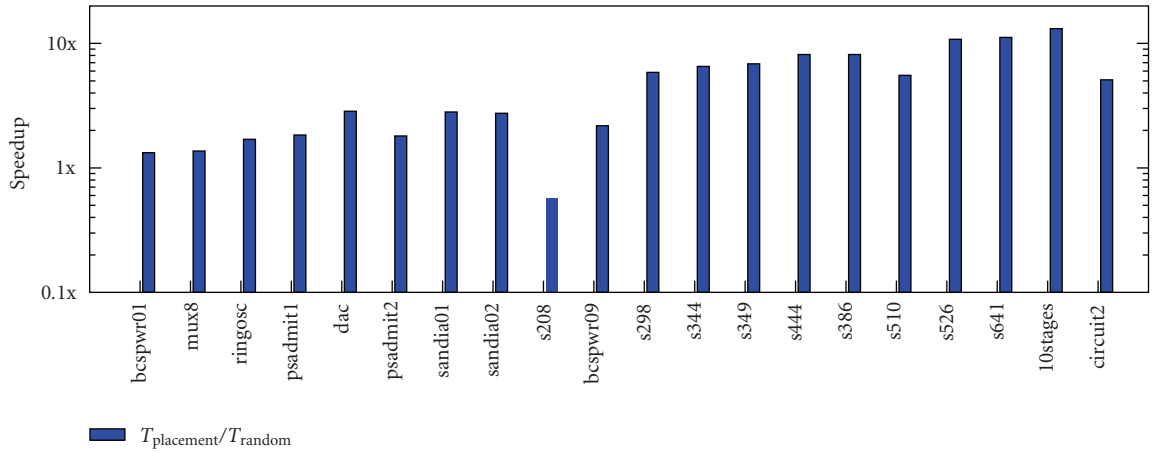


Figure 14: Impact of *Placement* at 25 PEs (Sparse Matrix Solve computation).

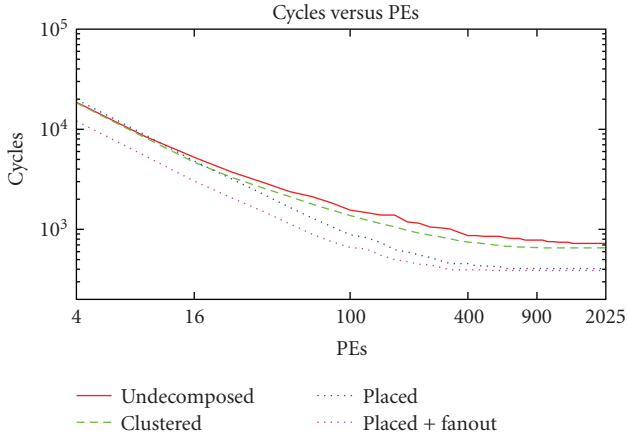


Figure 15: *Clustering*, *Placement,* and *Fanout-Routing* (`ibm01`).

scheduling PE operations and consequently we observe a nonnegligible improvement (1.2×) in performance. Workloads with a good balance between communication time and compute time will achieve a significant improvement from fine-grained synchronization due to greater opportunity for overlapped execution.

5.3. Cumulative Performance Impact. We look at cumulative speedup contributions and relative scaling trends of all optimizations for all workloads at 25 PEs, 256 PEs, and 2025 PEs. The relative impact and importance of these optimizations shift as a function of system size. In some cases, a particular optimization is irrelevant at a particular PE count point in the NoC design space; for example, fanout routing is most useful at small system sizes and placement is important at larger system sizes.

At 25 PEs, Figure 17, we observe modest speedups in the range 1.5× to 3.4× (2× mean) which are primarily due to *Fanout Routing*. *Placement* and *Clustering* are unable to contribute significantly since performance is dominated by computation. *Fine-Grained Synchronization* also provides some improvement, but as we will see, its relative contribution increases with PE count.

At 256 PEs, Figure 18, we observe larger speedups in the range 1.2× to 8.3× (3.5× mean) due to *Placement*. At these PE sizes, the performance bottleneck begins to shift to the network, so reducing traffic on the network has a larger impact on overall performance (see previous discussion in Section 5.1). We continue to see performance improvements from *Fanout Routing* and *Fine-Grained Synchronization*.

At 2025 PEs, Figure 19, we observe an increase in speedups in the range 1.2× to 22× (3.5× mean). While there is an improvement in performance from *Fine-Grained Synchronization* compared to smaller PE cases, the modest quantum of increase suggests that the contributions from other optimizations are saturating or reducing.

Overall, we find ConceptNet workloads show impressive speedups up to 22×. These workloads have decomposable nodes that allow better load balancing and have high locality. They are also the only workloads which have the most need for *Decomposition*. Bellman-Ford workloads also show good overall speedups as high as 8×. These workloads are circuit graphs and naturally have high locality and fanout. Matrix-Multiply workloads are mostly unaffected by these optimization and yield speedups not exceeding 4× at any PE count. This is because the compute phase dominates the communicate phase; compute requires high-latency (9 cycles/edge from Table 2) floating-point operations for each edge. It is also not possible to decompose inputs due to the nonassociativity of the floating-point accumulation. As an experiment, we decomposed both inputs and outputs of the `fidapm37` workload at 2025 PEs and observed an almost 2× improvement in performance.

5.4. Cumulative Area and Energy Impact. For some low-cost applications (e.g., embedded), it is important to minimize NoC implementation area and energy. The optimizations we discuss are equally relevant when cost is the dominant design criteria.

To compute the area savings, we first pick the smallest PE count (PE_{unopt} in Figure 20) that is within 1.1× the

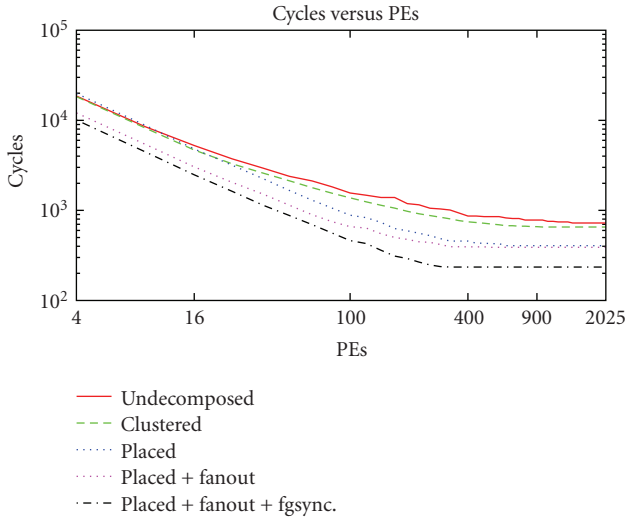


FIGURE 16: *Clustering*, *Placement*, *Fanout-Routing*, and *Fine-Grained Synchronization* (`ibm01`).

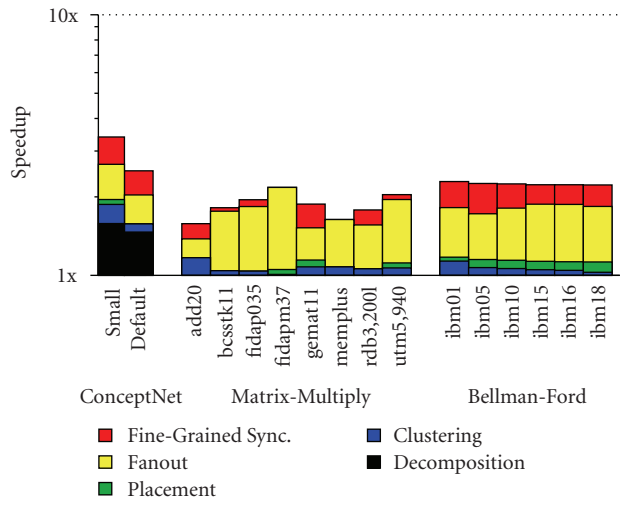


FIGURE 17: Performance ratio at 25 PEs.

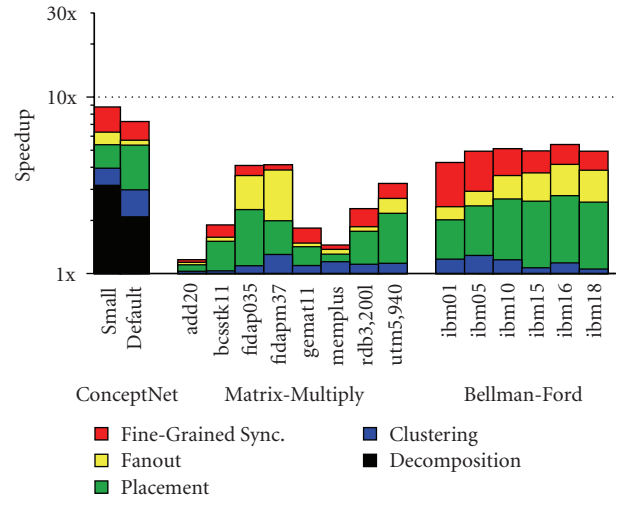


FIGURE 18: Performance ratio at 256 PEs.

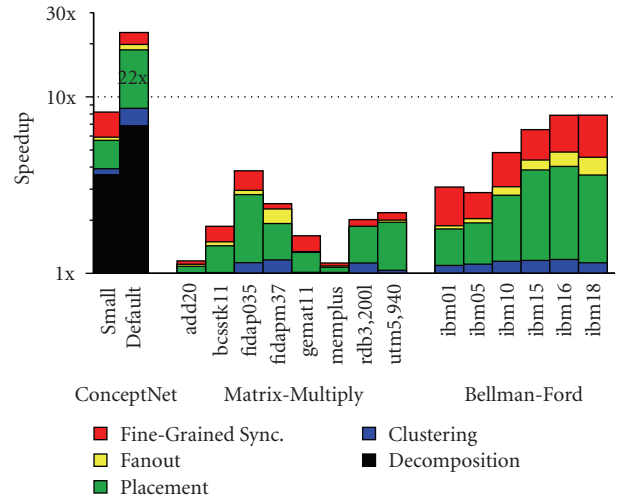


FIGURE 19: Performance ratio at 2025 PEs.

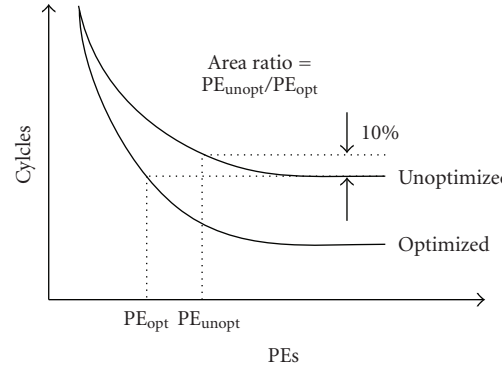


FIGURE 20: How we compute area savings.

cycle count of the best possible application performance for the unoptimized workload (the 10% slack accounts for diminishing returns at larger PE counts). For the fully optimized workload, we identify the PE count (PE_{opt} in Figure 20) that yields performance equivalent to the best unoptimized case. We report these area savings in Figure 21. The ratio of these two PE counts is 3–15 (mean of 9), suggesting these optimizations allow much smaller designs.

To compute energy savings, we use the switching activity factor and network cycles to derive dynamic energy reduction in the network. Switching activity factor is extracted from the number of packets traversing the *Split* and *Merge* units of a Mesh Switch over the duration of the simulation

$$\text{Activity} = \left(\frac{2}{\text{Ports}}\right) \times \left(\frac{\text{Packets}}{\text{Cycles}}\right). \tag{6}$$

In Figure 22, we see a mean 2.7× reduction in dynamic energy at 25 PEs due to reduced switching activity of the optimized workload. While we only show dynamic energy savings at 25 PEs, we observed even higher savings at larger system sizes which have even higher message traffic.

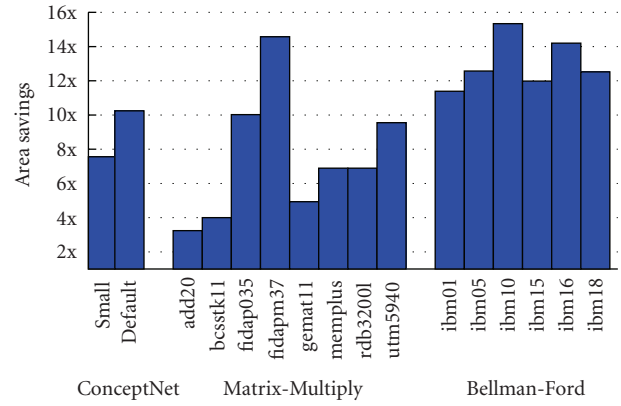


Figure 21: Area ratio to baseline.

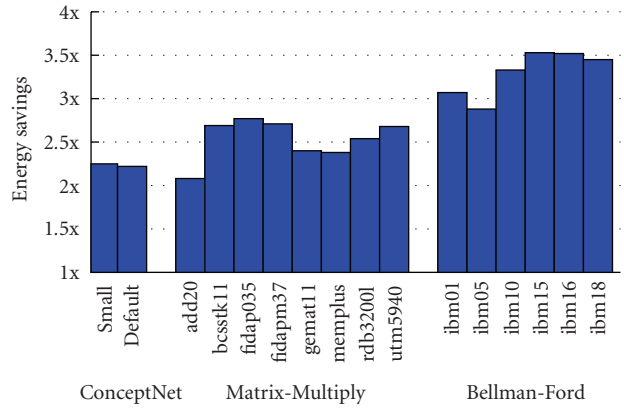


Figure 22: Dynamic energy savings at 25 PEs.

6. Conclusions and Future Work

Large, on-chip networks that support highly-parallel, fine-grained applications will be required to handle heavy message traffic. Load balancing, communication bandwidth, IO serialization, and synchronization costs will play a key role in determining the performance and scalability of such systems. We develop a traffic compiler for sparse graph-oriented workloads to automatically optimize network traffic and minimize these costs. We demonstrate the effectiveness of our traffic compiler over a range of real-world workloads with performance improvements between 1.2× and 22× (3.5× mean), PE count reductions between 3× and 15× (9× mean), and dynamic energy savings between 2× and 3.5× (2.7× mean) for the BSP workloads. We also show speedups of 0.5–13× (geomean 3.6×) for Sparse Matrix Solve (Token Dataflow) workloads when performing a high-quality placement of the dataflow graphs. For large workloads like `cnet-default`, our compiler optimizations were able to extend scalability to 2025 PEs. We observe that the relative impact of our optimizations changes with system size (PE count) and our automated approach can easily adapt to different system sizes. We find that most workloads benefit from *Placement* and *Fine-Grained Synchronization* at large PE counts and from *Clustering* and *Fanout Routing* at small PE counts. While we have demonstrated these optimizations for a specific compute model, the techniques are applicable to an even larger space of possible compute models.

References

[1] R. K. Brayton and C. McMullen, "The decomposition and factorization of boolean expressions," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '82)*, pp. 49–54, April 1982.

[2] D. Yeung and A. Agarwal, "Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient," *SIGPLAN Notices*, vol. 28, no. 7, pp. 187–197, 1993.

[3] M. deLorimier, N. Kapre, N. Mehta et al., "GraphStep: a system architecture for sparse-graph algorithms," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 143–151, April 2006.

[4] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '09)*, pp. 190–198, December 2009.

[5] L. G. Valiant, "Bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[6] T. A. Davis and E. P. Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software*, vol. 37, no. 3, pp. 1–17, 2010.

[7] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 82–91, May 1990.

[8] W. Ho and T. Pinkston, "A methodology for designing efficient on-chip interconnects on well-behaved communication patterns," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, p. 377, 2006.

[9] G. V. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for MPEG-2 video applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 1, pp. 108–119, 2004.

[10] Y. Liu, S. Chakraborty, and W. T. Ooi, "Approximate VCCs: a new characterization of multimedia workloads for system-level MpSoC design," in *Proceedings of the 42nd Design Automation Conference (DAC '05)*, pp. 248–253, June 2005.

[11] V. Soteriou, H. Wang, and L.-S. Peh, "A statistical traffic model for on-chip interconnection networks," in *Proceedings of the 14th International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 104–116, 2006.

[12] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," in *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*, pp. 187–198, July 2006.

[13] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 1025–1040, 2005.

[14] R. Mullins, A. West, and S. Moore, "Low-latency virtual-channel routers for on-chip networks," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, pp. 188–197, June 2004.

[15] N. Kapre, N. Mehta, M. deLorimier et al., "Packet switched vs. time multiplexed FPGA overlay networks," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 205–214, April 2006.

[16] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *Computer*, vol. 26, no. 2, pp. 62–76, 1993.

[17] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Norwell, Mass, USA, 1992.

[18] A. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pp. 37–46, February 1999.

[19] D. Greenfield, A. Banerjee, J. G. Lee, and S. Moore, "Implications of rent's rule for NoC design and its fault-tolerance," in *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS '07)*, pp. 283–294, May 2007.

[20] A. Caldwell, A. Kahng, and I. Markov, "Improved algorithms for hypergraph bipartitioning," in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 661–666, January 2000.

[21] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Enginering Approach*, Elsevier, New York, NY, USA, 2003.

[22] C.-W. Tseng, "Compiler optimizations for eliminating barrier synchronization," *SIGPLAN Notices*, vol. 30, no. 8, pp. 144–155, 1995.

[23] H. Liu and P. Singh, "ConceptNet—a practical commonsense reasoning tool-kit," *BT Technology Journal*, vol. 22, no. 4, pp. 211–226, 2004.

[24] National Institute of Standards and Technology (NIST), "Matrix market," June 2004, http:///math.nist.gov/MatrixMarket/.

[25] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proceedings of the 13th ACM International Symposium on Field Programmable Gate Arrays (FPGA '05)*, pp. 75–85, February 2005.

[26] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs," in *Proceedings of the 8th International Conference on Field-Programmable Technology (FPT '09)*, pp. 190–198, December 2009.

[27] *The Programmable Logic Data Book-CD*, Xilinx, Inc., San Jose, Calif, USA, 2005.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 972375, 16 pages
doi:10.1155/2011/972375

Research Article

On Self-Timed Circuits in Real-Time Systems

Markus Ferringer

Department of Computer Engineering, Embedded Computing Systems Group, Vienna University of Technology, Treitlstraße 3, 1040 Vienna, Austria

Correspondence should be addressed to Markus Ferringer, ferringer@ecs.tuwien.ac.at

Received 6 August 2010; Accepted 8 January 2011

Academic Editor: Michael Hübner

While asynchronous logic has many potential advantages compared to traditional synchronous designs, one of the major drawbacks is its unpredictability with respect to temporal behavior. Having no high-precision oscillator, a self-timed circuit's execution speed is heavily dependent on temperature and supply voltage. Small fluctuations of these parameters already result in noticeable changes of the design's throughput and performance. Without further provisions this jitter makes the use of asynchronous logic hardly feasible for real-time applications. We investigate the temporal characteristics of self-timed circuits regarding their usage in real-time systems, especially the Time-Triggered Protocol. We propose a simple timing model and elaborate a self-adapting circuit which shall derive a suitable notion of time for both bit transmission and protocol execution. We further introduce and analyze our jitter compensation concept, which is a threefold mechanism to keep the asynchronous circuit's notion of time tightly synchronized to the remaining communication participants. To demonstrate the robustness of our solution, we perform different tests and investigate their impact on jitter and frequency stability.

1. Introduction

Asynchronous circuits elegantly overcome some of the limiting issues of their synchronous counterparts. The often-cited potential advantages of asynchronous designs are—among others—reduced power consumption and inherent robustness against changing operating conditions [1, 2]. Recent silicon technology additionally suffers from high parameter variations and high susceptibility to transient faults [3]. Asynchronous (delay insensitive) design offers a solution due to its inherent robustness. A substantial part of this robustness originates in the ability to adapt the speed of operation to the actual propagation delays of the underlying hardware structures, due to the feedback formed by completion detection and handshaking. While asynchronous circuits' adaptive speed is hence a desirable feature with respect to robustness, it becomes a problem in real-time applications that are based on a stable clock and a fixed (worst-case) execution time. Therefore, asynchronous logic is commonly considered inappropriate for such real-time applications, which excludes its use in an important share of fault-tolerant applications that would highly benefit from its robustness. Consequently, it is reasonable to take a closer look at the actual stability and predictability of asynchronous logic's temporal behavior. After all, synchronous designs operate on the same technology, but hide their imperfections with respect to timing behind a strictly time-driven control flow that is based on worst-case timing analysis. This masking provides a convenient, stable abstraction for higher layers. In contrast, asynchronous designs simply allow the variations to happen and propagate them to higher layers. Therefore, the interesting questions are as follows: which character and magnitude do these temporal variations have? Can these variations be tolerated or compensated to allow the usage of self-timed circuits in real-time applications?

In our research project ARTS (Asynchronous Logic in Real-Time Systems) we are aiming to find answers to these questions. The project goal is to design an asynchronous TTP (Time-Triggered Protocol) controller prototype which is able to reliably communicate with a set of synchronous equivalents even under changing operating conditions. TTP was chosen for this reference implementation because it can be considered as an outstanding example for hard real-time applications. In this paper we present results based on our previous and current work. We will investigate

the capabilities of self-timed designs to adapt themselves to changing operating conditions. With respect to our envisioned asynchronous TTP controller we will define a timing model, and also study the characteristics of jitter (and the associated frequency instabilities of the circuit's execution speed) as well as the corresponding compensation mechanisms. We implement and investigate a fully functional transceiver unit, as required for the TTP controller, to demonstrate the capabilities of the proposed solution with respect to TTP's stringent requirements.

The work is structured as follows. In Section 2 we give some important background information on TTP, the research project ARTS, and the used asynchronous design style. Section 3 discusses related work and the basic jitter terminology. Afterwards, Section 4 discusses temporal characteristics of QDI circuits and presents some case studies. The main requirements, properties, and implementation details of the asynchronous time reference generation unit are presented in Section 5, right before experimental results are shown in Section 6. The paper concludes in Section 7 with a short summary and an outlook to future work.

2. Background

2.1. Time-Triggered Protocol. The Time-Triggered Protocol (TTP) has been developed for the demanding requirements of distributed (hard) real-time systems. It provides several sophisticated means to incorporate fault-tolerance and at the same time keep the communication overhead low. TTP uses extensive knowledge of the distributed system to implement its services in a very efficient and flexible way. Real-time systems in general and TTP in particular are described in detail in [4, 5].

A TTP system generally consists of a set of Fail-Silent Units (FSUs), all of which have access to two replicated broadcast communication channels. Usually two FSUs are grouped together to form a Fault-Tolerant Unit (FTU), as illustrated in Figure 1(a). In order to access the communication channel, a TDMA (Time Division Multiple Access) scheme is implemented. As illustrated in Figure 1(b), communication is organized in periodic TDMA rounds, which are further subdivided into various sending slots. Each node has *statically* assigned sending slots, thus the entire schedule (called Message Descriptor List, MEDL) is known at design-time already. Since each node a priori knows when other nodes are expected to access the bus, message collision avoidance, membership service, clock synchronization, and fault detection can be handled without considerable communication overhead. Explicit Bus Guardian (BG) units are used to limit bus access to the node's respective time slots, thereby solving the babbling idiot problem. Global time is calculated by a fault-tolerant, distributed algorithm which analyzes the deviations of the expected and actual arrival times of messages and derives a correction term at each node.

The Time-Triggered Protocol provides very powerful means for developing demanding real-time applications. The highly deterministic and static nature makes it seemingly unsuited for an implementation based on asynchronous logic, as a precise notion of time (which is usually provided by a crystal oscillator in synchronous systems) is missing. Hence, these properties also make TTP an interesting and challenging topic for our exploration of predictability of self-timed logic.

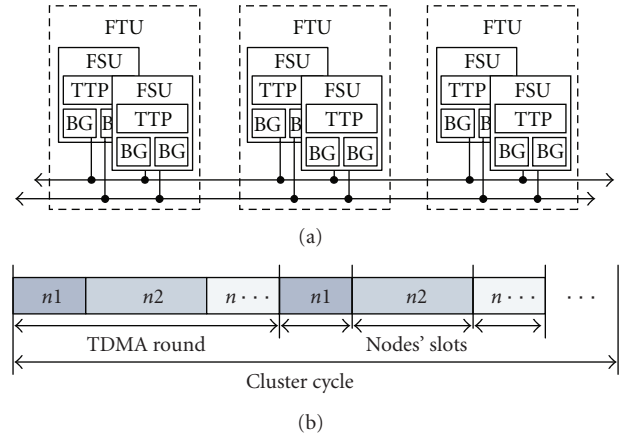


Figure 1: TTP system structure (a) and TDMA communication scheme (b).

2.2. ARTS Project. The aim of the ARTS (Asynchronous Logic in Real-Time Systems) research project is to investigate the temporal predictability and stability of asynchronous (QDI, quasi delay insensitive) hardware designs. More specifically, we want to compile models for the timing uncertainties of hardware execution times and extend these to make quantitative statements on the timing behavior of self-timed circuits. Our hope is that the theoretical and experimental analyses will also provide indications for improving the temporal stability of self-timed circuits.

The central concern for the project is the predictability of asynchronous logic with respect to its temporal properties. We therefore investigate jitter sources (e.g., data-dependencies, voltage fluctuations, temperature drift) and classify their impact on the execution time. Using an adequate model allows us to identify critical parts in the circuit and implement measures for compensation. Another issue concerns timeliness itself, as without a reference clock we do not have an *absolute* notion of time. Instead, we will use the strict periodicity of TTP to continuously resynchronize to the system and derive a time-base for message transfer.

The tangible project result shall be an asynchronous implementation of a TTP controller operating in an ensemble of conventional, synchronous controllers as illustrated in Figure 2(a). It is evident from the explanation in Section 2.1 that TTP's static bus access schedule as well as its clock synchronization and data transmission protocol rely on the stability of the constituent nodes' local clock sources. Therefore it seems quite daring to implement the controller logic in an asynchronous design style. However, moving such a deeply synchronous application to an asynchronous implementation is an interesting and informative challenge of its own, and—if we are successful—a very convincing

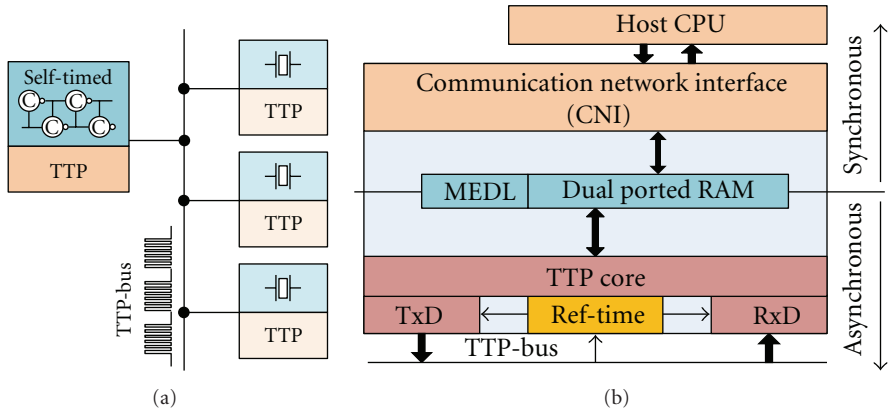


Figure 2: ARTS system setup (a) and TTP-node block diagram (b).

case study for demonstrating the temporal predictability of asynchronous logic. In this context we need to solve two fundamental problems, whereby it is mandatory to derive quantitative boundaries for the attained properties in both cases.

(1) We have to make our design operate stable enough to meet TTP's stringent requirements on execution times and jitter. Conceptually this issue has to do with the fact that control flow in self-timed circuits is flexible and not strictly time-driven, as in the synchronous paradigm.

(2) We have to provide a stable local time reference for bit timing and bus access. This issue originates from the fact that in synchronous systems the clock sources can also be used as time reference—which is missing in a self-timed approach.

Figure 2(b) shows the internal structure of the envisioned asynchronous TTP node. It is very similar to the existing TTP communication chip from TTTech Computertechnik AG [6], our project partner. The interface to the controlling host-CPU (CNI, Communication Network Interface) will remain synchronous, thus existing soft and hardwaresolutions for TTP can be used without modifications. However, the "central" parts (bus access, receiver-unit, transmitter-unit, macrotick generation, TTP-services, etc.) of the controller will be replaced by asynchronous implementations. Similar to the existing FPGA-based controller, a dual-ported RAM separates the CNI from the TTP-core, thereby forming not only a temporal firewall [5], but also splitting the synchronous from the asynchronous parts. (Both host-CPU and TTP controller share a global time-base, thus concurrent access to the dual-ported RAM can be controlled to avoid collisions.) It should be noted that the asynchronous controller implementation is intended solely as an academic case study and not as a prototype for an industrial design. However, a successful project outcome might considerably increase acceptance and even open new fields of applications for asynchronous designs.

The main challenge relies in the method of resynchronization, as the controller will use the data stream provided by the other communication participants to dynamically adapt its internal time reference. The chosen solution is to use a free-running, self-timed counter for measuring the duration of external events of known length (i.e., single bits in the communication stream). The so gained reference measurement can in turn be used to generate ticks with the period of the observed event. This local time-base should enable the asynchronous node to derive a sufficiently accurate time reference for both low-level communication (bit-timing, data transfer) as well as high-level services (e.g., macrotick generation). The disturbing impact of environmental fluctuations is automatically compensated over time, because periodic resynchronization will lead to different reference measurements, depending on the current speed of the counter circuit.

2.3. Asynchronous Design Style. Our focus is on QDI (quasi delay insensitive) circuits, as they exhibit more pronounced "asynchronous" properties than bounded delay circuits. More specifically, we use the level-encoded dual-rail approach (LEDR [7, 8], used in Phased Logic [9], e.g.), which represents a logic signal on two physical wires. We prefer the more complex 2-phase implementation over the popular 4-phase protocol [2, 10], as it is more elegant and we have already gained some practical experience with it. LEDR periodically alternates between two disjoint code sets for representing logic "HI" and "LO" (two *phases* $\varphi 0$ and $\varphi 1$, see Figure 3(a)), thus avoiding the need to insert NULL tokens as spacers between subsequent data items.

In this approach, completion detection comes down to a check whether the phases of all associated signals have changed and match. As usual, handshaking between register/pipeline stages is performed by virtue of a *capture done* signal. Figure 3(b) shows an exemplary LEDR circuit with two sequential registers ($\text{reg}_{0\to 2}$) and a feedback loop ($\text{reg}_{0\to 1\to 0}$). Direct feedback (i.e., without a shadow register like reg_1) is not possible, as race conditions and deadlocks may occur when a register issues its own acknowledges and requests. Also notice the phase inverter in the feedback path.

If external single-rail inputs are to be fed into LEDR circuits, special "interface gates" [9] must properly align

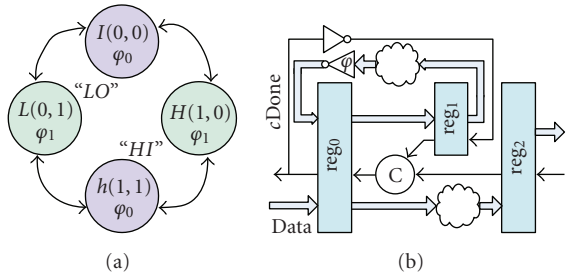


Figure 3: LEDR coding (a) and exemplary LEDR circuit structure (b).

("synchronize") these to the LEDR operation cycles. Concrete implementations and variations of such interface gates are described in [11].

The two-rail coding as well as the adaptive timing make LEDR circuits very robust against faults and changing operating conditions, unfortunately at the cost of increased area consumption and reduced performance. In principle, LEDR circuits belong to the class of QDI circuits, and typically the mandatory delay constraints are hidden inside the basic building blocks, while the interfaces between these modules are considered delay insensitive (i.e., unconstrained with respect to their timing delays).

2.4. Design Flow. For efficiently implementing complex asynchronous circuits it is of upmost importance that there exists a sophisticated software tool support. This is the main reason why the special area of automated synchronous-to-asynchronous converters gains more and more interest. This procedure, which is also called *desynchronization*, has the major advantage that designers need not care about the fallacies and pitfalls of asynchronous design directly. The systems can be described in a synchronous fashion, and an (optimally fully) automated tool chain converts it accordingly. A good overview to existing design flows using automated circuit conversion has recently been presented in [12]. The approach taken at our department [13] is also a form of desynchronization, and is explained in more detail in the next few paragraphs.

The general idea behind automated circuit conversion is to use a suitable circuit representation (one which designers are familiar with), and let a software tool convert the circuit into an asynchronous representation. In the process, the tool needs to identify concurrency, as well as temporal (temporal means the order of events; the exact timing is of course different because of the different design style) and causal dependencies, and must of course guarantee functional and temporal equivalence between the input and output circuits. Using an arbitrary synthesis tool, the synchronous design is compiled into a gate-level netlist consisting only of D-flip-flops and the supported combinational gates. Based on this netlist our tool replaces the all sequential components with asynchronous registers, and all logic gates with their (dual-rail) asynchronous counterparts. After analyzing all dependencies, the tool automatically derives an initial token configuration, inserts shadow registers where necessary and finally produces the asynchronous netlist. This file can now be used for simulation or compilation (technology mapping, placement and routing, ...) with any suitable software. The presented approach has the advantage that it is almost totally automated. In addition, designers are not restricted to a special asynchronous design style. Our tool is currently capable of generating LEDR as well as NCL (Null Convention Logic) circuits.

3. State of the Art

3.1. Related Work. As already mentioned, the focus of this paper will be on how to attain a stable time reference in the context of self-timed logic. From an abstract point of view this problem results from our attempt to insert a self-timed TTP node into an ensemble of otherwise fully synchronous nodes. For the purpose of our project, however, this is an important part of the deal, and situations like this may be encountered in practice as well. Let us first review some common options for building a time reference [14].

(1) *Crystal Oscillators.* This approach exploits mechanical vibrations paired with the piezoelectric effect, which attains highest precision at high frequencies. One of the severe drawbacks of crystal oscillators is their incompatibility with standard process technology. They need to be attached externally, which is area consuming, costly, and unreliable (as soldering contacts may break in harsh environments). Another drawback is the relatively long startup time of crystal oscillators (in the range of about ten milliseconds). Furthermore, susceptibility to mechanical vibrations, humidity and shock are considerably higher compared to alternative solutions.

(2) *RC Oscillators [15].* Here the time constant associated with charging a capacitance over a resistor is used as a time reference. While resistors and capacitors are very cheap components and can be integrated on silicon, they suffer from high fabrication variations as well as relatively large temperature and supply voltage dependencies. RC oscillators provide a good alternative to external crystal resonators, as long as high frequency and high-precision are not major concerns.

(3) *Integrated Silicon (Ring-)Oscillators.* In this approach the oscillations produced by a negative digital feedback loop, usually a ring spanning an odd number of inverters, are exploited [16]. The implementation is fully compatible with the CMOS fabrication process, but the produced frequency is determined by the delay path through the closed loop and hence heavily dependent on fabrication variations, supply voltage, and temperature. Different circuit structures are conceivable, from a simple chain of inverters to more complex solutions, as for example a free running self-timed circuit based on micropipelines [17].

(4) *Distributed Clock Generation [18]*. For the use in embedded systems, distributed algorithms can be implemented to generate clock signals in a fault-tolerant distributed way. Each node can have its own clock source (i.e., an instance of the distributed algorithm) that remains in synchrony with the others within some known precision bounds. This approach can be viewed as a complex distributed silicon ring oscillator, inheriting the properties of the method above, but being more complex and robust due to the desired fault tolerance.

Obviously, the most straightforward solution in our setting is the use of the self-timed circuit's natural processing cycles as a time reference. Being able to exploit the oscillations of self-timed circuits as a replacement for the crystal oscillator directly meets our project goals. We do not consider solution (4) as it is overly complex for our application.

Another important alternative for generating precise time references are self-timed oscillator rings, which seem to be perfectly suited for the chosen asynchronous design methodology. In contrast to (3) they are based on Sutherland's micropipelines instead of a simple chain of cascaded inverters. A lot of research has been conducted on self-timed oscillator rings. For example, in [19] a methodology for using self-timed circuitry for global clocking has been proposed. The same authors also used basic asynchronous FIFO stages to generate multiple phase-shifted clock signals for high-precision timing in [17]. Furthermore, it has been found that event spacing in self-timed oscillator rings can be controlled [16, 20]. The Charlie- and the drafting-effects have thereby been identified as major forces controlling event spacing in self-timed rings [17, 21].

3.2. Jitter. In synchronous systems we have the abstraction of an equally spaced time grid to which all transitions are aligned, and all deviations from this ideal behavior are commonly subsumed under the term jitter. Often jitter is associated with a synchronous clock source like a crystal oscillator, where it is obviously an undesired effect. Consequently, attempts have been made to identify the different sources and effects of jitter in order to mitigate the most relevant ones.

The literature generally distinguishes deterministic and random (indeterministic) jitter, as illustrated in Figure 4. The term *random* thereby refers to the statistical and random characteristics of jitter, and by that the magnitude is unbounded. In contrast, *deterministic* jitter sources have well-defined origins, are always bounded in magnitude, can be predicted, and are thus reproducible. (Notice that random effects may also have well-defined origins and be reproducible, but this only accounts for their statistical parameters.) The following list shortly explains the most common sources of jitter [22–24]. Notice that they are *not* mutually exclusive—even worse, measurements mostly indicate a combination of several if not all of these types.

(i) *Data-Dependent Jitter (DDJ)* is added to a signal according to the sequence of processed data values. Crosstalk, intersymbol interference, simultaneous switching noise, and so forth, are common sources of DDJ. In a jitter histogram, DDJ can often be identified as multiple separated peaks.

(ii) *Bounded Uncorrelated Jitter (BUJ)* subsumes deterministic jitter sources that are not caused by data-dependencies. Fabrication and process variations are suitable examples.

(iii) *Duty Cycle-Dependent Jitter (DCD)* has its origin in differences in the slopes of rising and falling signal edges. High and low pulses of a periodic signal appear to have different lengths, which manifests as two distinct peaks in the jitter histogram. A similar effect can be observed (even in case of matching slopes) if the decision threshold for binary values is not at 50%.

(iv) *Periodic Jitter (PJ)* is induced by periodic external events, such as switching power supply noise, and is per definition uncorrelated to any data-values. It results in pronounced peaks in FFT plots, for which reason it is also called sinusoidal jitter. In jitter histograms, the characteristic curve of PJ often looks like a bathtub.

(v) *Random Jitter (RJ)* can be seen as the (statistical) sum of multiple uncorrelated random effects (e.g., thermal or supply voltage noise), which is one of the main reasons for the Gaussian-like characteristics in jitter histograms.

With this abstract classification in mind, concrete manifestations of jitter can be defined [23, 24] for periodic signals.

(i) *Timing Jitter* is the deviation of a signal transition from its ideal position.

(ii) *Period Jitter* is the deviation of a signal's period from its nominal value.

(iii) *Cycle-to-Cycle Period Jitter* is the variation in cycle-periods of adjacent cycles.

(iv) *Long-Term (Accumulated) Jitter* is defined as deviation of the measured *multicycle* time-interval from the nominal value. Especially random jitter accumulates over time, and thus its absolute value increases when observing long time intervals.

In a practical circuit we typically observe a superposition of the diverse types of jitter. It is therefore an intricate task to distinguish them in a measurement, even though powerful support by special jitter oscilloscopes is available.

4. Temporal Characteristics

4.1. Stability Characteristics. Measuring jitter effects in asynchronous circuits differs from the synchronous case, because there are no specified reference values available for period or timing in general. After all, it is a desired property of asynchronous logic to adapt its speed of operation to the given conditions. As there is no dedicated clock in QDI circuits, we need to find another way to measure execution

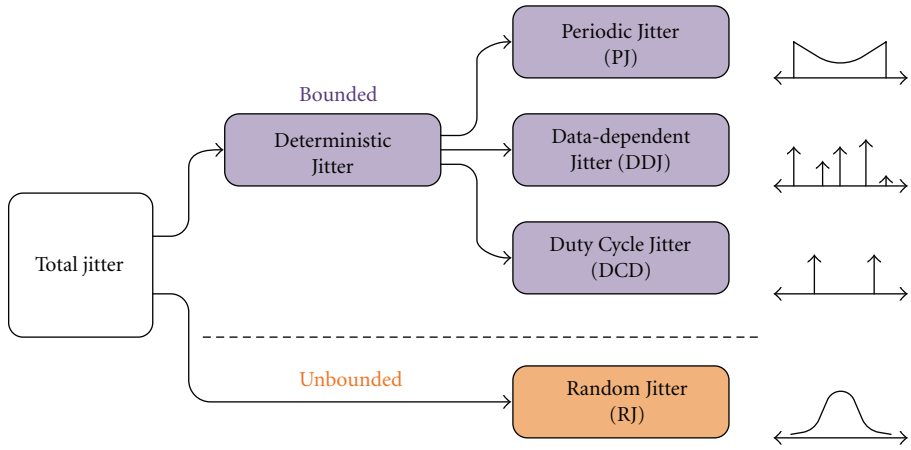


Figure 4: Jitter classification scheme [22].

speed and the associated variations in the durations of single execution steps. To this end, the phase of any register is a suitable measure, as it changes exactly once per execution cycle. The inherent handshaking guarantees the *average* rate of phase changes for all coupled registers to be the same. However, due to the fact that LEDR circuits are "elastic", there may be substantial differences in the execution speeds of adjacent pipeline stages for consecutive cycles. We define *execution period jitter*, or just execution jitter, to be the variation in the durations of phases of a *specific* LEDR-register. In order to classify the frequency stability of the single execution steps and the generated signals, we use Allan variance plots [25, 26], which provide the necessary means for a detailed analysis. Instead of a single number, Allan deviation is usually displayed as graph for gradually increasing durations τ of the averaging window. It therefore combines measures for both short (e.g., execution steps) and long (e.g., generated ticks for bit-timing) term stability in a single plot.

From an abstract point of view, we can categorize jitter into two major groups. On the one hand, *systematic jitter* (DDJ, global voltage and temperature change, e.g.) describes all effects that can be reproduced by our system setup. Consequently, for a given circuit, if we apply the same input transitions in the same state under the same operating conditions, we may expect the delay to be the same as well. If this is not the case, *random jitter* (local voltage and temperature fluctuations, noise, ageing, e.g.) has been experienced. Obviously, the latter cannot be controlled by the system setup. Recalling the different types of jitter from Section 3.2, we consider DDJ and PJ to be systematic, while BUJ, DCD and RJ are considered random. Although BUJ and DCD seem to be systematic after all (they show very little dynamics, which are almost constant over a chip's lifetime), it is hardly possible to influence them by means of system setup. The classification as "random" therefore seems adequate. It can also be expected that they are not a major source of frequency instabilities for QDI circuits. As we cannot control the random effects, we need to focus on systematic jitter when it comes to predictability and reproducibility for our envisioned TTP-controller.

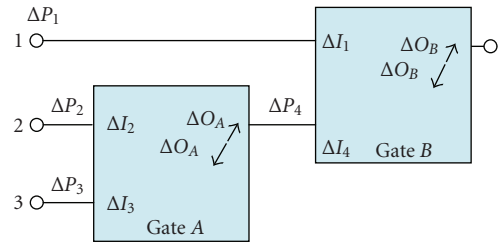


Figure 5: Timing model, example circuit.

(i) *Data-Dependent Execution Jitter (DDEJ)* deals with cases where the actual data values induce (systematic) jitter on a signal.

(ii) Consequently, *Data-Independent Execution Jitter (DIEJ)* subsumes all nondata-dependent systematic jitter effects (global changes of temperature and voltage, e.g.). We also classify PJ to be in this group, as the corresponding sources are data-independent and, typically, deterministic.

Keeping the above classification of jitter sources in mind, we now examine sources of data-dependent and random jitter from a logic designer's point of view. Figure 5 shows an example circuit with two gates A, B, four interconnect delays $\Delta P_{1,2,3,4}$, two input delays $\Delta I_{1,4}$, $\Delta I_{2,3}$ for each gate, and two output delays $\Delta O_{A,B}$. According to practical experience on FPGAs we also distinguish whether a rising or falling output transition occurs, indicated by the up/down arrows. Notice that even for symmetric functions like AND the input delays ΔI are not necessarily the same for all input terminals. Consequently, a transition on input 1 or 2 has a delay $\Delta 1 = \Delta P_1 + \Delta I_1 + \Delta O_B(\uparrow \text{ or } \downarrow)$, and $\Delta 2 = \Delta P_2 + \Delta I_2 + \Delta O_A(\uparrow \text{ or } \downarrow) + \Delta P_4 + \Delta I_4 + \Delta O_B(\uparrow \text{ or } \downarrow)$, respectively.

A more interesting case occurs when inputs 1 and 2 are connected. A transition on this combined input may need $\Delta 1$ one time and $\Delta 2$ another time, depending on which path is enabled by the involved gates (e.g., if the change on input 1 is sufficient for gate B to toggle, independently from A's output). This behavior, in combination with possibly

different output delays $\Delta O\uparrow\downarrow$ for opposite transitions, is the main reason for data-dependent jitter. Since LEDR designs are *strongly indicating*, which means that the realized logic functions always exhibit worst-case performance, the observed data-dependencies have their origin mainly in the internal structure of the single LEDR gates, and not in the Boolean expressions that are implemented by them. Out of this analysis it is easy to see that DDJ is systematic, although difficult to predict for complex paths. Obviously, the introduced delays are not constant, but are subject to so-called "PVT-variations". Fabrication tolerances, local/global supply voltage and temperature variations considerably change the delays. Thereby, deliberate changes in voltage or temperature will cause systematic jitter, while local fluctuations (noise, e.g.) typically result in random jitter. Given that random effects are statistically independent and follow a normal distribution, overall random jitter can be modeled by (statistically) integrating over all delays along a certain path. In order to derive a quantitative measure for data-dependent jitter, the longest and shortest paths through a given circuit must be examined.

As target platform for our prototype implementation we want to use FPGAs, which are not in any way optimized for asynchronous hardware. Which effects must be taken into account when elaborating the delay of LEDR-gates (also refer to Figure 5)? Notice that the above model can be applied hierarchically, thus it can be used for complex elements (composed of basic gates, e.g.) as well.

(1) Gate delays: we have already seen the delays associated with basic gates in Figure 5. These delays can also be applied to FPGAs, where look-up tables (LUTs) are used as basic gates to implement Boolean expressions.

(2) Internal interconnects are necessary if a complex gate cannot be realized out of a single LUT. However, depending on the exact placement and routing, these delays are not the same for all "internal" interconnects. By using precompiled components (hard macros) it can be assumed that all LEDR-gates of the same type also have approximately the same internal interconnect delays.

(3) External interconnects (connections between different LEDR gates) are certainly a central source of delays. Not only do they have considerably different values for different (input-)signals, they can also be expected to be distinctively different for all LEDR-gates, even those of the same type.

(4) Environmental conditions such as temperature drift and voltage fluctuations also influence the circuit's timing. For this model we assume these properties to affect the entire chip homogeneously.

Since we are operating at gate-level (with relatively large delays), transistor-level effects such as the Charlie- or Drafting-effects [20], as well as SSN (Simultaneous Switching Noise [27]) can be neglected. The latter is covered by DDJ anyway. However, there is a comparable behavior for the Charlie-effect at gate-level as well (which directly follows from the delay assumptions we made). We need to look at the relative arrival times of input-transitions at gates. Assume for Figure 5 that $\Delta 1 = 5$ ns and $\Delta 2 = 9$ ns. If both inputs arrive simultaneously, the output is stable after 9 ns. If, however, input 2 arrives 4 ns before input 1, the additional delay *after 1's arrival* is only 5 ns.

Table 1: Example circuits summary.

	4-bit counter	16-bit counter	16-bit LFSR
Logic elements	100	452	276
LEDR-gates	5	41	3
LEDR-registers	$2 * 4$	$2 * 16$	$2 * 16$
Logic depth	3	15	1
Performance	125 MHz (8 ns)	42 MHz (24 ns)	100 MHz (10 ns)
Phase C2C Jitter	3.0 ns (403 ps)	3.5 ns (710 ps)	0 ns (322 ps)
Count period	127 ns	1.6 ms	683 μs
Count C2C Jitter	$\sigma = 113$ ps	$\sigma = 57$ ns	$\sigma = 43$ ns

4.2. Case Studies. We now present three simple circuits and study their characteristics according to the properties elaborated in the previous section. All three designs consist of a free-running, closed-loop LEDR circuit with two registers (one of which being a shadow register with a phase inverter, as direct feedback is not possible, recall Figure 3(b) from Section 2.3). The first two examples are a 4-bit and a 16-bit counter, respectively. Both counters are realized as ripple-carry adders. The third design is a free-running 16-bit LFSR. The characteristic figures of each design are summarized in Table 1. Row "Logic Elements" lists the number of logic cells needed on the FPGA. "LEDR-Gates" and "LEDR-Registers" specify how many dedicated combinational LEDR-gates and registers are used, respectively. "Logic Depth" defines the maximum number of LEDR-gates connected in series, and "Performance" gives the average operating speed of the circuit. In row "Phase C2C Jitter" the mean observed cycle-to-cycle period jitter of single execution steps (phases) is listed, followed by the respective standard deviation in parentheses. "Count Period" shows the duration until counter overflow, and "Count C2C Jitter" finally specifies the standard deviation of cycle-to-cycle period jitter of entire counting-periods (i.e., 2^4 and 2^{16} steps, resp.).

4-Bit Counter. To better illustrate DDEJ, we manually inserted some wire delays in order to obtain the more pronounced jitter histogram of Figure 6 (the values of Table 1 are based on the original circuit). One can see that there are 11 peaks, which are a superpositions of 16 separate peaks, each of which corresponding to one of the 16 possible counter states. It can further be seen that the single humps have (approximately) Gaussian characteristics, but in contrast to real normal distribution they are bounded quite sharply. The distinct peaks are direct consequences of data-dependencies induced by the current state (i.e., the actual counter value) of the circuit.

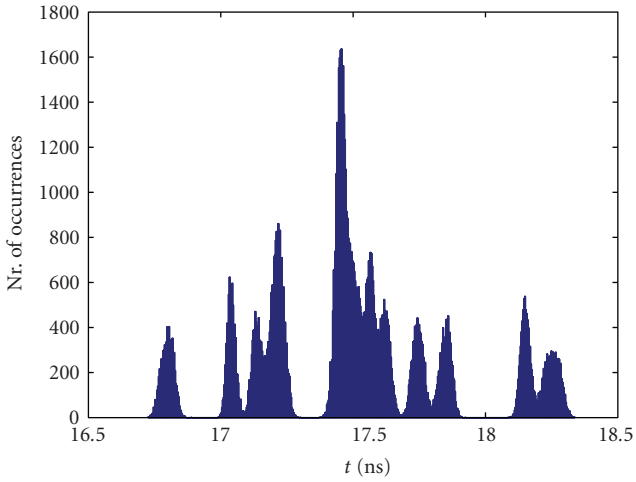


Figure 6: Jitter histogram, 4-bit counter, DDJ.

16-Bit Counter. Figure 7(a) shows the jitter histogram of the counter's cycle-to-cycle execution period jitter. A 16-bit counter has 2^{16} slightly different execution times, and their superpositions result in one big Gaussian-like distribution *for each phase* (therefore, "Phase C2C Jitter" has a mean value other than zero). Again, data-dependencies are responsible for the significant differences in the durations of phases φ_0 and φ_1. There are minor variations in the propagation delays of gates due to the actual phase-values. These accumulate while passing through the logic stages and manifest as two separate peaks in the jitter histogram. For a counter there is also another, systematic effect which further amplifies these differences. The odd phases φ_1 are strictly coupled with odd counter values (as long as there is an even number of counting steps). Therefore, odd phases have considerably more logic ones in their input values than even phases, hence further aggravating the data-dependencies. In addition, Figure 7(b) depicts the cycle-to-cycle jitter of entire counting periods. No data-dependencies (and thus no phase-dependencies) can be observed any more. The graph shown is the accumulated random jitter over 2^{16} execution steps. One could expect the standard deviation of an entire counter period to be (cf. Table 1) $\sigma_{\text{count}} = \sigma_{\text{step}}\sqrt{2^{16}} \approx 182$ ns, but as all data-dependent effects are exactly the same for each complete period, only random jitter remains (which is considerably less, $\sigma_{\text{count}} = 57$ ns).

Linear Feedback Shift Register (LFSR). The internal logic just consists of three XNOR-gates, thus being extremely efficient in terms of performance and area consumption. For reasons of performance, our design uses Galois LFSRs, thus having a maximum logic depth of only one gate equivalent. Consequently, the data-dependent effects are considerably less severe because of the reduced logic depth (no accumulation through logic stages possible). This is also evident in Figure 7(a) and Table 1. The LFSR does not show significant differences for phases $\varphi_{0,1}$ (resulting in a mean value of zero for "Phase C2C Jitter"), and the overall width of the histogram is substantially less compared to the counter.

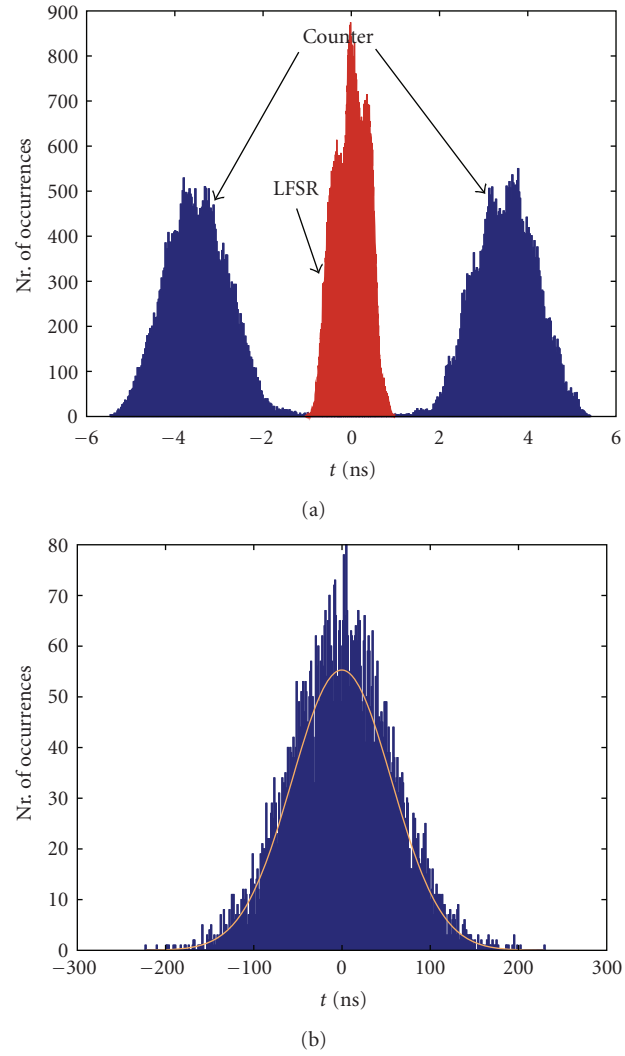


Figure 7: Execution period C2C jitter of 16-bit counter and LFSR (a), C2C jitter histogram of counting periods of 16-bit counter (b).

5. Asynchronous Reference Time

In order to allow for reliable TTP communication, the resulting asynchronous controller must have a precise notion of time. As there is no reliable reference time available in the asynchronous case, we design a circuit that uses the TTP communication stream to derive a suitable, stable time-base. We construct an adjustable tick-generator and periodically synchronize it to *incoming* message-bits. In our configuration, the bitstream of TTP uses Manchester coding, thus there is at least one signal transition for each bit which we can potentially use for recalibration. The Manchester encoding is a line code which represents the logical values 0 and 1 as falling and rising transitions, respectively. Consequently, each bit is transmitted in two successive symbols, thus the needed communication bandwidth is double the data rate.

The top part of Figure 8(a) shows three bits of an exemplary Manchester coded signal, whereby the transitions at 50% of the bit-time define the respective logical values. This encoding scheme has the advantage of being self-clocking, which means that the clock signal can be recovered from the bit stream. From an electrical point of view, Manchester encoding allows for DC-free physical interfaces.

Figure 8(a) further illustrates the properties that our design needs to fulfill. As already mentioned, Manchester coding uses two symbols to transmit a single bit, thus the "featuresize" τ_{ref} of the communication stream is half the actual bit-time τ_{bit}. It can also be seen that the sampling points need to be located at 25% and 75% of τ_{bit}, respectively. We intend to achieve this quarter-bit-alignment by doubling the generated tick-frequency ($\tau_{\text{gen}} = \tau_{\text{ref}}/2$). Consequently, each rising edge of signal ref-time defines an optimal sampling point. As our circuit is implemented asynchronously, the generated reference signal will be subject to jitter. Furthermore, temperature and voltage fluctuations will also change the reference's signal period. It is therefore necessary to make the circuit self-adaptive to changing operating conditions.

5.1. Concept and Requirements. A key problem in our project is how to provide a reference time for the transmission and reception of a digital Manchester-coded bit stream on the communication channel. For already explained reasons we want to use some type of self-clocked loop for this purpose. In order to derive a suitable concept for a solution, let us first compile the requirements.

(1) With a typical 1 MBit/s transmission rate the bit length τ_{bit} is 1 μs, but the Manchester coding exhibits a "feature size" of $0.5\tau_{\text{bit}}$, that is, 500 ns (see Figure 8(a)). As a consequence we have to sample (at least) twice per τ_{bit}, ideally at 25% and 75% of the bit time.

(2) The time reference needs to remain synchronized with the other local references in the system. This requires periodic resynchronization even in the synchronous case. It is therefore mandatory to have a reference whose timing can be precisely adjusted.

(3) In the synchronous case the resolution of the adjustment is determined by the local clock generator. With a typical clock frequency of 40 MHz we have a resolution R of some 25 ns. It seems reasonable to strive for a similar resolution in our case.

(4) Re-synchronization is performed every TDMA round in the synchronous case (state correction). As we expect the asynchronous reference to be considerably less stable than a crystal clock, we have to perform resynchronization more often and provide a rate correction as well.

(5) For the purpose of our study we want to consider all provisions to compensate for the nonideal behavior of our reference. Among these are the elimination of long-term effects by virtue of periodic recalibration, masking of random effects by means of averaging, and avoidance of systematic effects by means of design measures. These points are discussed in more detail in Section 5.2.

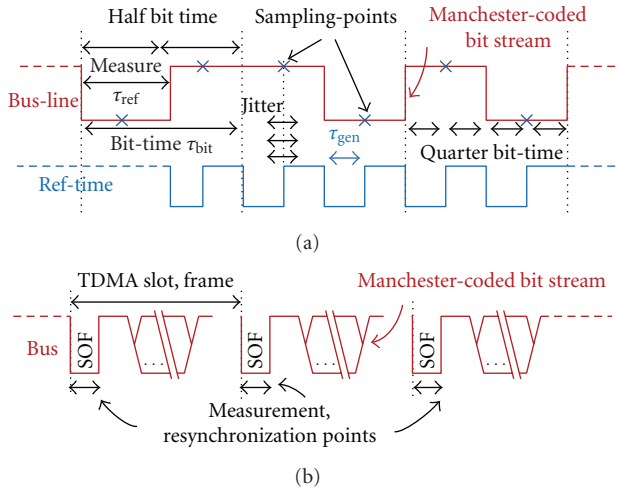


FIGURE 8: Manchester code with sampling points (a), TTP-slots, resynchronization (b).

Requirements (2) and (3) spoil our hope to use the operation cycles of the complete asynchronous controller as a time reference—these are neither fast enough nor adjustable. Therefore we decided to use a separate, small circuit as a time reference that is not dependent on the controller's control flow. A counter suggests itself here to count up to a threshold, whose adjustment already implements the rate correction desired in (4). The price for this decoupling is the need for an explicit synchronization of the operation cycles of the remaining controller logic to this reference.

We will exploit the deterministic nature of TTP and use features of known length in the periodic data stream provided by the other communication participants as a reference to periodically adjust our local timing. (We are well aware that this may become a circular argument in case of all nodes in the system being implemented asynchronously. This is, however, not our intention in the project.) According to (4) we have to adjust the reference as often as possible. We can take advantage of the start of frame (SOF) sequence (HI followed by LO with a length of $0.5\tau_{\text{bit}}$ each) for our measurement. More specifically we use the first HI as our "reference half-bit" (see Figure 8(b)). Measuring just a half-bit cell instead of a much longer interval clearly increases the quantization error. However, longer intervals tend to become dependent on the system configuration (number of involved nodes, configured message length, etc.), thereby considerably complicating the measurement circuitry because more control logic is necessary. This increased complexity not only downgrades performance (which in turn increases the quantization error), but also introduces more jitter and makes timing analysis/predictions substantially harder.

The proposed procedure comprises two phases: (i) *a measurement phase m* during which the reference counter's threshold is determined by starting at 0 at the beginning

of the reference half-bit and simply stopping the counter at the observed half-bit's end. (ii) *a reproduction phase r* during which the observed half-bit length is periodically reproduced by having the counter wrap around to 0 as soon as it reaches the threshold determined above (with a proper initial alignment of 0.25τbit according to (1)).

5.2. Properties. The above procedure implies a *state correction* of the local time, as the internal time is corrected upon detection of the start of frame. In addition a *rate correction* is achieved by adjusting the threshold for every bit. The latter allows for a very tight matching between the current sender's actual bit length and the period of our reference counter (that is subject to variations caused by changing operating conditions, e.g.). *Random* effects are automatically averaged by periodically counting to the measured threshold value. The temporal proximity of measurement and associated reproduction phases is beneficial, as it facilitates an effective compensation of long-term variations (long with respect to the frame length). In other words, the disturbing impact of environmental fluctuations is automatically compensated over time, because the periodic resynchronization events will lead to different reference values depending on the current speed of the counter circuit.

The obvious questions that arise are, which properties does our solution have with respect to frequency stability, and how can changing environmental conditions be dealt with. The following list summarizes all effects that must be taken into account and discusses their impact on our design. To this end, we need to define some parameters for a simple quantification. τ_{bit} has already been introduced as the duration of one Manchester coded bit on the bus. We further define $\tau_{\text{step},m}$ and $\tau_{\text{step},r}$ to be the average durations of execution cycles in *measurement phase* and *reproduction phase*, respectively. τ_{step} is used if the minor difference between these two phases is not important. Finally, τ_{ref} denotes the duration of the reference signal to measure. Consequently, $\text{cnt}_{\text{ref}} = \lfloor \tau_{\text{ref}}/\tau_{\text{step},m} \rfloor$ is the average number of execution steps (i.e., the counter threshold) for the measured pulse of length τ_{ref}.

(i) The *quantization error* can be expressed as $|\text{err}_{\text{quant}}| \leq 2\tau_{\text{step}}$. Since the synchronization circuits presented in [11] provide new data in phase φ_1 only (i.e., only every second phase), the starting and ending transitions introduce a quantization error of up to $\mp 2\tau_{\text{step}}$, respectively. By keeping τ_{step} as low as possible, $\text{err}_{\text{quant}}$ can be improved accordingly. (Using another synchronization circuit may not be possible for all target technologies, but can potentially decrease $|\text{err}_{\text{quant}}|$ to an upper bound of τ_{step}).

(ii) *Systematic errors* are introduced by data-dependent jitter, as mentioned in the previous sections. Two major cases need to be distinguished for our design. (i) The single execution steps while counting up to the measured threshold value show considerable DDJ with respect to each other. (ii) As *measurement* and *reproduction phase* are different states with slightly different register/input values, their average execution speed typically does not match exactly, that is, $\tau_{\text{step},m} \neq \tau_{\text{step},r}$. While (i) does not affect the overall counting period (because the intermediate DDJ is always the same for each counting-cycle), (ii) introduces errors that can only be compensated by clever circuit design or complex correction measures at logic level. The relative deviation of the generated time reference from its measured value can be expressed as factor $f_{\text{dev}} = \tau_{\text{step},r}/\tau_{\text{step},m}$ and should optimally be $f_{\text{dev}} = 1$.

(iii) *Systematic, long-term effects* are mainly caused by slow changes in temperature or supply voltage. Given these fluctuations are slow enough (compared to one TDMA slot), they are compensated automatically at each resynchronization point (cf. Figure 8(b), because depending on the current speed of the counter circuit there will be a different reference/threshold value).

(iv) *Short-term effects* (either random or systematic) that occur faster than a TDMA slot cannot be compensated by our design. In such a case the self-timed circuit accumulates timing errors due to the changed operating speed and will eventually loose synchrony to the remaining cluster.

(v) *Random effects* cannot be compensated easily. However, when averaging over long periods, statistical outliers become less important and frequency stability improves. For our design, averaging occurs automatically over the periodic counting-cycles. Consequently, as for quantization errors, large values for cnt_{ref} are desirable (in contrast to systematic errors, where lower threshold values are preferable in case $f_{\text{dev}} \neq 1$).

Out of these properties, the following essential consequences are derived. The overall systematic error can be written as $\text{err}_{\text{sys}} = \text{err}_{\text{quant}} + (1 - f_{\text{dev}})\text{cnt}_{\text{ref}}\tau_{\text{step},m} = \text{err}_{\text{quant}} + \text{cnt}_{\text{ref}}(\tau_{\text{step},m} - \tau_{\text{step},r})$. While f_{dev} is almost constant for a given circuit, $\text{err}_{\text{quant}}$ can be different for each measurement (thus err_{sys} is variable as well). Furthermore, long-term accumulated jitter (by definition indeterministic and unbounded) causes additional frequency-inaccuracies. Assuming a normal distribution for jitter induced in every execution step, the accumulated jitter can be approximated as $j_{\text{acc}} = N(0, \sigma_{\text{acc}}^2)$ with $\sigma_{\text{acc}}^2 = \text{cnt}_{\text{ref}}\sigma_{\text{step}}^2$ (where σ_{step}^2 is the variance of a single execution step).

Figure 9 illustrates the above types of error. A signal with a high-period of $100\,\mu$s serves as reference for the generation of a $50\,\mu$s time-base (solid line on the very right). The reproduced signal is measured and plotted as jitter histogram (Gaussian-like area). err_{sys} is the deviation of the average signal-period from its nominal value at $50\,\mu$s. On the other hand, j_{acc} manifests as Gaussian jitter with variance σ_{acc}^2 around the mean duration of approximately $49.93\,\mu$s.

5.3. Implementation. The basic structure of the circuit is shown in Figure 10. As one can see, the interface of our design is quite simple. There is only one input (*bus-line*,

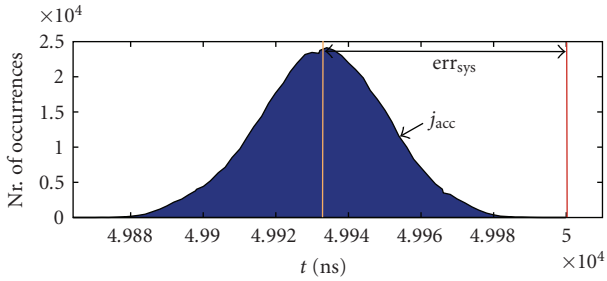


Figure 9: Systematic and random error (jitter).

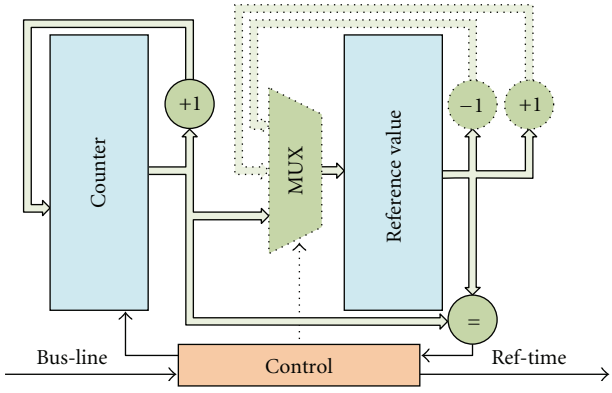


Figure 10: Basic structure of the timer-reference generation circuit.

the receive-line of the TTP bus), as well as one output (ref-time, an asynchronously generated signal with known period). The dashed components and the MUX have been added to the circuit to allow rate correction on a per-bit basis in combination to the absolute measurement of τ_{ref} during the first bit of a message (SOF, Start-of-Frame). If the control block detects the SOF signature, it resets the free-running counter unit. The asynchronous counter periodically increments its own value at a certain (variable) rate, which mainly depends on the circuit structure, placement and routing, and environmental conditions. After time τ_{ref}, the corresponding end of SOF will eventually be detected by the control-block. As a consequence, the current counter value is preserved in register ref-val (reference value) and the counter is restarted. The controller is now able to reproduce the measured low-period τ_{ref} by periodically counting from zero to ref-val, and generating a transition on ref-time for each comparematch. In order to achieve the 25%/75%-alignment, we double the output frequency by simply halving ref-val. The Manchester coded bitstream further allows to use each bit as additional resynchronization point. We therefore exploit the fact that there is at least one transition per bit in the bitstream. We now can compare the expected and actual points in time when transitions on the bus occur, and slightly adjust ref-val by just in-/decrementing it by one, depending on whether the observed transitions where early or late (with respect to our internal expectations). This rate correction significantly increases the reachable accuracy of the time reference generation unit, as the quantization errors are compensated by averaging over time. Furthermore, speed differences which have their origin in the nonmatching speeds of measurement and reproduction phase $\tau_{\text{step},m}$ and $\tau_{\text{step},r}$, can also be compensated with this strategy.

In order to increase the achievable precision of our system, it is important to optimize the circuit of Figure 10 accordingly. The following two changes can be applied to significantly increase performance and reduce the gate count. (With both optimizations applied we are able to reduce the duration of a single execution step from approximately 30 ns down to 14 ns.)

(i) The counter modules (i.e., the incrementers and decrementers) can be replaced by a simple LFSR. This not only reduces the overall gate count, but also significantly increases the maximum speed. LFSRs can be built using at most three XNOR gates for almost all register widths. While ordinary counters have the advantage of producing strictly monotonic outputs, LFSR generate pseudo random numbers. For our application a deterministic order of states is sufficient, monotonicity of counting states is not a requirement. LFSRs are thus a suitable optimization.

(ii) The absolute measurement of the SOF can be removed. For this to work, however, the ref-val needs to be initialized with a value approximately matching the expected operating frequency. If this value is too far off, the module will not be able to synchronize itself to the bit stream correctly. While this optimization also decreases area consumption and at the same time increases the maximum performance (control logic is simpler and 3-way MUX can be replaces by a less complex 2-way MUX), adaption to the correct frequency might take relatively long in case ref-val is far off its optimum value. In addition, finding a feasible initialization value requires thorough timing analysis or measurements on the target platform.

6. Experimental Results

In this section we will present a detailed analysis of the experiments we performed with the proposed circuit from Figure 10. We will vary temperature and operating voltage and monitor the generated time reference under these changing conditions. Simultaneously, we will also evaluate the robustness and effectiveness of the three compensation mechanism implemented in the final design.

(1) *Low-Level State Correction.* Measuring period τ_{ref} of the SOF sequence retrieves an absolute measure of the reference time. However, as only one measurement is performed per message, quantization errors and other systematic, data-dependent delay variations significantly restrict the achievable precision. The possible resolution depends on the speed of the free-running counter, and is at about 25 ns for our current implementation. (Notice that FPGAs are

not in any way optimized for LEDR circuits. Dual-rail encoding introduces not only considerable interconnect delays, but also significant area overhead compared to ordinary synchronous logic.)

(2) *Low-Level Rate Correction*. As the Manchester code always provides a signal transition at 50% of τ_{bit}, we can continuously adapt the measured reference value ref-val. We only allow small changes to ref-val: It is either incremented or decremented by one, depending on whether the expected signal transitions are late or early, respectively. The advantage of this additional correction mechanism is that quantization errors and data-dependent effects are averaged over time, thus increasing precision.

(3) *High-Level Rate Correction*. The softwarestack controlling the message transmission unit can add another level of rate correction. As it knows the expected (from the MEDL) and actual (from the transceiver unit) arrival times of messages, the difference of both can be used to calculate an error-term. High-level services and message transmission can in turn be corrected by this term to achieve even better precision. The maximum resolution which can be achieved by this technique depends on the baud-rate, and is half a bit-time.

Remark. We are well aware that the presented results can only be seen as snapshot for our specific setup and technology. Changing the execution platform will certainly change the specific outcomes of the measurements, as jitter and the corresponding frequency instabilities mainly depend on the circuit structure and the used technology. However, from a qualitative point of view, our results are valid for other platforms and technologies as well. The presented model and the proposed circuits are flexible enough to be applied to different technologies. However, the main problem is the necessity to perform concrete measurements for each target technology in order to obtain meaningful quantitative evaluations. Temporal behavior and specific jitter characteristics are always dependent on fabrication variations and the operating environment. While the theoretical model needs to be complemented by thorough measurements, the proposed circuits are capable of tolerating these (statistical) variations because of the continuous calibration of internal timing.

6.1. Time Reference Generator. Before we start with the message transmission unit, which implements all of the above compensation mechanisms, we want to take a closer look at the basic building block (cf. Figure 10). Clearly, compensation method (3) is not present, as we just investigate the time reference generation unit. This unit does not actually receive or transmit messages, it just generates signal ref-time out of the incoming signal transitions on the TTP bus. The measurement setup is fairly simple. There is a (synchronous) sender, which periodically sends Manchester coded messages. The asynchronous design uses these messages to generate its internal time-reference. All measurements have been taken while the bus was idle. This way, we can observe the circuit's capability of reproducing the measured duration without any disturbing state- or rate correction effects. If not stated otherwise, the measurements are taken at ambient temperature and nominal supply voltage.

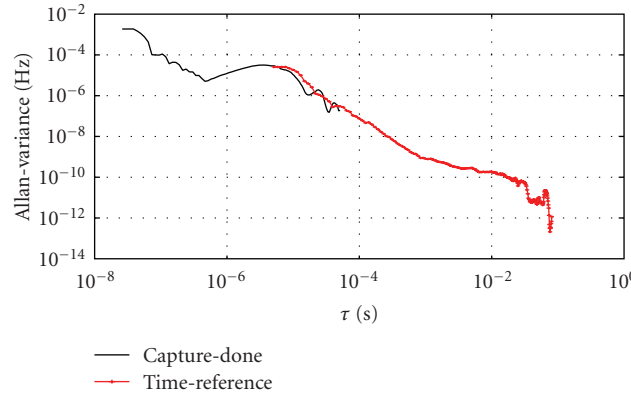


Figure 11: Allan-Variance.

First we take a look at the frequency stability of ref-time and *c*Done. The first part of the Allan-plot in Figure 11, ranging from approximately $2 * 10^{-8}$ s to 10^{-4} s on the *x*-axis, is obtained by monitoring the handshaking signal *capturedone* from a register cell. The second part, which starts at $3 * 10^{-6}$ s and thus slightly overlaps with *capturedone*, has been obtained by measuring ref-time. Notice that it is no coincidence that both parts in the figure almost match in the overlapping section: Signal ref-time is based upon the execution of the low-level hardware and is therefore directly coupled to the respective jitter and stability characteristics. It is obvious from the graph that the stability increases to about 10^{-10} Hz for $\tau \approx 10^{-2}$ s. Furthermore, the reference signal is far more stable than the underlying generation logic (*c*Done), as periodically executing the same operations compensates data-dependent jitter and averages random jitter. Although the underlying low-level signals jitter considerably due to data-dependent jitter, the circuit's output is orders of magnitudes more stable, as these variations are canceled out during the periodic executions.

One of the major benefits of the proposed solution is its robustness to changing operating conditions, thus we additionally vary the environment temperature and observe the changes in the period of ref-time. We heat the system from room temperature to about 83°C, and let it cool down again. Figure 12 compares ref-val to the signal period of ref-time. While the ambient temperature increases, ref-val steadily decreases from 265 down to 256. The period of ref-time makes an approximately 19 ns-step (the duration of a single execution step) each time ref-val changes. During the periods where the changes in execution speed cannot be compensated (because they are too small), ref-time slowly drifts away from the optimum at 5 μs. Without any compensation measures the duration of ref-time would be about 5180 ns at the maximum temperature, instead of being in the range of approximately 5 μs $\pm$ 38 ns (i.e., the duration of $\pm$ two execution steps), no matter what temperature. Notice

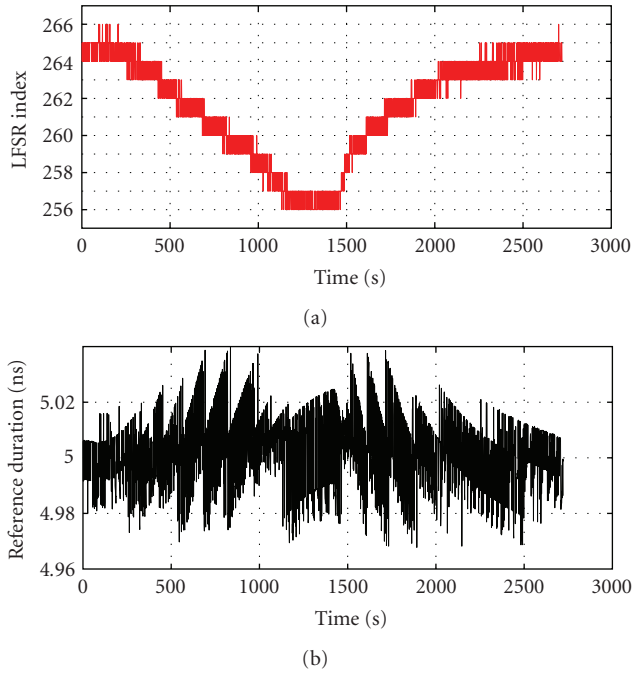


FIGURE 12: ref-val versus timer reference period for temperaturetests.

that the performance of the self-timed circuit decreases by 3.5% at the maximum temperature, which seems to be relatively low, but it certainly is a showstopper for reliable TTP communication.

Far more pronounced delay variations can be obtained by changing the core supply voltage. We applied 0.8 V to 1.68 V in steps of 20 mV core voltage to our FPGA-board. This time the execution speed of our self-timed circuit increased from about 80 ns per step to approximately 15 ns per step, as shown in Figure 13(b). This plot illustrates the jitter histogram on the y-axis versus the FPGA's core supply voltage on the x-axis. Thereby, the densities of the histogram are coded in gray-scale (the darker the denser the distribution). It is evident from the figure that performance increases exponentially with the supply voltage. This illustration also shows other interesting facts. For one, almost all voltages have at least two separate humps in their histograms. These are caused by data-dependencies that originate in the different phases $\varphi_{0,1}$. Furthermore, for low voltages, additional peaks appear in the histograms and the separations between the phases increase as well. This can be explained as data-dependent effects caused by different delays through logic stages that are magnified while the circuit slows down. This property is better illustrated in Figure 13(a), where cycle-to-cycle execution jitter is plotted over the supply voltage. The graph appears almost symmetrically along the x-axis, which is caused by the continuous alternation of phases.

We conclude that varying operating conditions not only affect the speed of asynchronous circuits, but also the respective jitter characteristics. In this perspective, slower circuits tend to have higher jitter, which is further magnified by increased quantization errors due to the low sampling rate.

6.2. Transceiver Unit. The message transmission unit will implement all three compensation methods mentioned at the beginning of Section 6. We intend to use this unit directly in the envisioned asynchronous TTP controller, thus we need to examine the gained precision of this design with respect to timeliness. The interface from the controlling (asynchronous) host to the subdesign of Section 6.1 is realized as dual-ported RAM: Whenever the bus transceiver receives a message, it stores the payload in combination with the receive-timestamp in RAM and issues a receive-interrupt (we define the internal time to be the number of ticks of signal ref-time, i.e., the number of execution steps performed by the bus transceiver unit). Likewise, the host can request to transfer messages by writing the payload and the estimated sending time into RAM and asserting the transmit request.

During reception of messages, the circuit can continuously recalibrate itself to the respective baud rate, as Manchester code provides at least one signal transition per bit. However, between messages and during the asynchronous node's sending slot, resynchronization is not possible. In these phases we need to rely on the correctness of ref-time. The Start-Of-Frame sequence of each message must be initiated during a relatively tight starting window, which is slightly different for all nodes and is continuously adapted by the TTP's distributed clock synchronization algorithm. Failing to hit this starting window is an indication that the node is out-of-sync.

As we are interested in the accuracy of hitting the starting window, we configured the controlling host in a way that it triggers a message-transmission 25 bit-times after the last bit of an incoming message. We simultaneously heated the system from room temperature to about 68°C to check on the expected robustness against the respective delay variations. The results are shown in Figure 14, where the deviation from the optimal sending-point (in units of bit-times) and the operating temperature are plotted against time. Similar to Figure 12 one can see that while the circuit gets warmer (and thus slower), the deviation steadily increases. As soon as the accumulated changes in delay can be compensated by the low-level measurement circuitry (i.e., ref-val decreases), the mean deviation immediately jumps back to about zero. We can see in the figure that the timing error is in the range from approximately -0.1 to $+0.2$ bit-times, which will surely satisfy the needs of TTP.

The next property we are interested in is the circuit's behavior with respect to different baud rates. Although low bitrates have the advantage of minimizing the quantization error, jitter has much more time to accumulate compared to high data rates. It is thus not necessarily true that lower baud rates result in a more stable and precise time reference. On the other hand, if the data rate is too high, it is not possible to reproduce τ_{ref} correctly, and even small changes of the reference value ref-val lead to large relative errors in the resulting signal period. The optimum baud rate will therefore be located somewhere between these extremes. Figure 15 illustrates this by plotting the mean deviations of the optimum sending points versus the bitrate (the "corridor" additionally shows the respective standard deviations). Notice that the y-axis shows the relative deviation in units

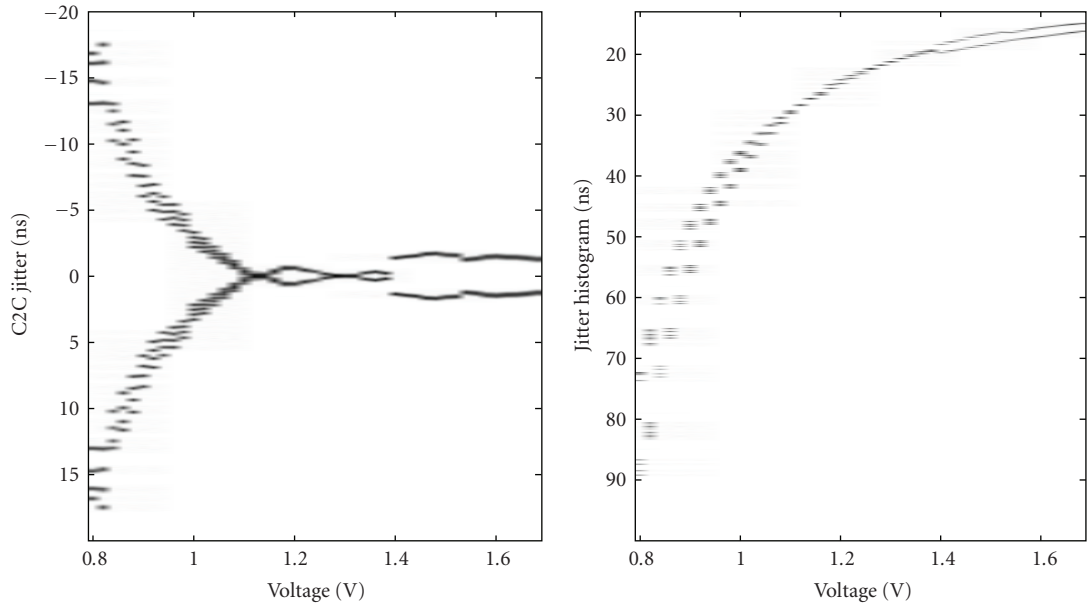


Figure 13: (a) Cycle-to-cycle jitter, (b) jitter histogram.

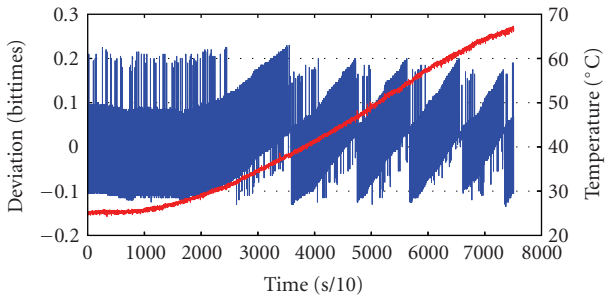


Figure 14: Relative deviation from optimal sending slot and operating temperature.

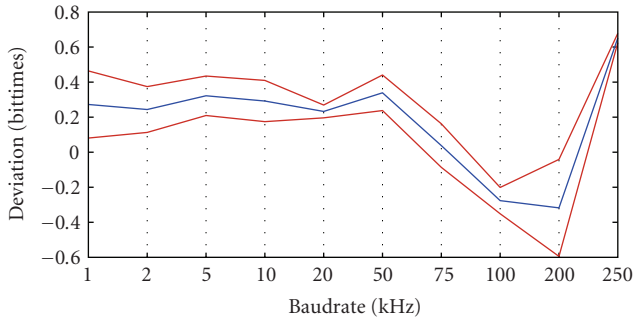


Figure 15: Mean relative deviation from optimal sending point versus baud rate.

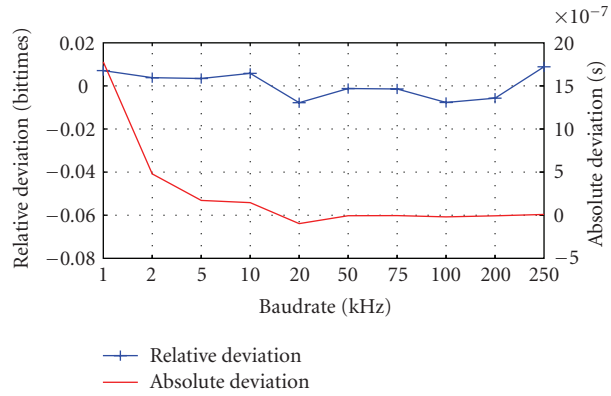


Figure 16: Mean relative/absolute deviation from optimum bit-time.

of bit-times. Therefore, for example, the absolute deviation of the 1 kHz bit-rate is more than 50 times larger than that of 50 kHz. Clearly, TTP does not support baud rates as low as 1 kHz. Reasonable data rates are at least at 100 kHz and above (up to 4 Mbit/s for Manchester coding). Our current setup allows us to use 100 kbit/s for communication with acceptable results. However, we hope to be able to achieve 500 kbit/s in our final system setup (with a more sophisticated development platform and a further optimized design).

Finally, we take a look at the accuracy of the generated time reference for different baud rates. Figure 16 therefore shows the mean relative (again in units of bit-times) and the absolute (in seconds) deviations of the actual reference periods from their nominal values. For all baud rates, the relative deviations are within a range of approximately ±0.01 bit-times, or ±1%, while the absolute timing errors are significantly larger for baud rates below 50 kbit/s.

7. Conclusion

In this paper we introduced the research project ARTS, provided information on the project goals and explained the concept of TTP. We proposed a method of using TTP's bit stream to generate an internal time reference (which is needed for message transfer and most high-level TTP services). With this transceiver unit for Manchester coded messages we performed measurements under changing operating temperatures and voltages. The results clearly show that the proposed architecture works properly. The results further indicate that the achievable precision is in the range of about 1%. This is not a problem while other (synchronous) nodes are transmitting messages, as resynchronization can be performed continuously. However, during message transmission of our node, the design depends on the quality of the generated reference time. Our measurement show that we are able to hit the optimum sending point with a precision of approximately ± 0.3 bit-times (assuming an interframe gap of 25 bits), which should be enough for the remaining nodes to accept the messages.

The presented approach is of course not only limited to TTP, although it clearly is optimized for TTP's unique properties. For instance, receiver and transmitter modules for all kinds of asynchronous serial protocols (e.g., UART) can be implemented using our solution. The important—and probably limiting—part is, independently of the application, to find suitable resynchronization points or patterns. While these are provided by TTP on a periodic, regular and fixed basis, a start-bit of defined length or a special bit pattern could be used for other communication protocols. Generally speaking, self-timed or (Q)DI asynchronous circuits are difficult to use if strictly timed actions need to be performed, because there are no events of defined duration available. With our approach, however, a basic notion of time can be established even in the absence of a highly stable clock signal.

There still is much work to be done. The presented temperature and voltage tests are only a relatively small subset of tests that can be performed. One of the most interesting questions concerns the dynamics of changing operating conditions. How rapidly and aggressively can the environment change for the asynchronous TTP controller to still maintain synchrony with the remaining system? It should be clear from our approach that an answer to this question can only be given with respect to the concrete TTP schedule, as message lengths, interframe gaps, baud rate, and so forth. directly influence the achievable precision of our solution. The next steps of the project plan include the integration of the presented transceiver unit into an asynchronous microprocessor, the implementation of the corresponding software stack, and the interface to the (external) application host controller. Once the practical challenges are finished, thorough investigations of precision, reliability and robustness of our asynchronous controller will be performed.

Acknowledgment

The ARTS project receives funding from the FIT-IT program of the Austrian Federal Ministry of Transport, Innovation and Technology (bm:vit, http://www.bmvit.gv.at/), project no. 813578.

References

[1] C. J. Myers, *Asynchronous Circuit Design*, John Wiley & Sons, New York, NY, USA, 2001.

[2] J. Sparso and S. Furber, *Principles of Asynchronous Circuit Design—A Systems Perspective*, Kluwer Academic Publishers, Boston, Mass, USA, 2001.

[3] N. Miskov-Zivanov and D. Marculescu, "A systematic approach to modeling and analysis of transient faults in logic circuits," in *Proceedings of the 10th International Symposium on Quality Electronic Design (ISQED '09)*, pp. 408–413, March 2009.

[4] H. Kopetz and G. Grundsteidl, "TPP—a time-triggered protocol for fault-tolerant real-time systems," in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS '93)*, pp. 524–533, June 1993.

[5] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Boston, Mass, USA, 1997.

[6] *AS8202NF—TTP-C2NF Communication Controller*, TTTech Computertechnik AG, Revision 2.1 edition, http://www.austriamicrosystems.com/.

[7] M. E. Dean, T. E. Williams, and D. L. Dill, "Efficient self-timing with level-encoded 2-phase dual-rail (LEDR)," in *Proceedings of the University of California/Santa Cruz Conference on Advanced Research in VLSI*, pp. 55–70, MIT Press, 1991.

[8] A. J. McAuley, "Four state asynchronous architectures," *IEEE Transactions on Computers*, vol. 41, no. 2, pp. 129–142, 1992.

[9] D. H. Linder and J. C. Harden, "Phased logic: supporting the synchronous design paradigm with delay-insensitive circuitry," *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 1031–1044, 1996.

[10] K. M. Fant and S. A. Brandt, "NULL Convention Logic^{TM}: a complete and consistent logic for asynchronous digital circuit synthesis," in *Proceedings of International Conference on Application-Specific Systems, Architectures and Processors (ASAP '96)*, pp. 261–273, Chicago, Ill, USA, August 1996.

[11] M. Ferringer, "Coupling asynchronous signals into asynchronous logic," in *Proceedings of the 17th Austrian Workshop on Microelectronics (Austrochip '09)*, Graz, Austria, October 2009.

[12] M. Simlastik and V. Stopjakova, "Automated synchronous-to-asynchronous circuits conversion: a survey," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, vol. 5349 of *Lecture Notes in Computer Science*, pp. 348–358, 2009.

[13] J. Lechner, *Implementation of a design tool for generation of FSL circuits*, M.S. thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2008.

[14] Maxim-IC, "Microcontroller Clock-Crystal, Resonator, RC Oscillator, or Silicon Oscillator?" application Note 2154, 2003, http://www.maximic.com/appnotes.cfm/an_pk/2154.

[15] F. Bala and T. Nandy, "Programmable high frequency RC oscillator," in *Proceedings of the 18th International Conference on VLSI Design: Power Aware Design of VLSI Systems*, pp. 511–515, January 2005.

[16] A. J. Winstanley, A. Garivier, and M. R. Greenstreet, "An event spacing experiment," in *Proceedings of International Symposium on Asynchronous Circuits and Systems*, pp. 47–56, Manchester, UK, April 2002.

[17] S. Fairbanks and S. Moore, "Analog micropipeline rings for high precision timing," in *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, vol. 10, pp. 41–50, April 2004.

[18] M. Ferringer, G. Fuchs, A. Steininger, and G. Kempf, "VLSI implementation of a fault-tolerant distributed clock generation," in *Proceedings of the 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '06)*, pp. 563–571, Arlington, Va, USA, October 2006.

[19] S. Fairbanks and S. Moore, "Self-timed circuitry for global clocking," in *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC '05)*, pp. 86–96, March 2005.

[20] V. Zebilis and C. P. Sotiriou, "Controlling event spacing in self-timed rings," in *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC '05)*, pp. 109–115, March 2005.

[21] J. Ebergen, S. Fairbanks, and I. Sutherland, "Predicting performance of micropipelines using charlie diagrams," in *Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 238–246, San Deigo, Calif, USA, March-April 1998.

[22] Tektronix, "Understanding and Characterizing Timing Jitter," 2003.

[23] M. Shimanouchi, "An approach to consistent jitter modeling for various jitter aspects and measurement methods," in *Proceedings of International Test Conference*, pp. 848–857, Baltimore, Md, USA, October-November 2001.

[24] I. Zamek and S. Zamek, "Definitions of jitter measurement terms and relationships," in *Proceedings of IEEE International Test Conference (ITC '05)*, pp. 25–34, November 2005.

[25] D. W. Allan, N. Ashby, and C. C. Hodge, *The Science of Timekeeping*, application Note 1289, 1997, http://www.allanstime.com/Publications/DWA/Science_Timekeeping/TheScienceOfTimekeeping.pdf.

[26] D. A. Howe, "Interpreting oscillatory frequency stability plots," in *Proceedings of the IEEE International Frequency Control Symposium and PDA Exhibition*, pp. 725–732, May 2002.

[27] B. Butka and R. Morley, "Simultaneous switching noise and safety critical airborne hardware," in *Proceedings of IEEE Southeastcon*, pp. 439–442, March 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 952560, 14 pages
doi:10.1155/2011/952560

Research Article

Experiment Centric Teaching for Reconfigurable Processors

Loïc Lagadec, Damien Picard, Youenn Corre, and Pierre-Yves Lucas

Université de Brest, CNRS, UMR 3192 Lab-STICC, ISSTB, 20 avenue Le Gorgeu, 29285 Brest, France

Correspondence should be addressed to Loïc Lagadec, loic.lagadec@univ-brest.fr

Received 22 July 2010; Accepted 17 December 2010

Academic Editor: Michael Hübner

This paper presents a setup for teaching configware to master students. Our approach focuses on experiment and leaning-by-doing while being supported by research activity. The central project we submit to students addresses building up a simple RISC processor, that supports an extensible instructions set thanks to its reconfigurable functional unit. The originality comes from that the students make use of the Biniou framework. Biniou is a research tool which approach covers tasks ranging from describing the RFU, synthesizing it as VHDL code, and implementing applications over it. Once done, students exhibit a deep understanding of the domain, ensuring the ability to fast adapt to state-of-the-art techniques.

1. Introduction

Innovative lectures and lab courses are required to offer high quality training in the field of configware. Being either an electrical engineer (EE) or a computer scientist (CS) expert will not be enough to meet the needs we foresee in terms of interdisciplinary for the future. As teachers, our goal is not to output Computer-Assisted-Design (CAD) end-users but highly educated experts, who will easily self-adapt to new technologies.

Our contribution to this in-depth rethinking of curricula goes through providing cross expertise training centered around CAD environments design. CAD tools embed the full expertise both from an architectural and from an algorithmic point of view. Affording the design of CAD environments ensures a full understanding of the domain.

As teachers, we make use of some research tools we have developed, that offer a full design suite for reconfigurable accelerators. The key principle behind this is to let students design and implement simple schemes (processors, processor-to-accelerator coupling, etc.) while taking advantage of research tools that promote high productivity. After students have manipulated these toys examples, they show a promising learning curve when addressing state-of-the-art technology (processor soft cores, Xilinx design suite, FSL Fast Serial Links, etc.). This second stage is when performances issue arises. At this point, some discussions happen: fine-versus coarse-grained accelerators, compiler friendly architecture, reconfigurable functional unit versus coprocessor, and so forth.

Splitting the learning activities in such a way emphasizes simplicity. A first consideration is that a simple design always takes less time to finish than a complex one, exhibits more readability, and offers a better support for further refactoring. Another thing about simple designs is that they require knowledge to recognize. Knowledge is different from information. Information is what you get as a student, when gaining access to a lecture. However, you can have plenty of information and no knowledge. Knowledge is insight into your problem domain that develops over time. Our teaching approach aims at accompanying students from information to knowledge.

This paper reports this experience. The rest of the paper is organized as follows: Section 2 introduces the lecture's context along with the experiment centric approach we followed. Section 3 focuses on the project we submit to students. Section 4 shifts from the toy example to a more realistic scope. Section 5 summarizes the benefits of our approach.

2. Experiment Centric Teaching

2.1. Local Scope

2.1.1. Local Curriculum. The master curriculum "Software for Embedded Systems" opened two years ago at the

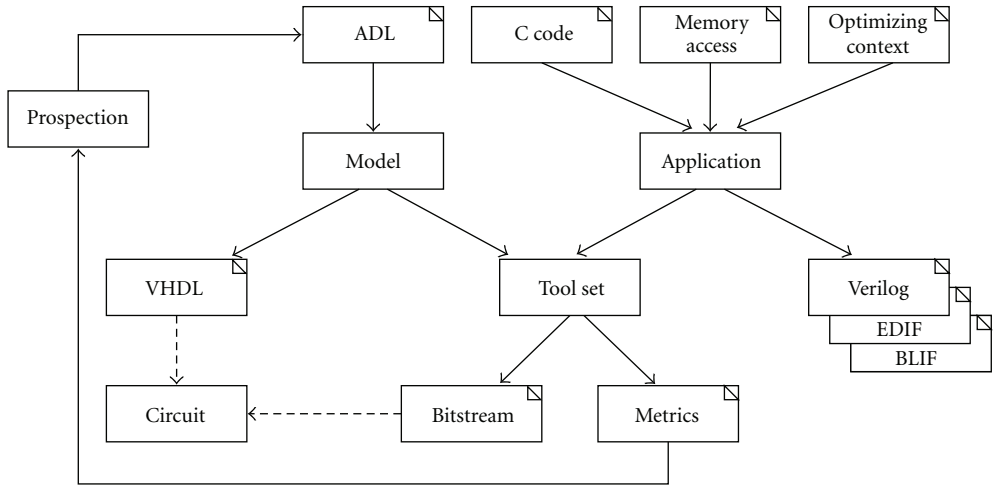


Figure 1: Overview of the Biniou flow.

university of Brest. This curriculum addresses emerging trends in embedded systems and highly focuses on reconfigurable embedded systems, with a set of courses for teaching hardware/configware/software codesign. The master gathers students from both CS and EE former curricula. The current master size is 12, coming from half a dozen countries. Half of the students are former local students hence own a sound background in terms of CS but suffer from lacks in electronic system design.

The reconfigurable computing courses are organized around two main topics covering the hardware (architectures) and software (CAD tools and compiler basics) aspects. These courses enable students to build from their previous knowledge a cross-expertise giving a complete vision of the domain.

A strength of this teaching approach is to partially rely on a research environment rather than purely on Xilinx hands-on tutorials. This offers the opportunity to exercise internal changes on algorithms and architectures, and to address both state-of-the art concepts and both some more prospective topics such as innovative—and still confidential in the industry—architectural trends.

First an overview of the reconfigurable computing (RC) landscape is introduced. Both industrial and academic architectural solutions are considered. This course is structured in three parts:

(i) overview of RC for embedded systems (2 sessions),

(ii) virtualization techniques for RC (2 sessions),

(iii) Modeling and generation of reconfigurable architectures (1 session). The second item addresses both state-of-the-art tools and algorithms in one hand as well as locally designed tools in another hand. The key idea is that students tend towards learning classical (or vendors's) tools so that they can bring a direct added-value to any employer of the field, hence get in an interesting and well-paid job.

However, tools obviously encapsulate the whole domain-specific expertise, and letting students "open the box" closes the gap between "lambda users" and experts. This takes up the challenge of providing a valuable and innovative curriculum. Obviously a single class is not wide enough to address all the-above mentioned items, but this course is closely integrated with some others such as "Numeric and Symbolic synthesis" or "Test & Simulation".

2.1.2. Legacy CAD Development. The research group behind this initiative is the Architectures & Systems team from the Lab-STICC (UMR 3192). This group owns a legacy expertise in designing parallel reconfigurable processor (the Armen [1] project was initiated in 1991) but has been focusing on CAD environment developments (Madeo framework [2]) for the past 15 years. The Madeo framework is an open and extensible modeling environment that allows to represent reconfigurable architectures then acts as a one-stop shopping point providing basic functionality to the programmer (place&route, floorplanning, simulation, etc.). The Madeo project ended in 2006 while being integrated as facilities in a new framework. This new framework, named Biniou, embeds additional capabilities such as, from the hardware side, VHDL export of the modeled architecture and, from the software side, wider interchange format and extended synthesis support. Biniou targets reconfigurable System-On-Chip (SOCs) design and offers middleware facilities to favor a modular design of reconfigurable IPs within the SOC.

Figure 1 provides an overview of Biniou. In the application side (right) an application is specified as C-code, memory access patterns and some optimizing contexts we use to tailor the application. This side outputs some post-synthesis files conforming to mainstream formats (Verilog, EDIF, BLIF, PLA). Results can be further processed by the Biniou Place and Route (P&R) layer to produce a bitstream. Of course the bitstream matches the specification of the underlying reconfigurable target, being the target modeled using a specific Architecture Description Language (ADL). A model is issued on which the P&R layer can operate as previously mentioned, and a behavioral VHDL description

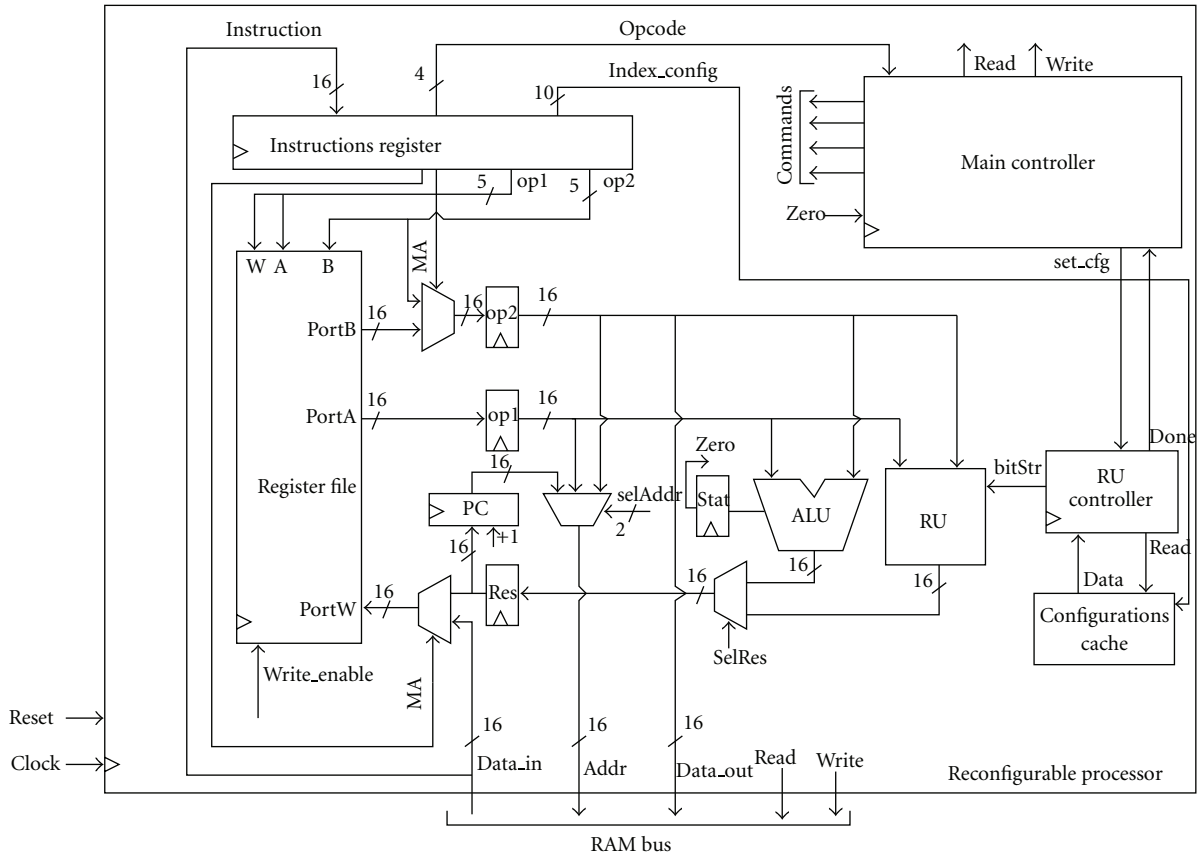


Figure 2: Schematic of the entire reconfigurable processor.

of the target is generated for simulation purposes and FPGA implementation.

Once a bitstream is generated out of the application specification, the designer can download it to configure its platform.

Also some debugging facilities can be added either in the architecture itself or as parts of the application [3].

2.1.3. From Research to Teaching. Biniou has been exercised as a teaching platform for Master 2 students. This happened in a reconfigurable computing course. In addition to lectures, students practice reconfigurable computing through practical sessions and exercise their new skills through a project. This project covers VHDL hand-writing, reconfigurable architecture modeling and programming, code generation, and modules assembly in order to exhibit a simple processor with a reconfigurable functional unit. This extra-unit allows to extend its instructions set.

2.2. Practical Sessions. Practical sessions are organized as three activities. The first activity is to gather documentation and publications related to a particular aspect of the course; the students have to present their short bibliographic study individually in front of the whole class.

The second activity is centered around algorithms used to implement applications over a reconfigurable architecture: point-to-point and global routers, floorplanners, placers. Some data structures such as Transitive Closure Graphs (TCG) are introduced later on in order to point out the need for refactoring and design patterns use [4]. This bridges the software expertise to the covered domain (CAD tools for reconfigurable architecture).

The third activity is related to tools and formats. Three slots are dedicated to VHDL that most of the students do not know. Manual description of fine grained reconfigurable architecture is introduced within this amount of time.

Some sessions are dedicated to practicing required tools; students manipulate logic synthesis tools (SIS [5], ABC), file formats conversion (Verilog, EDIF, BLIF, PLA), and behavioral synthesis according to some data access pattern (Biniou). We also offer a web-based tool [6] to output RTL netlist that students use to exercise several options for netlist generation.

Students create their own FPGA using Biniou, that is further reused in the project under a tuned up version.

2.3. Project Description. The project consists in designing a simple RISC processor, that can perform spatial execution through a Reconfigurable Functional Unit (RFU).

Coupling an RFU along with a processor to get a reconfigurable processor is one out of other alternatives for accelerating intensive tasks. The concept of instruction set metamorphosis [7] is defined and a set of architectures are described. For example, P-RISC [8], Garp [9], XiRISC [10], and Molen [11]. A specific focus is set on the Molen programming model and its architectural organization. The Molen approach is presented as a meeting point between the software domain (sequential programming and compiler) and the hardware domain (specific instruction designed in hardware).

Figure 2 illustrates the schematic view of the whole processor, including the RFU.

The processor supports a restricted instructions set, that conforms to a SET-EXECUTE-STORE Molen paradigm [11]. In order to keep the project reasonably simple, we restrict the use of the RFU to implementing Data Flow Graphs (DFGs) on one hand, and we provide students with the Biniou framework on the other hand. Restricting the use of the reconfigurable part as a functional units also mitigates the complexity of the whole design. However, this covers the need for being reachable by average students while preserving the ability to arouse's top students curiosity, by offering a set of interesting perspectives for further developments.

This project let students build and stress new ideas in many disciplines related to reconfigurable computing such as spatial versus temporal execution, architectures, programming environments, and algorithms.

2.3.1. Context. This project takes place during the fall semester, from mid October to early January. A noticeable point is that almost no free slots within the timetable are dedicated to this project, that overlaps with courses as well as with "concurrent" projects. This intends to stress students and make them aware of handling competing priorities.

2.3.2. Expected Deliverables. We define three milestones and three deliverables. The milestones are practical sessions in front of the teacher.

Three main milestones are as follows.

M1: RISC processor, running its provided test programs.

M2: RFU, with Galois Field-based operations implemented as bitstream.

M3: Integration, final review.

2.3.3. Schedule. The schedule is provided during the project "kick-off". To prevent students from postponing managing this project we use the collaborative platform to monitor activities, to specify time-windows for uploading deliverables, and to broadcast updates/comments/additional information. Reminders can be sent by mail when the deadline is approaching. Once the deadline expires, over-due deliverables are applied a penalty per extra half-day.

Table 1: Instruction layout.

15	14	13	12	11	10	9	8	7	6	5	4	···	0
Opcode				MA				OP1				OP2	

3. Project

3.1. Processor Soft-Core. Designing such a simple processor carries no extra value and several teaching experiments are reported [12]. However, keeping in mind that half the students have never exercised writing VHDL description, and given practice makes success, we decided to let students design their own processor. Although, a preliminary version with missing control structures was provided in order to ensure a minimal compatibility through the designs. Obviously, the matter here was to ease evaluation from a scholar point of view as well as to force students to handle kind of legacy system and refactoring rather than full redesign.

We also provided the instruction set and opcode. In an ideal world, and with a more generous amount of time to spend on the project, as the design is highly modular, building a working design by picking best-fit modules out of several designs would have also been an interesting issue.

3.1.1. Decoder. It outputs signals from input instruction according to the layout on Table 1. This information is provided to ensure compatibility as well as programmability (as no compiler support is considered).

3.1.2. Test Bench Program. Students are familiar with agile programming, test-driven development and characterization tests. When designing a processor, the same approach applies but at a wider granularity (program execution instead of unit test). Hence, we distributed some test bench programs. Analyzing at specific timestamps (including after the application stops) the internal states (some signals plus registers contents) leads to design scoring.

3.2. Reconfigurable RFU Design

3.2.1. Background. In order to give to students the main architectural concepts behind FPGAs, we first focus on a simple mesh of basic processing elements composed of one 4 entries Look-Up Table (LUT) each. Combination of the basic blocks (LUT, switch, buses, and topology) is presented as a template to be extended (in terms of routing structure and processing elements) for building real FPGA. A more realistic example from the industry (a Xilinx Virtex-5) is considered with a highlight on template basic blocks in Xilinx schematics. As a result, students are able to locate the essential elements for a better understanding of state-of-the-art architectures. Drawbacks of fine-grained architectures such as low computation density and routing congestion are highlighted to introduce coarse-grained architectures. This type of reconfigurable architecture is firstly presented as a specialization of FPGA suited for DSP application domain.

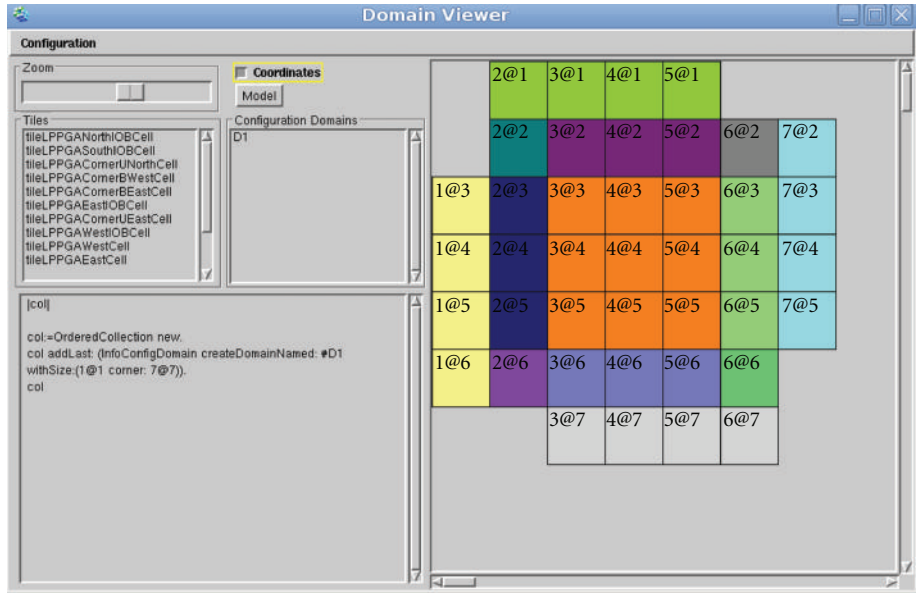


Figure 3: On the right, view of the different cell types composing the matrix (border cells, middle cells, IO cells). On the left, configuration domains are defined as a set of rectangular boxes. They can be reconfigured independently from each other.

Architectures presented are Kress-Array [13], Piperench [14], PACT XPP [15], and Morphosys [16]. Programming model issues are discussed with a comparison between software oriented approach (generally using subsets of C) and hardware approach (netlist based descriptions). A case study of the DREAM architecture is presented with an emphasis on the compiler friendly approach of the tools targeting the PiCoGA [17, 18].

3.2.2. Modeling. Before entering the generation phase, students learn to hand-design an FPGA. Every elements of a basic FPGA are detailed and a corresponding VHDL behavioral description is provided. The bottom-up description starts from atomic elements, such as pass gates, multiplexers, that are combined to form input/output blocks and configurable logic blocks. A daisy chain architecture is detailed as well as a configuration controller.

Then, the second part describes the Biniou generation of the architecture from an ADL description. An FPGA is described using an ADL increasing the level of abstraction compared to a VHDL description. The configuration plan is described as a set of domains to support partial reconfiguration. The approach relies on model transformation, with an automatic VHDL code generation from a high-level description.

3.2.3. RFU Structure. As a preliminary approach, students have to design an island style mesh architecture, what means sizing the matrix, defining a basic cell, and isolating border cells that deserve special attention. The basic cell is either used as is for the internal cells and tuned to generate the border cells because their structure is slightly different from the common template. Defining the domains appears as shown by Figure 3.

The basic cell schematic view is provided by Figure 4.

Ultimately, the full matrix appears as an array of N^2 cells as illustrated by the snapshot of the Biniou P&R layer (Figure 5).

3.3. Reconfigurable Functional Unit Integration. The reconfigurable functional unit (RFU) is composed of three main components: the reconfigurable matrix (RM) generated by Biniou, a configuration cache, and the RFU controller both hand-written (see bottom right in Figure 2).

Configuration is triggered by the processor controller which reacts to a SET instruction by sending a signal to the RFU controller. The RFU controller drives the configuration cache controller, which provides back a bitstream on demand.

The processor controller gets an acknowledgment after the configuration completed.

One critical issue about the processor-RFU coupling lies in data transfers to/from the RFU. Students have to design a simple adapter which connects a set of RFU's iopads to the processor registers holding input and output data (Op1, Op2, and Res in Figure 2).

Figure 6 gives a detailed view of the adapter.

3.4. Application Synthesis over the RFU. To let students figuring out the benefit of adding the RFU to the processor design, it is desirable that students can assess and compare the impact of several options. One classical approach lies in isolating a portion of the application to be further converted into an accelerated function. In this case, we

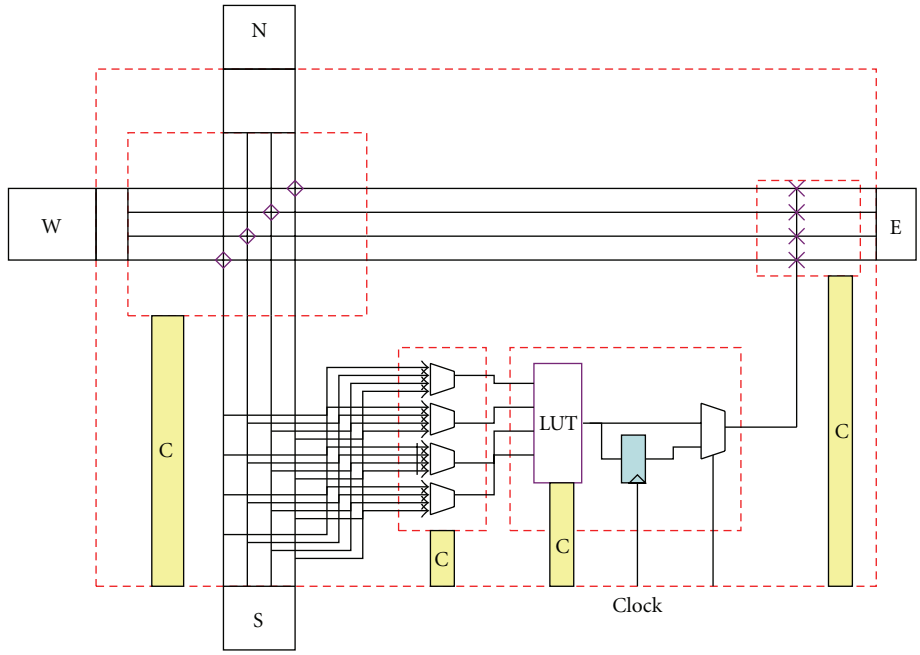


Figure 4: Structure of a basic cell (middle cell) within the RFU matrix.

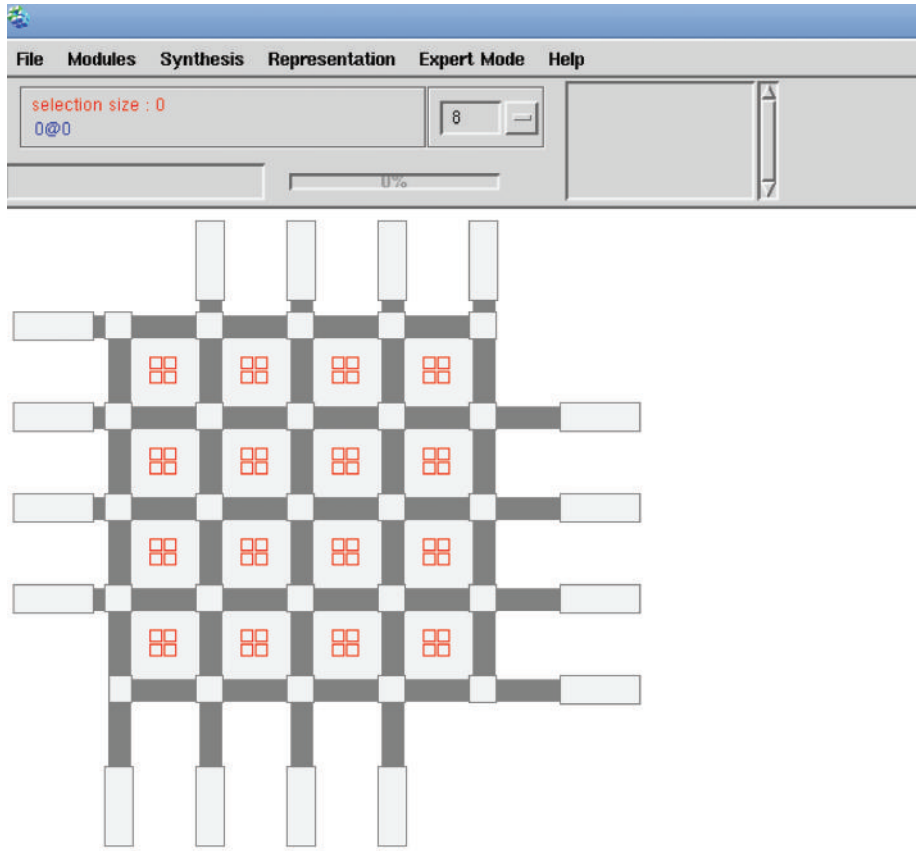


Figure 5: Whole view of the RFU.

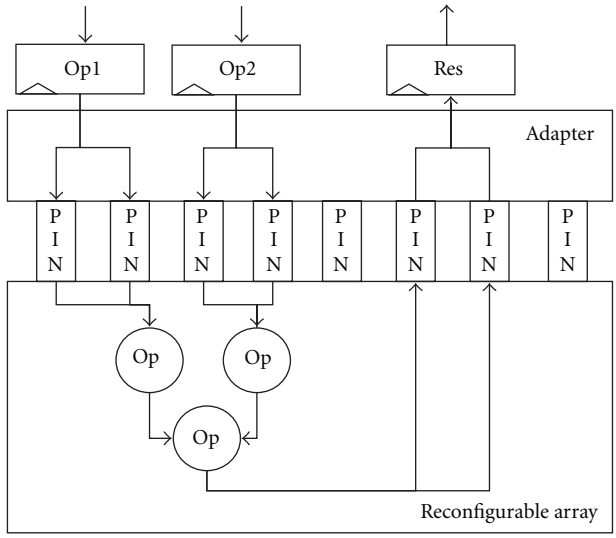


Figure 6: RFU is interfaced with the processor registers through an adapter.

implement a DFG to exhibit spatial execution. Another option consists in defining novel primitive operators. As an example, defining a multiplier instead of performing several processors instructions (addition, shifts, etc.) can make sense due to a high reuse rate.

In both cases, the RFU extends the instructions set.

Additionally, the underlying arithmetic can vary keeping the instructions set stable despite adding new variants for implementing these instructions. This goes through either a library-based design or dedicated synthesizers. Libraries are typically targeted to a reduced set of predefined macroblocks, and they are not easily customizable to new kinds of functions or use-cases.

We chose to focus on the second topics as this seems to carry extra added-value compared to classical flows, while reducing the need for a coding extra effort thanks to provided synthesis facility.

Figure 7 illustrates the Biniou behavioral application synthesizer. The optimizing context here is made up of typing as Galois Field GF16 values the two parameters. A so-called high-level truth table is computed per graph node for which values are encoded and binarized. The logic minimization [19] produces a context-dependent BLIF file.

This BLIF file is further processed by the Biniou P&R layer. As application is simple enough to keep the design flatten, no need exists for using a floorplanner. However, for modular designs, a TCG-based floorplanner [20] is integrated within Biniou.

Some constraints are considered, such as making some location immutable to conform to the pinout of the adapter (Figure 6) with regards to the ones assigned to the I/O of a placed and routed application (see Figure 8).

Once the P&R process ends, a bitstream is generated. Each element of the matrix both knows its state (used, free, which one out of N, etc.) and its layout structure. The full layout is gained by composing recursively (bottom up) these sub-bitstreams. An interesting point is that the bitstream structure can vary independently from the architecture by applying several generation schemes. As a result, in a partial reconfiguration scope, the students benefit from enriched architectural prospection capabilities. In the frame of the project an example of bitstream structure is provided by Figure 9.

3.5. Reports and Oral Defense. Students had to provide three reports, one per milestone. The reports conformed to a common template and ranged from 10 to 25 pages each. The last report embedded the previous ones so that the final document was made available straight after the project and students were given second opportunity to correct their mistakes.

Some recommendations were mandatory such as embedding all images as source format within the package, so that we could reuse some of them. As an illustration, more or less half of the figures in this papers come from students reports. The students had no constraints over the language but some of them chose to give back English-written reports. We selected some reports to be published on line as examples for next year students.

The last deliverable was made up of a report, working VHDL code and an oral defense. Students had to expose within 10 minutes, in front of the group, course teachers, and a colleague responsible for the "communication and job market" course.

Some students chose to center their defense around the project and the course versus project adequation, some others around the "product", that was their version of the processor.

3.6. Results Coming out of the Project. The simulation environment is ModelSim [21] as illustrated by Figure 10. The loader module—that loads up the program—was not provided but students could easily get one by simply reusing and adapting the generated test bench. Only one group out of five got it right.

This allowed to set a properly initialized state prior to execution's start. Of course, this was a critical issue, and students would have done well to fix it in an early stage as tracing values remained the one validation scheme. This was all the more important as the full simulation took a long time to complete and rerun had a real cost for students.

The simulation of the processor itself is time-affordable but the full simulation takes around 4 hours, including bitstream loading, and whole test bench program execution.

3.6.1. Optimizations. Students came to us with several policies to speed up the simulation. A first proposal is to let simulation happen at several abstraction levels, with a high rate of early error detection. Second, some modules have been substituted by a simpler version. As an example, by providing a RFU that only supports 8 bits ADD/SUB operations, the bitstream size is downscaled to 1 bit with no compromise on the architecture decomposition itself. This approach is very interesting as it confines changes to

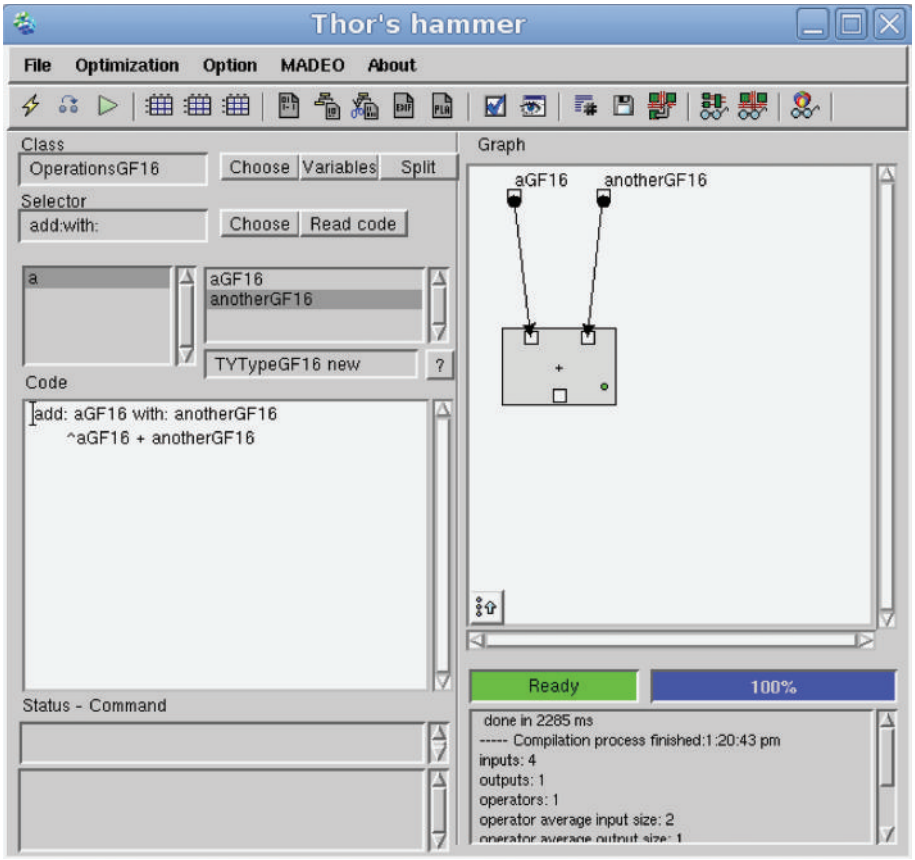


Figure 7: Specification of a GF16 adder.

the inside of the RFU while still preserving the application programming interface. In addition, it joins back the concern of grain increase in a general scope (i.e., balancing the computation/flexibility and reducing the bitstream size). Also this approach must be linked to the notion of "mock object" [22], software engineers are familiar with, when accelerating code testing.

Third, as the application is outputed as RTL code, the code can be used as a hard FU instead of using reconfigurable one. In this way, the students validated the GF-based synthesis. Grabbing these last two points, the global design can be validated very fast, being the scalability issue. This issue has been ignored during the project, but is addressed as the global design is given a physical implementation.

3.6.2. Analysis. The students sampling cannot be considered representative from a statistical point of view. However, some preliminary remarks seem to make sense.

Figure 11 shows that the deliverable 2 is harder to complete than the first one, but that more than half of the students got a success rate between 70% and 90%.

We chose to make students pair-achieve the project. In this way, beyond simply averaging the prerequisites matching so that the pairs are equally offered a chance to succeed, we intended to favor incidental learning as pointed out by chanck [23].

The increase of the standard deviation (Figure 12) highlights that one group failed in properly using the toolset (left border, Figure 11); another way to analyze this is that the toolset allowed to overcome the complexity of deliverable 2. Another interesting point is that the global understanding raises up during the full project, being the group who gave up after the first milestone (right border, Figure 11). The difference between regular and restricted lines is that restricted lines ignore this group. Finally, the standard deviation line points out that most homogeneous results came from integration, manual design of the processor, and last using the tool set.

4. Real Case Study

4.1. Experimentation Platform. The physical implementation was out of the scope of this project mainly due to some timetable hard constraints. Not all of the students proceeded in implementing their circuits. But the lessons we have learned are really inlined with the feedback we got from those of our students who applied for an internship in another lab.

The development platform we use for this demonstrator is a Virtex-5 FXT ML510 Embedded Development Platform from Xilinx.

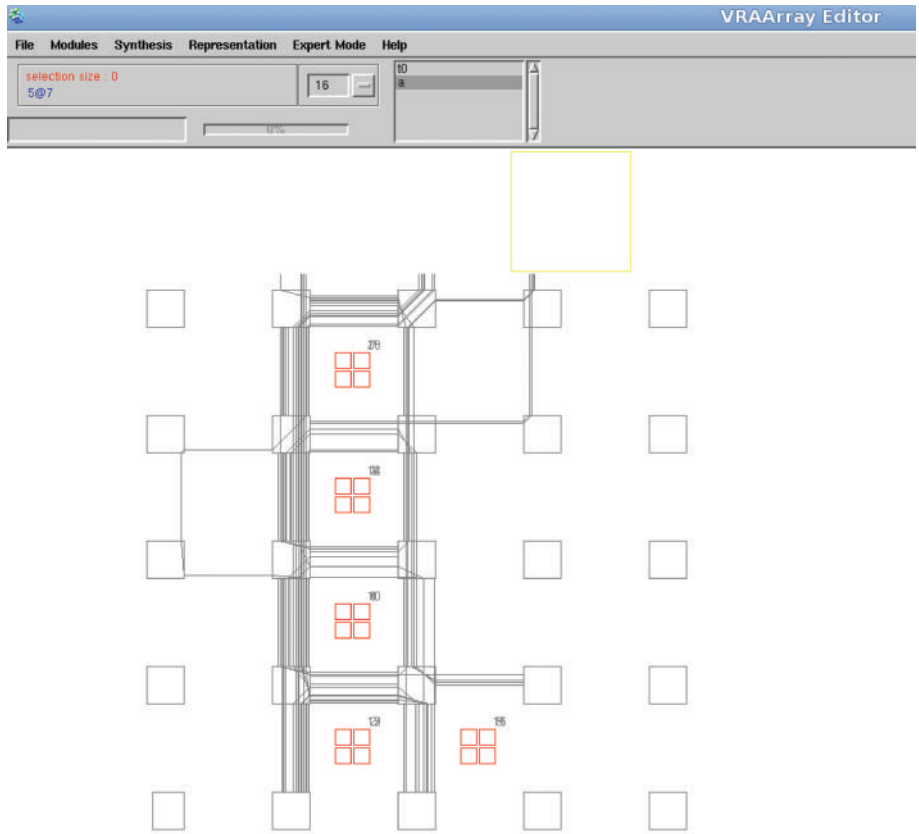


Figure 8: An application placed and routed over the RFU.

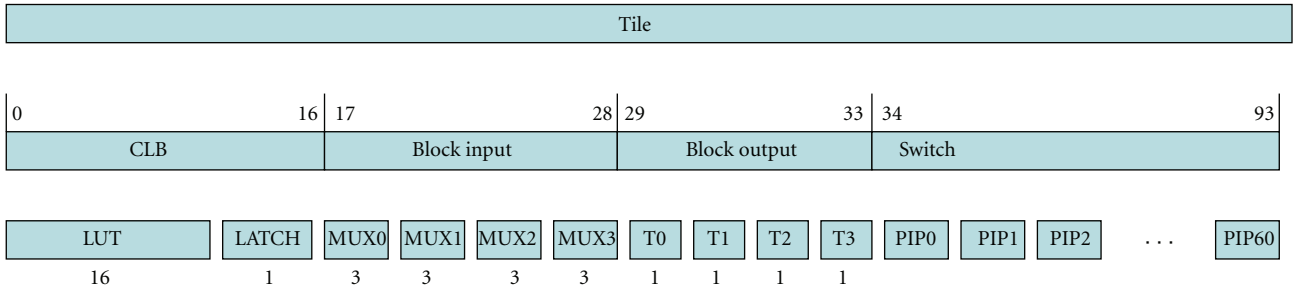


Figure 9: Example of a bitstream hierarchical organization.

4.2. Processor. A first noticeable difference between their former experience and the real case implementation lies in abandoning their hand written processor. Instead, the students had to instantiate a soft core.

4.2.1. Soft-Core. The soft-core processor is a Micro-Blaze and comes along with a full software environment.

4.2.2. Programmability. Not only, using this soft-core ensures a knowledge of state-of-the-art techniques but also it eases porting application. On the other hand, mixing soft and hard components within a single application is pretty clear to students who extended by hand the ISA of the toy processor.

4.2.3. Simulation. Another interesting features is the observability the simulation environment provides. On the opposite, gaining visibility during ModelSim simulation required to group/color/rename signals in the first processor. This is also important for performances extraction as scanning a *done* signal was used for time measurement.

4.3. Accelerator. The first version of the accelerator was a fine grained mesh. However, these architectures suffer from a long synthesis process, hence some coarser-grained architectures have been proposed in the literature to overcome this limitation. The second version reflects this architectural shift by exhibiting coarse-grained elements.

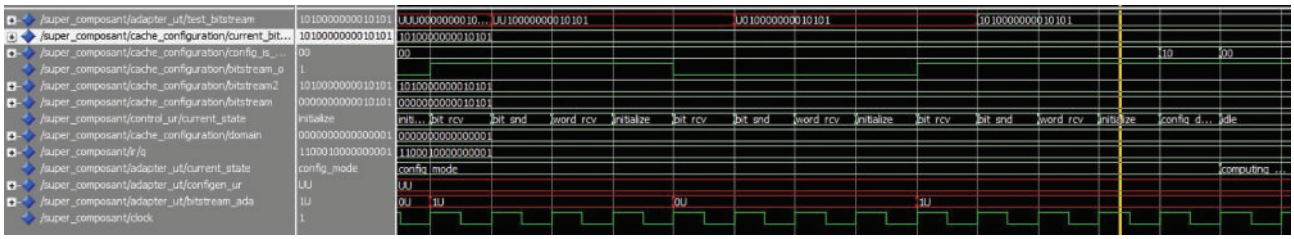


Figure 10: Modelsim simulation.

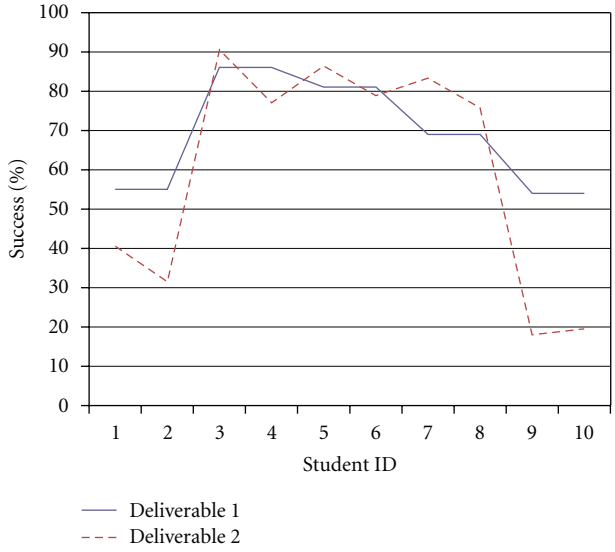


Figure 11: Results (% success) from milestones 1 and 2.

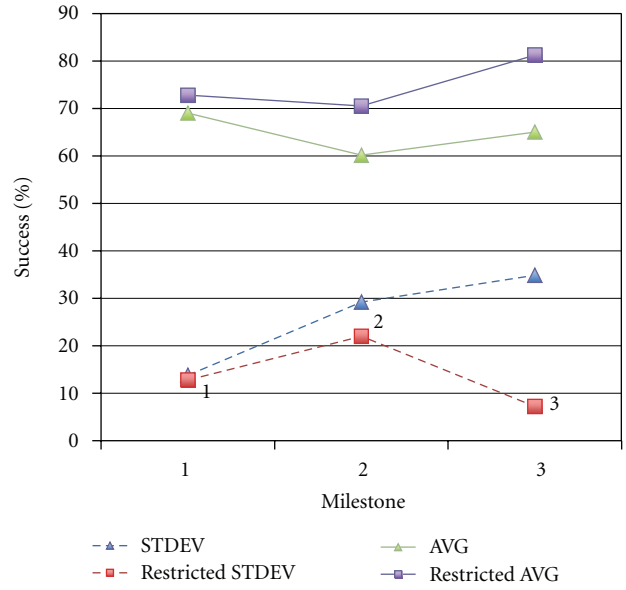


Figure 12: Global results.

4.3.1. Grain Considerations. In addition to these general considerations, students faced a performance issue when implementing fine-grained mesh over an FPGA. First, the synthesizer exhibited very low frequency. Secondly, the placement efficiency was unsurprisingly very poor.

At this point, another option emerged. A coarser grained architecture, inspired of PicoGA [24] but not as complex, was considered. The new architecture is organized as pipelined stripes. Logic elements are ALUs.

4.3.2. Impact over the Software Environment. The Biniou P&R relies on a Pathfinder [25] algorithm. The students got wrong configuration until we provided them a refactored version of the placer, that conforms to the stripe-based organization.

4.4. Processor-Accelerator Pairing. The third move between the project and the real case lies in changing the way the processor and the accelerator are connected to each other. The processor must support non blocking accelerated function calls which prohibits the former coupling scheme.

4.4.1. Coupling. Instead we asked the students to isolate the accelerator as an autonomous entity (coprocessor). The implementation was realized using FSLs, which is a classical option. Combining network concerns (FIFO, hand-shake, negotiation, etc.) with the simple adapter (Figure 6) made FSL a very natural concept to computer engineers.

4.4.2. Timing Constraints. Gaining high performances requires to force constraints when calling the ISE synthesizer.

4.4.3. Layout. Figure 14 illustrates a layout of a coarse-grained reconfigurable architecture (see Figure 13) acting as an accelerator for a Micro-Blaze.

4.5. Manual Domain Space Exploration. Once acquired a sound knowledge of the domain (architecture, platform, tools), students started to address Domain Space Exploration (DSE). First, this stage was kept manual still following the precept of "simplicity" and "just-fit approach".

4.5.1. Considered Cases. The first dimension for variability is the matrix sizing. Several instances have been designed ($5 * 2$, $5 * 4$, $5 * 10$, $40 * 40$). The second axis is the reconfiguration grain. For a similar matrix, several instances are issued with a different partial reconfiguration page size

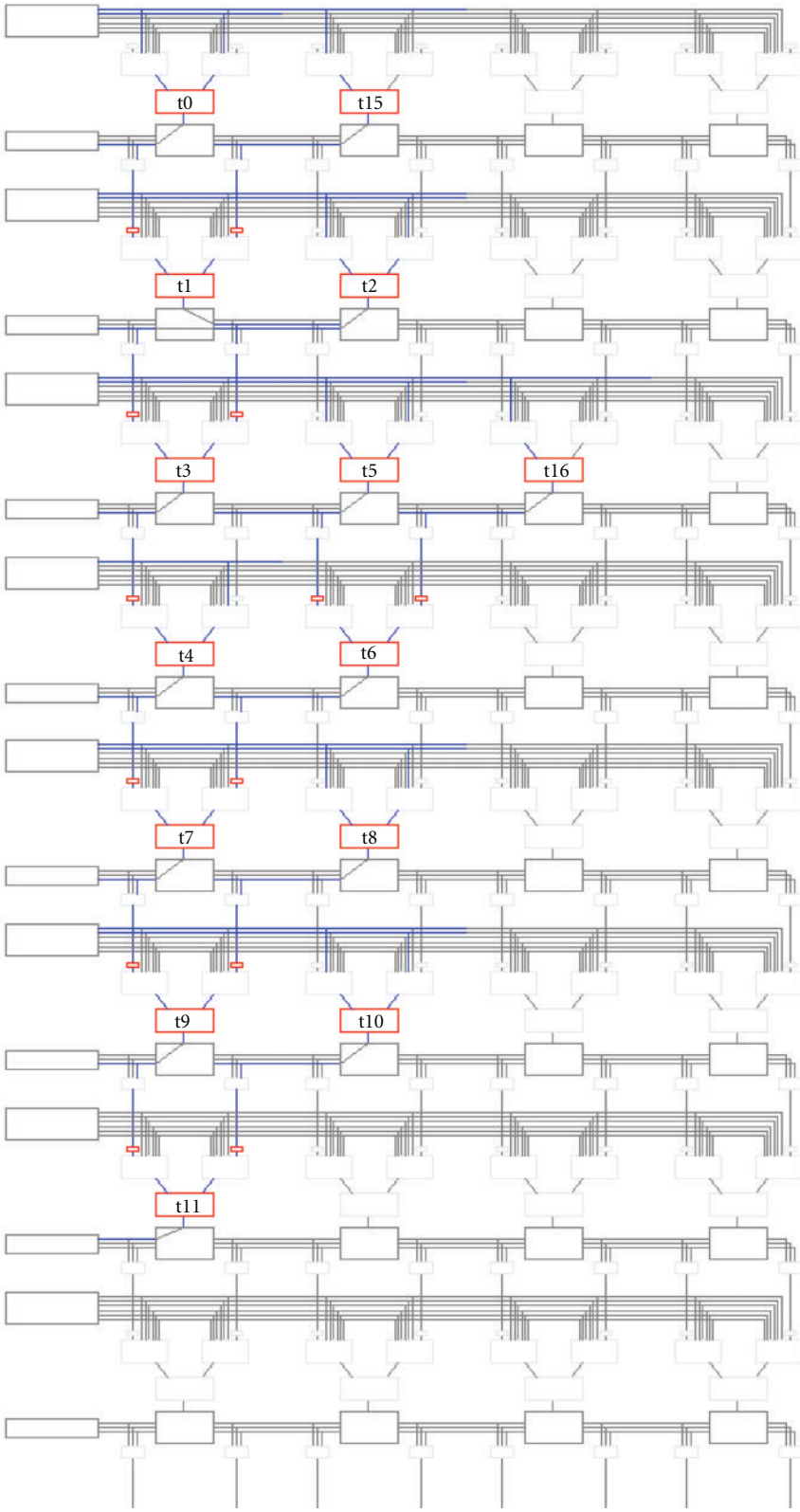


Figure 13: The view Biniou provides over a Coarse Grained Reconfigurable Architecture under use.

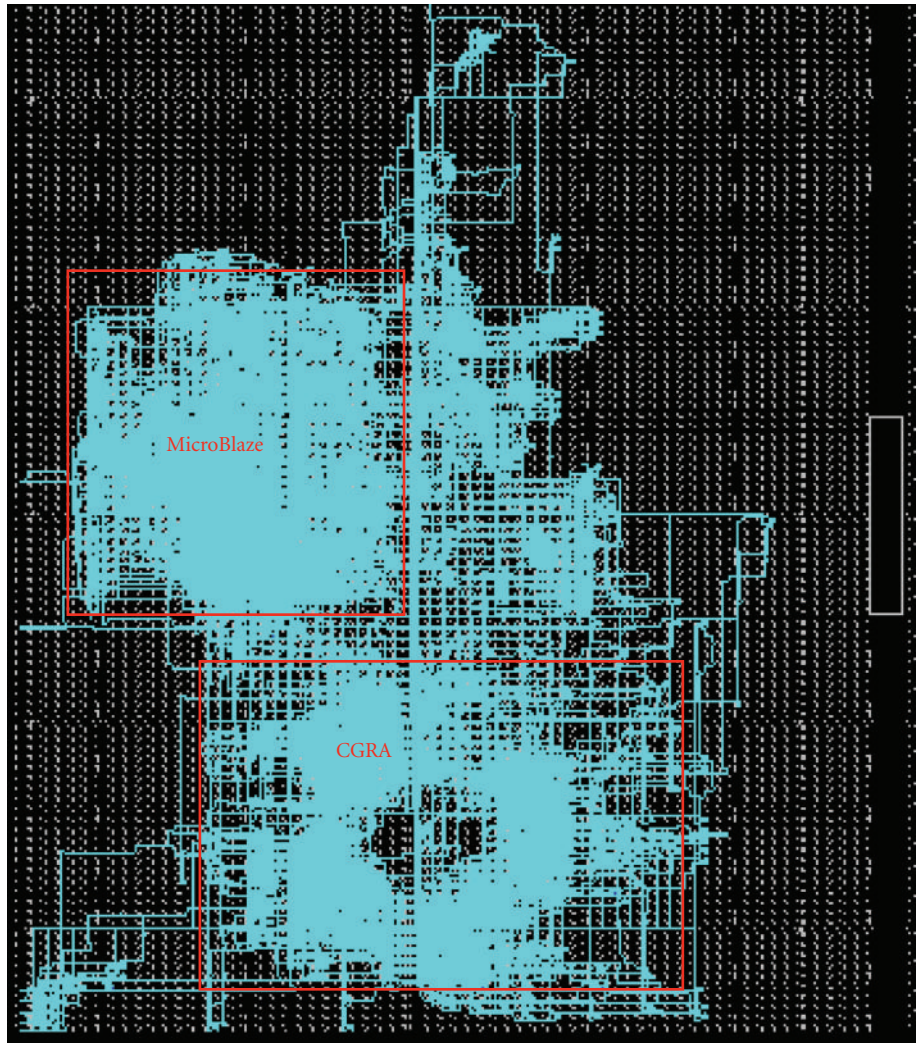


Figure 14: A resulting layout with a MicroBlaze connected to a coarse grained reconfigurable architecture through FSL.

each. Then, the last measured impact is related to the number of configuration contexts.

4.5.2. Metrics. It is important to measure the quality of solutions, especially the specific amount of a certain resource and architectural solution needs. Examples of such resources would be area, time, or memory storage.

4.5.3. Speed up Measurements. Computing a speed up requires two things: first, measuring an execution time, then comparing versus a reference execution time. A nonobvious point to students is how to make a fair measurement. As an example, the coarse grained architecture may affect the processor's frequency. Hence, two speed-up must be analyzed. The first one makes use of a pure software execution time whereas the second one considers the execution time of a full software variant running on a processor/coprocessor architecture.

Of course, this speed-up remains highly application dependent. A FIR execution has been considered as this was enough for teaching purposes; as an example, the speed up factor for an FIR with 8 coefficients and 6500 data hits 31.6.

4.6. Towards an Automatic DSE. Creating spike solutions helps to figure out answers to tough technical or design problems. A spike solution is a very simple program to explore potential solutions. Students are encouraged to design spike solutions to stress some hypothesis before any announcement. The spike must be built only to address the problem under examination and ignore all other concerns. The goal is to reduce the risk of a technical problem or to increase the reliability of their feelings and estimate.

Spike solutions are applied for grabbing synthesis information and scripting the design tool suite.

4.6.1. Synthesis Report Analysis. The synthesis reports provide a set of information for quality measurement. The first metric is the amount of used resources. This appears as used Luts/FlipFlops pairs, plus internal fragmentation. The students have no control over the algorithms, and some

Table 2: Sizing matrix impact over frequency, resources, and synthesis time.

Dimensions	5×2	5×4	5×10	40×40
Freq.	102.8	102.8	101.7	53.8
# Slices	492	969	2397	4999
Cpu	35	45	100	47355

Table 3: Multiple context impact over frequency, resources, and synthesis time.

dimensions	# Contexts	Freq.	# Slices	Cpu
5×4	1	102.8	969	45
	2	99.8	1105	55
	5	99.8	1423	83
	10	98.0	2184	128
5×10	1	101.7	2397	100
	2	99.7	2747	137
	5	100.2	3522	238
	10	99.2	5415	468

results are difficult to analyze. As an example, in Figure 12, the depopulated center of the coprocessor may reflect the torus nature of the coarse-grained architecture. Nevertheless, stressing the constraints change the topology at the expense of a frequency scaling down.

Frequency is the second metric that the students concentrated on, all the more so as violations can occur which invalidate the full design.

The students knew how to find the relevant information. Going further though would have required to write a parser, then to extract scoring out of generated reports. This would be an interesting step forward command/scoring the tool suite.

4.6.2. Xilinx SDK Scripting. In order to detect the system files that are involved in a potential scripting, a first design is done through the user interface. Then, all modified files are reported, and a `diff` command is issued to let the students precisely locate internal changes. Then, code generation happens and recompiling the projet results in refactoring the design.

4.6.3. DSE Results. Tables 2 and 3 summarize for illustration purposes some of the DSE results the students collected.

5. Conclusion

This paper presents an experience report of course setup for master students discovering configware. This course tends to overcome the information pick-up limit to offer a real knowledge to students. This goes through manual design of toy examples that forces students to emphasize simple designs. Once acquired such an insight, commercial design suite are introduced for up-to-date training. Beside, research tools support complex tasks such as reconfigurable platform design, and DSE in a general way.

5.1. Forces. One interesting point regarding this project lies in the change in the students feeling. When we presented at the first time the project, they thought they would never complete the goals. After the first milestone, one group gave up to avoid paying the over due penalty and bounded their work to the first deliverable. They finally reached 7 points out of 20. The other groups faced the challenge and discovered that the key issue lies in getting proper tools to free oneself from manually developing both architectures and application mapping. The final results were very likely acceptable and we collected several working packages.

With this experience in mind, students are now ready for entering a very competitive job market. They share a deep understanding of both hardware design over reconfigurable architecture, microprocessors, reconfigurable cross integration, and tools and algorithms development.

This effect has been clearly pointed out when migrating from a toy example to real design environment. This move has offered several dimensions for DSE: reconfigurable unit grain, processor, coupling, and so forth.

5.2. A Very Positive Feedback. The actual success of this teaching experience lies in the highly efficient learning curve we noticed when students started to experience Xilinx design Kit. Obviously, neither the test bench examples we first provided nor the students population size are sufficient to practice real metrics-based measurements. Exploring the benefits of this approach (e.g., measuring speed-up) requires an easy path from a structured programming language such as C to the processor execution. Hence, the application's change would carry no need for hand-written adjustments. From our point of view, such an add-on in the project would be a fruitful upgrade to the course, and would spawn new opportunities for cross H/S expertise; keeping in mind that the reconfigurable computing course intends to get out with highly trained students sharing skills in both area.

Developing a small compiler was out of the scope of this project due to some timing constraints, but remains one hot spot to be further addressed. This could benefit from some Biniou facilities such as the C-entry synthesizer.

An open option is then to benefit from another course and invited keynoters to fulfill the prerequisites so that adapting/developing simple C parser becomes feasible in the scope of our project, at the cost of around an extra week.

5.3. Going Further. The second very positive feedback we got is that students are ready for new experiences, even with research tools that do not offer the same QoS than commercial design suite. This offered a path to reconfigurable units design with a full high level synthesis support.

Now, an interesting option is to introduce more efficient RFU, by generating coarse-grained architectures that support virtualization. Applying virtualization techniques allows to leverage some well-known limitations of reconfigurable architectures: limited amount of resources, lack of high-level programming model, and nonportability of bitstream.

Biniou offers a smart framework for design-space exploration of reconfigurable IPs. Fine-grained architectures offer

a nice teaching testbed, but shifting from fine to coarse-grained architecture rather make sense for current technologies. This brings no extra cost as Biniou fully supports this architectural scheme. Instead, this carries extra value as it underlines the resulting shift from "hardware" netlist design to "software" operation graphs editing.

Ensuring students will get the appropriate strength to self-adapt to such changing environment remains our educational goal. Once done, hard-soft co-design and applicative needs adequation driven platform development are on their way.

References

[1] J. M. Filloque, E. Gautrin, and B. Pottier, "Efficient global computations on a processor network with programmable logic," in *Parallel Architectures and Languages Europe (PARLE '91)*, pp. 69–82, 1991.

[2] L. Lagadec and B. Pottier, "Object oriented meta tools for reconfigurable architectures," in *Reconfigurable Technology: FPGAs for Computing and Applications II*, Proceedings of SPIE, pp. 69–79, November 2000.

[3] L. Lagadec and D. Picard, "Software-like debugging methodology for reconfigurable platforms," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, May 2009.

[4] S. R. Alpert, K. Brown, and B. Woolf, *The Design Patterns Smalltalk Companion*, Addison-Wesley, Boston, Mass, USA, 1998.

[5] E. M. Sentovich et al., "Sis: a system for sequential cirquit synthesis," Tech. Rep. UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, Berkeley, Calif, USA, May 1992.

[6] "Madeo-web, the madeo+ web version," http://stiff.univ-brest.fr/MADEO-WEB/.

[7] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, pp. 11–18, 1993.

[8] R. Razdan, K. S. Brace, and M. D. Smith, "PRISC software acceleration techniques," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 145–149, October 1994.

[9] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "Garp architecture and C compiler," *Computer*, vol. 33, no. 4, pp. 62–69, 2000.

[10] F. Campi, R. Canegallo, and R. Guerrieri, "Ip-reusable 32-bit vliw risc core," in *Proceedings of the 27th European Solid-State Circuits Conference*, pp. 445–448, 2001.

[11] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. L. M. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[12] V. Angelov and V. Lindenstruth, "The educational processor Sweet-16," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 555–559, August 2009.

[13] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Using the KressArray for reconfigurable computing," in *Configuralble Computing: Technology and Applications*, Proceedings of SPIE, pp. 150–161, November 1998.

[14] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Matt, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[15] J. Becker and M. Vorbach, "Coarse-grain reconfigurable XPP devices for adaptive high-end mobile video-processing," in *Proceedings of IEEE International SOC Conference*, pp. 165–166, September 2004.

[16] G. Lu, M. hau Lee, H. Singh, N. Bagherzadeh, F. J. Kurdahi, and E. M. Filho, "Morphosys: a reconfigurable processor targeted to high performance image application," in *Proceedings of the International Symposium on Parallel and Distributed Processing*, pp. 661–669, 1999.

[17] F. Campi, A. Deledda, M. Pizzotti et al., "A dynamically adaptive DSP for heterogeneous reconfigurable platforms," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 9–14, April 2007.

[18] C. Mucci, C. Chiesa, A. Lodi, M. Toma, and F. Campi, "A C-based algorithm development flow for a reconfigurable processor architecture," in *Proceedings of the International Symposium on System-on-Chip (SoC '03)*, pp. 69–73, November 2003.

[19] L. Lagadec, B. Pottier, and O. Villellas-Guillen, "A lut-based high level synthesis framework for reconfigurable architectures," in *Domain- Specific Processors : Systems, Architectures, Modeling, and Simulation*, S. Batttacharyya, E. Deprettere, and J. Teich, Eds., pp. 19–39, Marcel Dekker, 2003.

[20] J. M. Lin and Y. W. Chang, "TCG: a transitive closure graph-based representation for non-slicing floorplans," in *Proceedings of the 38th Design Automation Conference*, pp. 764–769, June 2001.

[21] "Modelsim," http://www.model.com/.

[22] D. Picard and L. Lagadec, "Multilevel simulation of heterogeneous reconfigurable platforms," *International Journal of Reconfigurable Computing*, vol. 2009, Article ID 162416, 12 pages, 2009.

[23] R. Schank, Tech. Rep., Institute for the Learning Sciences (ILS), Northwestern University.

[24] A. Lodi, M. Toma, and F. Campi, "A pipelined configurable gate array for embedded processors," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '03)*, pp. 21–29, February 2003.

[25] L. McMurchie and C. Ebeling, "PathFinder: a negotiation-based performance-driven router for FPGAs," in *Proceedings of the 3rd ACM International Symposium on Field-Programmable Gate Arrays*, pp. 111–117, February 1995.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 985931, 17 pages
doi:10.1155/2011/985931

Research Article

Exploration of the Power-Performance Tradeoff through Parameterization of FPGA-Based Multiprocessor Systems

Diana Göhringer,[1] Jonathan Obie,[1] André L. S. Braga,[2] Michael Hübner,[3] Carlos H. Llanos,[2] and Jürgen Becker[3]

[1] *Object Recognition Department, Fraunhofer IOSB, 76275 Ettlingen, Germany*
[2] *Department of Mechanical Engineering, University of Brasilia (UnB), 70910-900 Brasilia, DF, Brazil*
[3] *ITIV, Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany*

Correspondence should be addressed to Diana Göhringer, diana.goehringer@iosb.fraunhofer.de

Received 6 September 2010; Accepted 10 February 2011

Academic Editor: Gilles Sassatelli

The design space of FPGA-based processor systems is huge, because many parameters can be modified at design- and runtime to achieve an efficient system solution in terms of performance, power and energy consumption. Such parameters are, for example, the number of processors and their configurations, the clock frequencies at design time, the use of dynamic frequency scaling at runtime, the application task distribution, and the FPGA type and size. The major contribution of this paper is the exploration of all these parameters and their impact on performance, power dissipation, and energy consumption for four different application scenarios. The goal is to introduce a first approach for a developer's guideline, supporting the choice of an optimized and specific system parameterization for a target application on FPGA-based multiprocessor systems-on-chip. The FPGAs used for these explorations were Xilinx Virtex-4 and Xilinx Virtex-5. The performance results were measured on the FPGA while the power consumption was estimated using the Xilinx XPower Analyzer tool. Finally, a novel runtime adaptive multiprocessor architecture for dynamic clock frequency scaling is introduced and used for the performance, power and energy consumption evaluations.

1. Introduction

Parameterizable function blocks used in FPGA-based system development, open a huge design space, which can only hardly be managed by the user. Examples for this are arithmetic blocks like divider, adder, and soft IP-multiplier, which are adjustable in terms of bit width and parallelism. Additional to arithmetic blocks, soft-IP processor cores provide a variety of parameters, which can be adapted to the requirements of the application to be realized with the system. Especially, Xilinx offers, with the MicroBlaze Soft-IP 32-bit RISC processor [1], a variety of options for characterizing the core individually. These options are, amongst others, the use and size of cache memory, the arithmetic unit, a memory management unit, and the number of pipeline stages. Furthermore, the tools offer to deploy semiautomatically up to two processor cores as multiprocessor on one FPGA. Certainly more cores are available for the system design by performing the custom tool chain. Every option as described above can be adjusted to find an optimized parameterization of the single processor core in relation to the target application. For example, a specific cache size can speed up the application tremendously, but also the optimal partition of functions onto the two cores has a strong impact on the speed and power consumption of the system. These examples show the huge design space, even if only one parameter is used. It is obvious that the deployment of multiple parameters for system adjustment leads to a multidimensional optimization problem, which is not, or at least very hardly, manageable by the designer. In order to gain experience regarding the impact of processor parameterization in relation to a specific application scenario, it is beneficial to evaluate, for example, the performance and power consumption of an FPGA-based system and compare the results to a standard design with a default set of parameter. The result of such an investigation is a first step for developing standard guidelines for designers

and an approach for an abstraction of the design space in FPGA-based system design. This paper presents first results of a parameterizable multiprocessor system on a Xilinx Virtex-4 FPGA, where the parameterization of the processor is evaluated in respect to power consumption and performance. Moreover, the varying partition of the different application scenarios is evaluated in terms of power consumption for a fixed performance. For this purpose, a tool flow for analyzing the power consumption through generating the switching activity interchange format (SAIF) file or the value change dump (VCD) file from the postplace and route simulation will be introduced. The presented flow enables to generate the most accurate power and energy consumption estimation from this level of abstraction. A further output of the presented work is an overview of the impact of parameterization to the performance, power and energy consumption for Xilinx Virtex-4 and Virtex-5 FPGAs. The results can be used as a basic guideline for designers who want to optimize their system performance and the power and/or energy consumption. This basic guideline can be a starting point for the analysis of wider application scenarios that can produce a more complete guideline for the definition of task escalation strategies and for the parameterization of FPGA-based MPSoCs. The paper is organized as follows. In Section 2, related work is presented. Section 3 describes the power estimation tool flow used for the presented approach. The novel system architecture deployed for analyzing the performance and the power consumption of the different applications is presented in Section 4. The application scenarios are described in Section 5. In Section 6, the application integration and the results of the performance and power consumption evaluation are provided. Finally, the paper is closed by presenting the conclusions and future work in Section 7.

2. Related Work

Reduction of the dynamic and static power consumption is very important especially for embedded systems, because they often use batteries as a power source.

Therefore, many researchers, for example, Meintanis and Papaefstathiou [2], explored the power consumption of Xilinx Virtex-II Pro, Xilinx Spartan-3, and Altera Cyclone-II FPGAs. They estimated the power consumption at design time using the commercial tools provided by Xilinx and Altera. They further explored the differences between the measured and estimated power consumption for these FPGAs. Becker et al. [3] explored the difference between measured and estimated power consumption for the Xilinx Virtex-2000E FPGA. Furthermore, they explored the behavior of the power consumption, when using dynamic reconfiguration to exchange the FPGA-system at runtime.

Other works focus on the development of own tools and models for efficient power estimation at design time for FPGA-based systems. Poon et al. [4] present a power model to estimate the dynamic, short circuit, and leakage power of island-style FPGA architectures. This power model has been integrated into the VPR CAD flow. It uses the transition density signal model [5] to determine signal activities within the FPGA. Weiss et al. [6] present an approach for design time power estimation for the Xilinx Virtex FPGA. This estimation method works well for control-flow-oriented applications but not so well for combinatorial logic. Degalahal and Tuan [7] present a methodology to estimate dynamic power consumption for FPGA-based system. They applied this methodology to explore the power consumption of the Xilinx Spartan-3 device and to compare the estimated results with the measured power consumption.

All these approaches focus either on the proposal of a new estimation model or tool for estimating the power consumption at design time, or they compare their own or commercial estimation models and tools with the real measured power consumption. The focus of the investigations presented in this paper is to show the impact of parameterization of IP cores, specifically the MicroBlaze soft processor, which differs from the approaches mentioned above where the topic is more on tool development for power estimation.

The novelty of our approach is to focus on the requirements of the target application and to propose a design guideline for system developers of processor-based FPGA systems. This means providing guidance in how to design a system to achieve a good tradeoff between performance and power and energy consumption for a target application. To develop such a guideline, the impact of the frequency, different processor configurations, and the task distribution in a processor-based design are investigated in this paper for different application scenarios.

3. Tool Flow for Power Measurement

Xilinx provides two types of tools for power consumption estimation: Xilinx Power Estimator (XPE) [8] and Xilinx Power Analyzer (XPower) [9].

The XPE tool is based on an excel spreadsheet. It receives information about the number and types of utilized resources via the report generated by the mapping process (MAP) of the Xilinx tool flow. Alternatively, the user can manually set the values for the number and type of used resources. The frequencies performed within the design have to be manually set by the user. The advantage of this method is that results are obtained very fast. The disadvantage is that the results are not very accurate, especially for the dynamic power consumption. This is because the different toggling rates of the signals are not taken into account. A further reason the results are not too accurate, is that they are based on the MAP report, and not on the postplace and route (PAR) report, which resembles the system used for generating the bitstream.

The XPower tool, the alternative to Xilinx Power Estimator, estimates the dynamic and static power consumption for submodules, different subcategories, and the whole system based on the results of a postplace and route (PAR) simulation. This makes the estimation results much more accurate compared to the XPE tool, because the physical place and route, as well as the utilized resources of the system, are taken into account for the power estimation. But even more important, due to the simulation of the PAR system with real input data, the toggling rates of the signals can

be extracted and used within the power estimation. For estimating the power consumption with the XPower tool, the following input files are required:

(i) native circuit description (NCD) file, which specifies the design resources,

(ii) physical constraint file (PCF), which specifies the design constraints,

(iii) switching activity interchange format (SAIF) or value change dump (VCD) file, which specify the simulated activity rates of the signals.

The NCD and the PCF files are obtained after the PAR phase of the Xilinx implementation tool flow. Both files provide the information needed for an accurate estimation of the total quiescent power of the device. To achieve a good estimation of the dynamic power consumption, an SAIF or VCD file is required, because they contain information about the toggling rates of all signals within the hardware realization on FPGA. Both files can be obtained after simulation of the PAR design with real input data using, for example, the ModelSim simulator. The XPower tool can also be used without an SAIF or VCD file. Then, default toggle rates are used to estimate the dynamic power of the design, which is not as accurate as using an SAIF or VCD file.

By choosing the specific tool, the user has to decide on a tradeoff between accuracy and design time. For a high accuracy, a longer design time is needed, by using XPower with the results from PAR and PAR simulation. If a rough estimation is sufficient, both the XPE tool and XPower, using only the PAR results, but no SAIF or VCD file, can be used.

Due to the higher accuracy, the XPower tool using PAR and PAR simulation results was used for the approach presented in this paper. As we wanted to estimate the power consumption for systems with one or two MicroBlaze processors, the hardware and the software executables of the different system were designed within the Xilinx Platform Studio (XPS) [10]. Figure 1 shows the flow diagram for doing power estimation with XPower for an XPS system.

After the hardware has been designed and implemented within the XPS environment, the SimGen [10] tool is used to generate the post-PAR timing simulation model of the system. This simulation model is used to do a post-PAR timing simulation of the design. Within this work, we used the newest Xilinx tool version 12.2. Here, two possible simulators exist: Xilinx ISIM and ModelSim simulator. Both can be used without any restrictions to generate the SAIF or VCD files. In this work, ISIM was used to generate the SAIF files for all the uniprocessor designs. ModelSim was used to generate the VCD files for the two processors design. In the last step, XPower is required to load the SAIF/VCD, the NCD, and the PCF files of the design and to estimate the dynamic and static power consumption. Care has to be taken, because, in a normal Xilinx implementation flow, the software executables are integrated into the memories of the processors after the bitstream has been generated. When using XPower and the post-PAR simulation, the memories of the processor have to be initialized in an earlier step. This means, into the post-PAR simulation model, otherwise the simulated system behavior and the VCD file would not be accurate. Using the Xilinx 12.2 tools, this memory initialization is done automatically by the tools, when the target devices are a Virtex-4 and Virtex-5 FPGA.

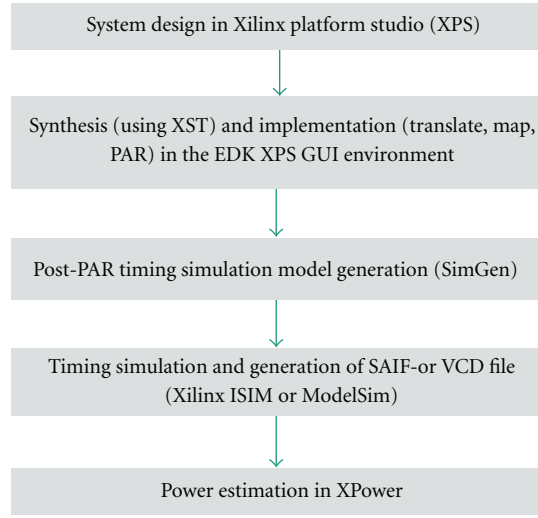


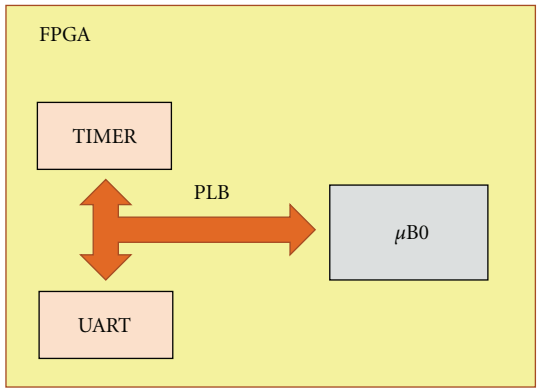
Figure 1: Diagram of the EDK XPower Flow.

4. Novel System Architecture

For evaluating the impact of the different design parameters onto the overall system performance, power, and energy consumption, three different processor designs have been built.

The first design is a uniprocessor, which is used to evaluate the impact of the clock frequency and the processor configurations onto the performance, power and energy consumption for two different Xilinx FPGAs: Virtex-4 and Virtex-5. Therefore, this system has been designed as shown in Figure 2. The number of I/O interfaces has been chosen in such a way that the same system can be used without changes for both FPGA boards. The only required changes are the physical I/O pin locations for the UART, clock, and reset pins. The system components remain unchanged, to allow a fair comparison between the two FPGAs. The input data for the applications is statically stored in the local on-chip memory of the processor, because an equal interface, for example PCI, was not present for both boards.

The system structure of the dual-processor system is shown in Figure 3. This system is used to evaluate the impact of different application partitions onto the overall performance, power dissipation, and energy consumption. This evaluation has been done on the Virtex-4 board only, because a PCI connection was needed to receive the input data for the applications from the host PC. Also, the differences between the power and energy consumption of the different FPGA families have been already done with the uniprocessor design, and, therefore, no differences are expected for the dual-processor design. Three new components have been designed and implemented: the virtual-IO, the bridge, and the reconfigurable clock unit. All three components have



PLB: processor local bus
UART: universal asynchronous receiver transmitter
μB: microblaze

Figure 2: Uniprocessor system.

been integrated into a library for the XPS tool. Therefore, they can be inserted and parameterized using the graphical user interface (GUI) of the XPS tool, which makes them easily reusable within other XPS designs.

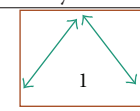
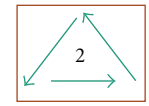
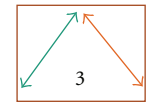
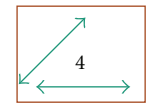
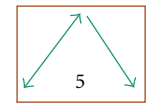
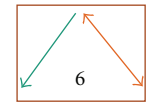
The virtual-IO receives data from the host PC and sends the results back to the host PC via the PCI bus. The virtual-IO communicates via the fast simplex links (FSLs) [11] with two MicroBlaze processors (μB0 and μB1). μB0 communicates with the user via the UART interface. It has a timer, which is used to measure the performance of the overall system. The two processors communicate with each other via FSLs over the bridge component. Depending on the fill level of the FIFOs within the bridge, reconfiguration signals are sent to the reconfigurable clock unit. The reconfigurable clock unit reconfigures the clocks of the two processors based on the reconfiguration signals issued by the bridge. The power and energy consumption of this dual-processor system is compared against the evaluation results of the uniprocessor system shown in Figure 4. This uniprocessor system uses also the virtual-IO component to receive the input data required for the applications. The bridge and the reconfigurable clock unit have been removed.

The following subsections explain the new components and their features more in detail.

4.1. Virtual-IO. The virtual-IO component [12] was designed to communicate with the host PC via the PCI bus. It provides an input and an output port to the PCI bus and one input and one output port for each MicroBlaze processor. It consists of two FIFOs, one for the incoming and one for the outgoing data of the PCI bus. Each FIFO is controlled via a finite state machine (FSM), as it is shown in Figure 5.

The virtual-IO is a wrapper around 6 different modules. The modules, their symbols, and functionalities are summarized in Table 1. The first module is virtual-IO 1, which sends data first to μB0 and then to μB1. It then receives the calculated results in the same order. The second module is virtual-IO 2, which sends data only to μB0. Results are only

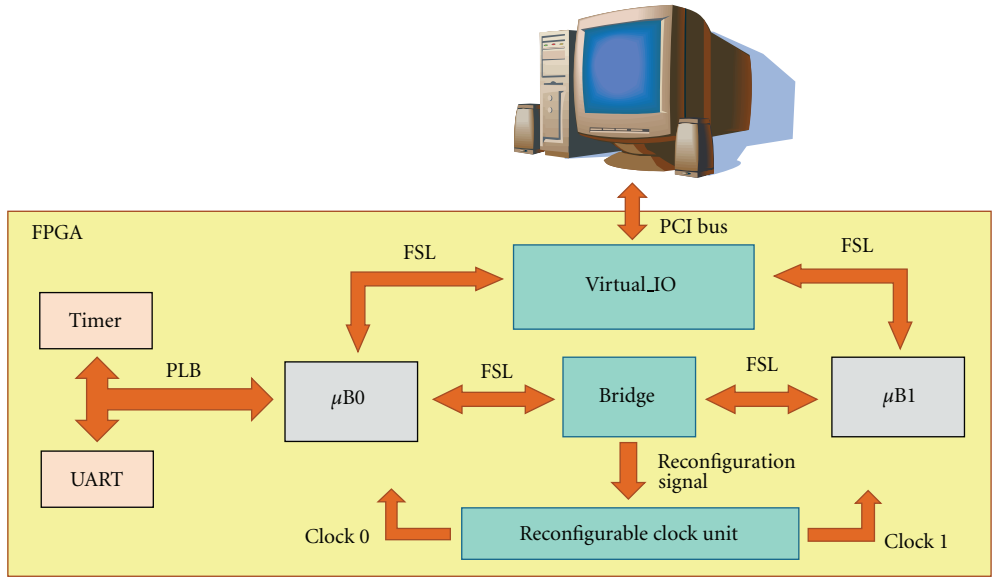
Table 1: Different modules of the virtual-IO component together with their symbol and basic functionality.

Virtual-IO module	Symbol	Function
Virtual-IO 1		Sends data first to μB0 then to μB1. Receives data in the same order
Virtual-IO 2		Sends data only to μB0. Receives data only from μB1
Virtual-IO 3		Sends data first to μB0, then to both and finally only to μB1. Receives data first from μB0, then from μB1
Virtual-IO 4		Sends data only to μB0. Receives data only from μB0. Used for uniprocessor designs
Virtual-IO 5		Sends same data to both processors. Receives results only from μB0
Virtual-IO 6		Sends same data to both processors. Receives results only from μB1

received over μB1. Therefore, μB0 sends its results to μB1, which then sends the results of μB0 together with its own results back to the virtual-IO 2. The third module is virtual-IO 3, which sends first data to μB0. Afterwards, it sends in parallel to both processors μB0 and μB1 the same data. Finally, it sends some data only to μB1. After the execution of the processors, first μB0 and then μB1 send their results back to the virtual-IO 3. The fourth module is virtual-IO 4, which is only connected to one of the processors, for example, μB0. Due to this, this module is used in all uniprocessor designs. For a dual-processor design it sends data to μB0, which then forwards parts of the data to μB1. After execution, μB1 sends its results back to μB0, which forwards the results of the execution of the two processors to the virtual-IO 4. The fifth module is virtual-IO 5, which sends the same data to both processors in parallel, but receives the results only via μB0. The sixth module is virtual-IO 6. It is very similar to virtual-IO 5. The only difference is that it receives the calculation results from μB1 instead of μB0.

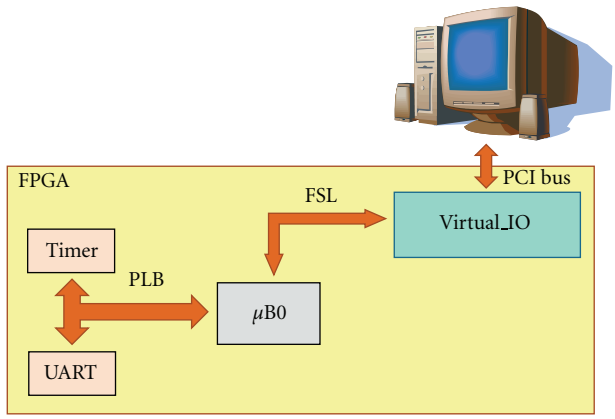
The modules can be selected in the XPS GUI via the parameters of the virtual-IO component. Other parameters that can be set by the user are the number of input and output words for each processor separately, the number of common input words, and the size of the image (only for image processing applications).

4.2. Bridge. The bridge module [12] is used for the interprocessor communication. It consists of two asynchronous FIFOs controlled by FSMs, to support a communication via the two different clock domains of the processors, as shown in Figure 6. This bridge component controls the fill level of



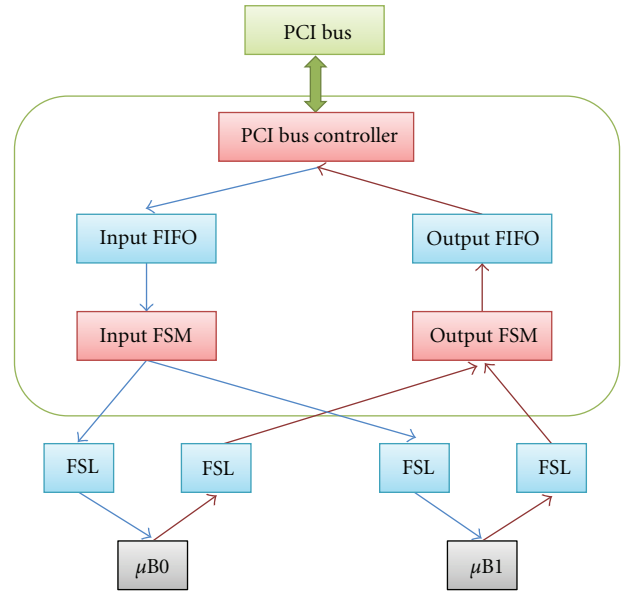
FSL: fast simplex link
PLB: processor local bus
UART: universal asynchronous receiver transmitter
μB: microblaze
PCI: peripheral component interconnect

Figure 3: Dual-processor design with three new components: virtual-IO, bridge, and reconfigurable clock unit.



FSL: fast simplex link
PLB: processor local bus
μB: microblaze
PCI: peripheral component interconnect
UART: universal asynchronous receiver transmitter

Figure 4: Uniprocessor system with the new virtual-IO component to enable a fair comparison with the dual-processor system.



FSL: fast simplex link
FSM: finite state machine
μB: microblaze

Figure 5: Virtual-IO component.

the two FIFOs. If one FIFO tends to be utilized with 75%, it is assumed that the processor, which reads from this FIFO, is too slow. As a result, a reconfiguration signal to increase the clock rate of this processor is sent to the reconfiguration clock unit.

4.3. Reconfigurable Clock Unit. Two different designs for the reconfigurable clock unit have been implemented and will be presented in the following subsections. The first implementation uses hardware reconfiguration to modify the frequency of clock signals. The advantage is that a variety of different clock signals can be provided at runtime. The disadvantage is that the reconfiguration requires a time period of 200 ms, which can be too long, depending on the application.

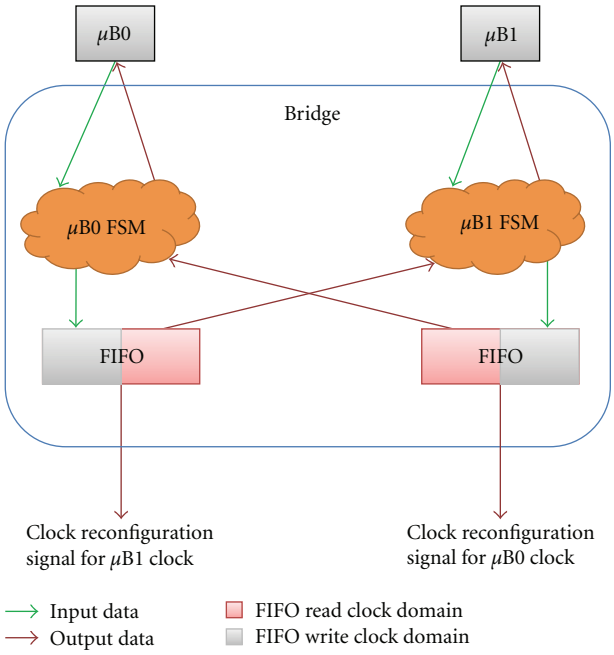


Figure 6: Internal structure of the bridge.

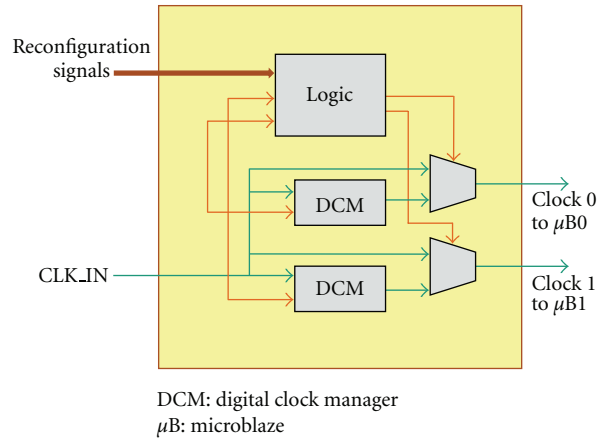


Figure 7: Internal structure of the reconfigurable clock unit.

The second implementation exploits the use of the multiple ports provided by the digital clock manager (DCM) [13] component of the Xilinx FPGAs. Here, clock buffer multiplexer primitives (BUFGMUXes) [14] are used to allow a faster switch between different clocks. The advantage is a faster switching time of few clock cycles between different frequencies. The drawback is that not as many different clocks are possible, as when dynamic reconfiguration is used. The number of the different possible clocks depends on the number of available DCMs and BUFGMUXes on the chosen FPGA device.

4.3.1. Reconfigurable Clock Unit Using Reconfiguration [12]. The internal structure of the reconfigurable clock unit is shown in Figure 7. It consists of two DCMs, two BUFGMUXes, and the logic component, which controls the reconfiguration of the DCMs.

The logic component shown in Figure 7 receives the reconfiguration signals from the bridge component. It then starts the reconfiguration of the DCM primitive for the slower processor. For the reconfiguration purposes, the specific ports provided by Xilinx for dynamic reconfiguration of the Virtex-4 DCM primitive are used. During the reconfiguration process, the DCM has to be kept in a reset state for a minimum of 200 ms. During this time interval, the outputs of this DCM are not stable and cannot be used. Instead of stalling the corresponding processor, the BUFGMUX primitive is used to provide CLK_IN, the original input clock of the two DCM, to the processor, whose DCM is under reconfiguration. The BUFGMUX is a special clock multiplexer primitive, which assures that no glitches occur when switching to a different clock. After the configuration of the DCM is finished, the BUFGMUX is used to switch back to the DCM clock. An alternative would be to stall the processor, while its clock is being reconfigured. Because 200 ms are quite a long time, especially for image processing applications where each 40 ms a new input frame is received from a camera, this would result in a loss of input data.

To prevent an oscillation caused by the frequency scaling mechanism, the controller logic will stop increasing the clock frequency, if 125 MHz for this MicroBlaze have been reached, which is the maximum frequency supported by the MicroBlaze and its peripherals, or if its clock frequency has been increased for three consecutive times. If the reconfiguration signal is furthermore asserted, meaning the processor is still too slow, then the DCM of the faster processor is reconfigured to provide a slower clock to the faster processor.

The internal structure of the logic block is shown in Figure 8. Its function is to supply the reset signal for the correct time span (at least 200 ms) required for correct dynamic reconfiguration (reset controller block) and to also provide the partial reconfiguration data to the corresponding DCM (reconfiguration monitor). The reconfiguration monitor block also monitors the LOCKED signals of the DCMs to know when the partial reconfiguration of a DCM is complete. The clock switcher block has the duty of providing the switching signal to the clock buffer multiplexer at the right time. Besides other minor functions, the logic component also implements the prevention of any race conditions that could result by both bridge component FIFOs having their respective clock reconfiguration signals active at the same time (it would not make sense to reconfigure both DCMs at the same time). For this purpose, the Logic component uses the XOR block on both reconfiguration signals coming from the bridge component, such that only one DCM can be partially reconfigured at one time. For its own synchronous activities, the logic component uses the input clock to the reconfigurable clock unit, which is also the input clock to the DCMs. The logic component also makes sure that the maximum clock frequency supported by the system (125 MHz) is not exceeded (reconfiguration counter block).

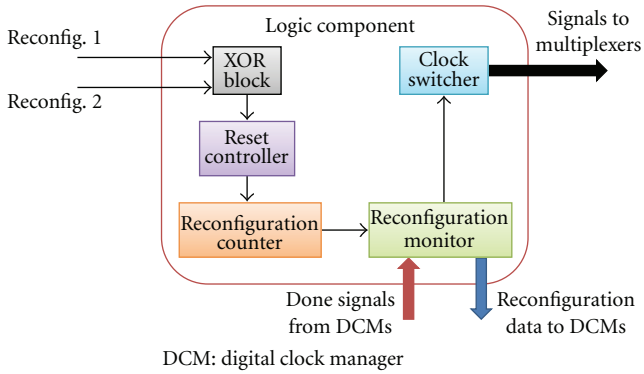


Figure 8: Internal structure of the logic block of the reconfigurable clock unit.

Therefore, the reconfiguration counter block keeps account of the number of times a DCM has been reconfigured such that no DCM can be configured more than 4 times. After 3 consecutive reconfiguration signals from the same bridge FIFO, the logic component actually slows down the originally faster processor by reconfiguring its clock to a lower value.

4.3.2. Reconfigurable Clock Unit Using Clock Buffer Multiplexers. Alternatively, instead of dynamically reconfiguring the DCM, different output ports of a DCM could be used to generate different clocks as shown in Figure 9. Using several BUFGMUXes, the different clocks could be selected.

Figure 10 shows the internal structure of the DCM_wrapper, consisting of two DCMs and several BUFGMUXes to allow fast switching between different clocks.

5. Application Scenarios

Four different applications scenarios were selected to explore the impact of the processor configurations, the task distribution, and the dynamic clock frequency scaling on the power consumption of FPGA-based processor systems. The four different algorithms are described in detail in the next subsections. The first algorithm is the well-known sorting algorithm called Quicksort [15]. It includes a number of branches and comparisons. The second algorithm is an image processing algorithm called normalized squared correlation (NCC), which consists of many arithmetic operations, for example, multiply and divide. The third algorithm is a variation of a bioinformatics algorithm called DIALIGN [16], which contains many comparisons and additions and subtractions. The fourth application is a character recognition algorithm using artificial neural networks (ANNs) [17], which consists of many floating point arithmetic operations. These algorithms with their different algorithm requirements, for example, branches, comparators, multiply and divide, add and subtract, and floating point, were used to provide a user guideline of designing a system with a good performance per power tradeoff for a specific application. By comparing the algorithm requirements of new applications with the four example algorithms, the system configurations of the most similar example algorithm is chosen as a starting system. Such a guideline to limit the design space is very important to save time and achieve a higher time-to-market, because the simulation and the power estimation with XPower are very time consuming. Also, the bitstream generation to measure the performance of the application on the target hardware architecture is time consuming. These long design times can be shortened by starting with an appropriate design, for example, the right processor configurations, a good task distribution, and a well-selected execution frequency.

5.1. Sorting Algorithm: Quicksort. Quicksort [15] is a well-known sorting algorithm with a divide and conquer strategy. It sorts a list by recursively partitioning the list around a pivot and sorting the resulting sublists. It has an average complexity of $O(n \log n)$.

5.2. Image Processing Algorithm: Normalized Squared Correlation. 2-D squared normalized correlation (NCC) is often used to identify an object within an image. The evaluated expression is shown:

$$C(p) = \frac{\left(\sum_{i=0}^{n}\sum_{j=0}^{m}\left(A_p(i,j)-\overline{A_p}\right)\times\left(T(i,j)-\overline{T}\right)\right)^2}{\left(\sum_{i=0}^{n}\sum_{j=0}^{m}\left(A_p(i,j)-\overline{A_p}\right)^2\right)\times\left(\sum_{i=0}^{n}\sum_{j=0}^{m}\left(T(i,j)-\overline{T}\right)^2\right)}, \quad (1)$$

where T: template image with n rows and m columns, A_p: subwindow of the search region with n rows and m columns, $\overline{T}$: mean of T, $\overline{A_p}$: mean of A_p.

This algorithm uses a template T of the object to be searched for and moves this template over the search region A of the image. A_p, the subwindow of the search region at point p with the same size as T, is then correlated with T. The result of this expression is stored at point p in the result image C. The more similar A_p and T are, the higher is the result of the correlation. If they are equal, the result is 1. The object is then detected at the location with the highest value.

5.3. Bioinformatic Algorithm: DIALIGN. DIALIGN [16] is an algorithm from bioinformatics domain, which is used for comparison of the alignment of two genomic sequences. It produces the alignment with the highest number of similar elements and, therefore, the highest score as shown in Figure 11.

5.4. Character Recognition Using Artificial Neural Networks (ANN). The character recognition algorithm was implemented as a multilayer perceptron (MLP) artificial neural network (ANN) to detect the numbers 0 to 9 and the characters "<" and ">". This example was extracted from a project that searches for connected pixels in images, preprocesses the connected pixels and performs the character recognition. This implementation was done according to [17]. For this work, only the preprocessed character set was employed. The network was trained using the preprocessed

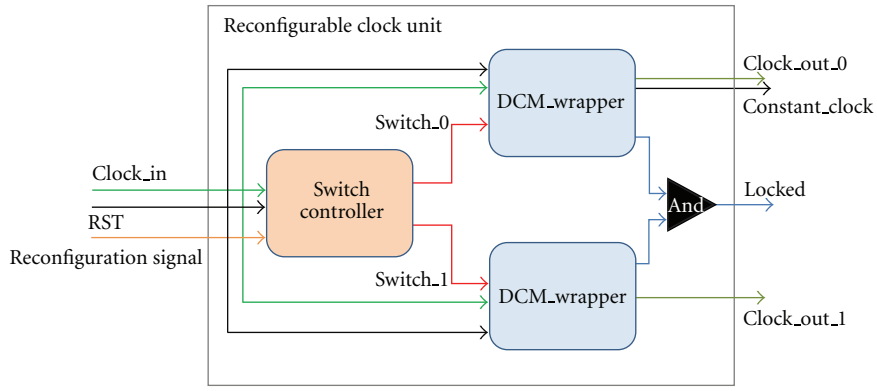


Figure 9: Alternative internal structure of the reconfigurable clock unit.

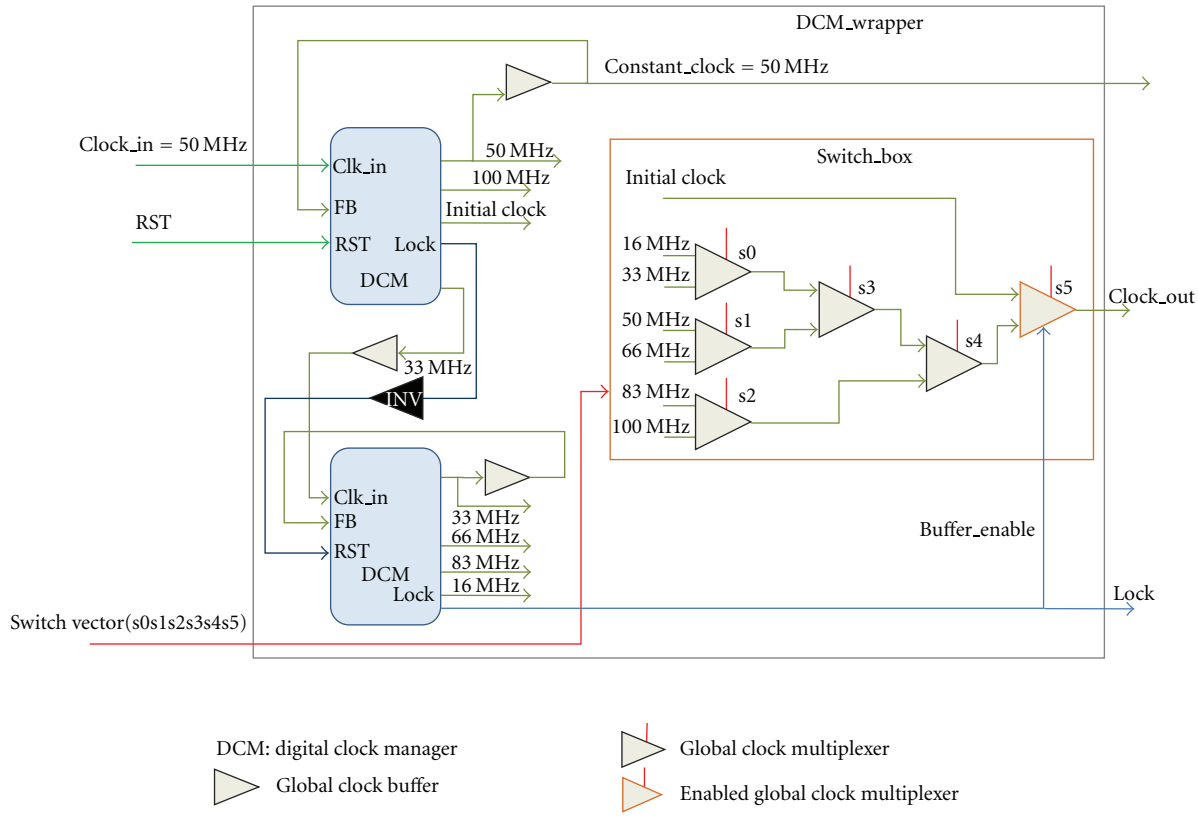


Figure 10: Internal structure of the DCM_wrapper component.

characters depicted in Figure 12. Two-thirds of the images were used for training and one-third for verification.

The trained ANN had 45 neurons in the hidden layer and 12 neurons in the output layer. The input layer contains 91 input elements, so each neuron in the hidden layer has 91 inputs. The output of these hidden neurons is given by (2), where i is the neuron index, j is the input number of the neuron, w_{ij} is the weight applied to the input j in that neuron (i), and b_i is a bias constant value for the neuron i

$$y_i = \frac{1}{1 + e^{(-b_i - \sum_{j=1}^{91} w_{ij} x_{ij})}}. \quad (2)$$

The output of each of the 12 neurons in the output layer is given by (3), where b_k is the bias constant for the output neuron k, y_i is the output of the ith neuron from the previous layer, and w_{ki} is the weight applied to that value. The recognized character is determined by the wining neuron, that is, the neuron for which the output is the nearest to 1;

$$y_k = -b_k - \sum_{i=1}^{12} w_{ki} y_i. \quad (3)$$

The characters were stored in MicroBlaze's main memory as bytes. During execution, they were expanded to arrays

Figure 11: Alignment of two sequences a and b with DIALIGN.

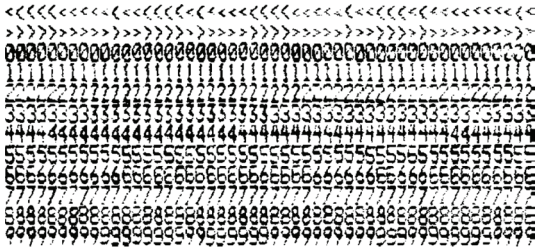


Figure 12: Character set used for training the network.

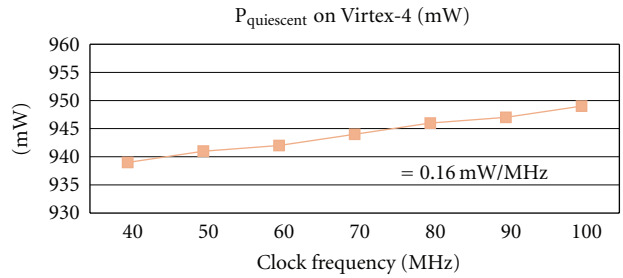


Figure 13: Impact of the clock frequency to the static power consumption of a uniprocessor design on a V4FX100.

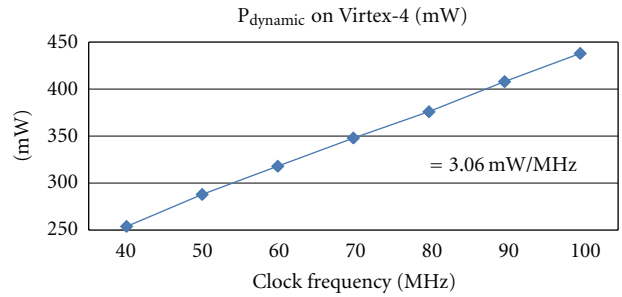


Figure 14: Impact of the clock frequency to the dynamic power consumption of a uniprocessor design on a V4FX100.

where each position contained the value of one pixel of the character image (1 or 0). A total number of 100 characters were presented during the experiments.

6. Integration and Results

Power consumption estimation and performance measurement were done for a Xilinx Virtex-4 (V4FX100) FPGA and a Xilinx Virtex-5 (V5LX110T) FPGA. The performance for the Virtex-4 FPGA was measured on the corresponding FPGA board from Alpha-Data [18] and the performance for the Virtex-5 FPGA was measured on the corresponding Xilinx XUP board. As measuring the exact power consumption of the FPGA on both boards is not possible, it was estimated at design time using the XPower tool flow as described in Section 3. The impact of the clock frequency, the configuration of the processor, and the task distribution onto the power consumption and the performance of the system has been explored, and the results are presented in the following subsections. For each exploration, some parameters had to be kept fixed to assure a fair comparison as shown in Table 2.

For the exploration of the impact of the clock frequency, the algorithm (NCC) and the processor configuration (default: 5-stage pipeline, no arithmetic unit (AU), no floating point unit (FPU)) have been kept fixed. For the exploration of the impact of the processor parameters, (default, AU, reduced pipeline (RP), AU + RP) ± FPU, the clock frequency was kept fixed at 100 MHz.

Finally, for the exploration of the task distribution, the processor configuration and the performance were kept fixed to lower the overall system power consumption, while maintaining the performance similar to the performance achieved with a reference uniprocessor design running at 100 MHz, which is a standard frequency for both Virtex-4- and Virtex-5-based MicroBlaze systems.

6.1. Impact of the Clock Frequency. In the following two subsections, the impact of the variation of the clock frequency to the power consumption of a Virtex-4 and a Virtex-5 FPGA was explored using a uniprocessor system, which executes the NCC algorithm on one MicroBlaze. The MicroBlaze was configured to have a 5-stage pipeline and no arithmetic unit. No AU means that the MicroBlaze has no integer multiplier/divider, no pattern comparator, and no barrel shifter. For the power consumption estimation, the results for quiescent, dynamic, and overall power consumption are given. The quiescent power consumption is also called static power consumption in the following, because it represents the power consumption of the user-configured FPGA without any switching activity. Furthermore, the execution time and the resulting energy consumption are presented. For each selected clock frequency, the processor system has been recompiled with the appropriate clock constraints.

6.1.1. Virtex-4 FX 100 FPGA. The results for the dynamic, the quiescent, and the total power consumption for the Virtex-4 FPGA are presented in Table 3 together with the execution time and the overall energy consumption.

The impact of the clock frequency onto the static and the dynamic power consumption for the Virtex-4 FPGA is presented in Figures 13 and 14, respectively. As can be seen, the static power consumption increases by around 0.16 mW/MHz while the dynamic power consumption increases by around 3.06 mW/MHz.

Table 2: Fixed parameters for the exploration of the impact of the clock frequency, the processor configuration, and the task distribution on the performance, power dissipation, and energy consumption.

Impact of	Fixed parameters	Variable parameters
Clock frequency	(i) Algorithm: NCC (ii) Processor configuration: default (5-stage pipeline, no AU, no FPU) (iii) # of processors: 1	(i) Clock frequency: 40–100 MHz (ii) FPGA: Virtex-4, Virtex-5
Processor configuration	(i) Clock frequency: 100 MHz (ii) No. of processors: 1	(i) Processor configurations: (default, AU, RP, RP + AU) ± FPU (ii) Algorithm: NCC, Quicksort, DIALIGN, ANN (iii) FPGA: Virtex-4, Virtex-5
Task distribution	(i) Execution time = execution time of a uniprocessor design at 100 MHz (ii) Processor configuration: 5-stage pipeline, integer multiplier, pattern comparator (iii) No. of processors: 2 (iv) FPGA: Virtex-4	(i) Application partitioning (ii) Algorithm: NCC, Quicksort, DIALIGN

Table 3: Impact of the variation of the clock frequency to the power consumption for V4FX100.

Clk Freq. (MHz)	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
40	254	939	1192	−14,1	791,11	943,00
50	288	941	1229	−11,4	632,89	777,82
60	318	942	1260	−9,2	527,41	664,53
70	348	944	1292	−6,8	452,06	584,06
80	376	946	1322	−4,7	395,55	522,92
90	408	947	1355	−2,3	351,60	476,42
100	438	949	1387	NA	316,44	438,91

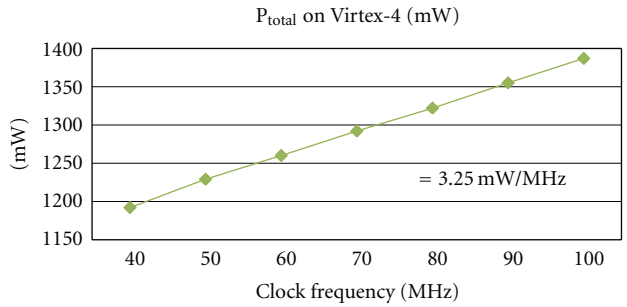


Figure 15: Impact of the clock frequency to the total power consumption of a uniprocessor on a V4FX100.

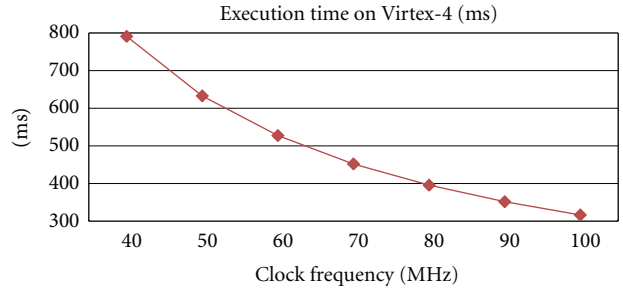


Figure 16: Impact of the clock frequency to the execution time of a uniprocessor design on a V4FX100.

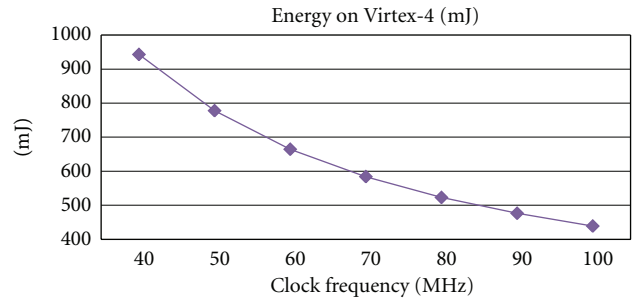


Figure 17: Impact of the clock frequency to the energy consumption of a uniprocessor design on a V4FX100.

Out of this results the impact onto the total power consumption, which is around 3.25 mW/MHz. The impact on the total power consumption is shown in Figure 15.

The impact on the execution time is shown in Figure 16.

The impact onto the overall energy consumption, which is around 8.42 mJ/MHz is shown in Figure 17.

6.1.2. Virtex-5 LX110T FPGA. The results for the dynamic, the quiescent, and the total power consumption together with the execution time and the overall energy consumption for the Virtex-5 FPGA are given in Table 4. As can be seen, the static power consumption is higher while the dynamic power consumption is lower compared to the Virtex-4 FPGA. This can be derived from the different CMOS processes: Virtex-5 has a 65 nm process while Virtex-4 has a 90 nm process.

The impact of the clock frequency onto the static and the dynamic power consumption for the Virtex-5 FPGA is presented in Figures 18 and 19, respectively. Here, a different behavior, compared to the Virtex-4 FPGA, can be seen. Due to the new Virtex-5 architecture, the static power

Table 4: Impact of the variation of the clock frequency onto the power consumption for V5LX110T.

Clk Freq. (MHz)	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
40	215	1230	1444	−5,87	791,11	1142,36
50	229	1230	1459	−4,89	632,89	923,38
60	223	1230	1453	−5,28	527,41	766,32
70	258	1231	1489	−2,93	452,06	673,12
80	254	1231	1485	−3,19	395,55	587,40
90	238	1230	1468	−4,30	351,60	516,16
100	302	1232	1534	NA	316,44	485,42

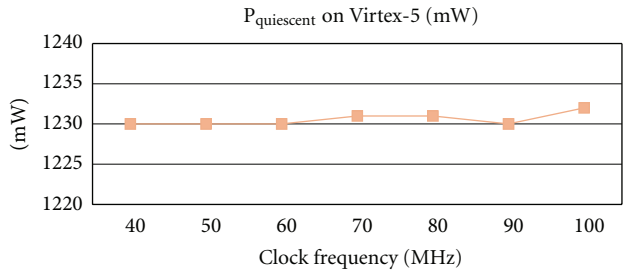


Figure 18: Impact of the clock frequency to the static power consumption of a uniprocessor design on a V5LX110T.

consumption is nearly constant for all clock frequencies. Minor variations occur to different placements of the design. The dynamic power consumption shows an unsteady behavior, when varying the clock frequency. A reason for this unsteady behavior could lie in the PAR process for Virtex-5 FPGAs, where different resources have been chosen for the different clock frequencies of the designs.

The total power consumption results from adding the dynamic and the static power consumption. As the static power consumption is nearly constant, the total power consumption shows the same behaviour over different clock frequencies than the dynamic power consumption. In total, the power consumption of the Virtex-5LX110T is around 200 mW higher than the one of the Virtex-4FX100 FPGA. The impact on the total power consumption is shown in Figure 20.

The impact on the execution time is equal for both FPGAs and is shown in Figure 21.

The overall energy consumption for the different clock frequencies is shown in Figure 22.

6.2. Impact of the Processor Configurations. For exploration purposes, a uniprocessor design consisting of a single MicroBlaze running at 100 MHz was used. The results were compared against a reference configuration, which was a MicroBlaze with a 5-stage pipeline and no arithmetic unit, which means no integer multiplier, no integer divider, no barrel shifter, and no pattern comparator. The following

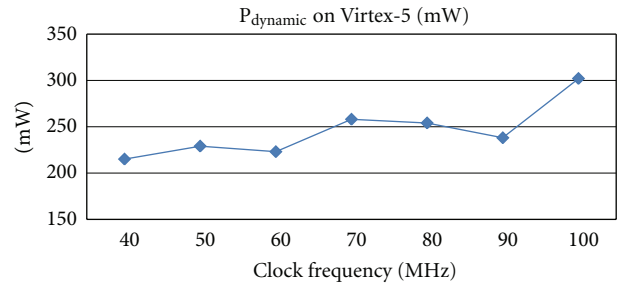


Figure 19: Impact of the clock frequency to the dynamic power consumption of a uniprocessor design on a V5LX110T.

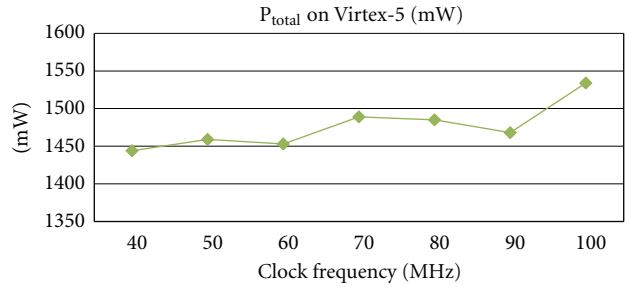


Figure 20: Impact of the clock frequency to the total power consumption of a uniprocessor design on a V5LX110T.

configurations were explored for the NCC, Quicksort and DIALIGN:

(i) default,
(ii) adding an arithmetic unit (AU),
(iii) reduction of the pipeline to 3 stages (RP),
(iv) combination of (i) and (iii) (AU + RP).

For the ANN, the configurations differ, as an additional parameter, the FPU, was added:

(i) default + FPU,
(ii) adding an arithmetic unit (AU + FPU),
(iii) reduction of the pipeline to 3 stages (RP + FPU),
(iv) combination of (i) and (iii) (AU + RP + FPU),
(v) arithmetic unit without FPU (AU).

The impact to the power consumption, the performance, and the energy consumption was explored for all four algorithms and is presented in the following subsections for Virtex-4 and Virtex-5, respectively. The impact is very different between the selected applications, due to the different algorithm requirements, as mentioned in Section 5 and its subsections.

6.2.1. Virtex-4 FX 100 FPGA. Figure 23 and Table 5 show the impact of the different configurations for the Quicksort algorithm. The combination of a reduction of the pipeline stages and the addition of the arithmetic unit provides the best solution in terms of performance, power and energy

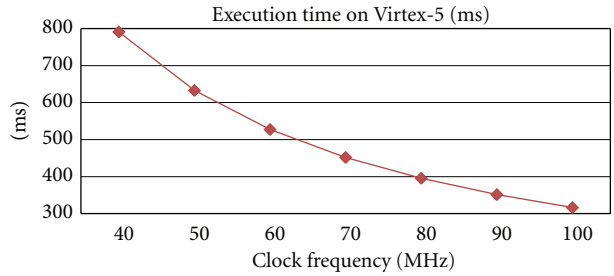


Figure 21: Impact of the clock frequency to the execution time of a uniprocessor design executing the NCC algorithm on a V5LX110T.

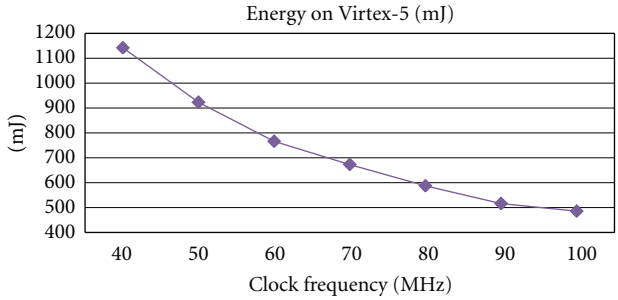


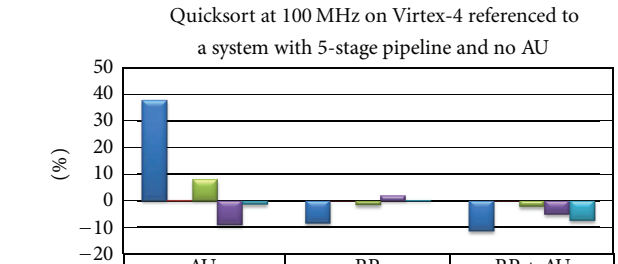
Figure 22: Impact of the clock frequency onto the overall energy consumption of a uniprocessor design executing the NCC algorithm on a V5LX110T.

Table 5: Impact of the MicroBlaze configurations (μB param) for the Quicksort algorithm at 100 MHz for V4FX100.

μB param	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
Default	243	938	1181	NA	3,02	3,57
AU	336	943	1280	8,38	2,74	3,51
RP	222	937	1159	−1,86	3,09	3,58
RP + AU	215	937	1152	−2,46	2,86	3,29

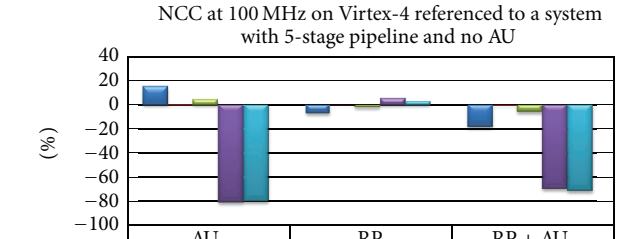
consumption. The reasons for this are, on the one hand, the multiple branches in the algorithm, which benefit from a reduction of the pipeline stages and, on the other hand, the multiple comparators, which benefit from the pattern comparator within the arithmetic unit. Therefore, the RP + AU system would be chosen. Using these systems, further performance and power evaluations could be done by adding or removing the different internal configurations of the arithmetic unit, as probably some, for example the integer divider, are not needed by the Quicksort algorithm, and, therefore, consume power but do not improve the overall performance.

Figure 24 and Table 6 show the impact of the different configurations for the NCC algorithm. As this algorithm requires many arithmetic operations, the addition of an AU improves the overall execution time very strongly (over 80%) while the reduction of the pipeline stages results in a slight degradation. This degradation is due to the reason that the execution of arithmetic operations takes more clock cycles, if the pipeline is reduced. Therefore, for this and similar



	AU	RP	RP + AU
Pdynamic	38.27	−8.64	−11.52
Pquiescent	0.53	−0.11	−0.11
Ptotal	8.38	−1.86	−2.46
Execution time	−9.27	2.32	−5.30
Energy	−1.67	0.41	−7.62

Figure 23: Impact of the MicroBlaze configurations for the Quicksort algorithm.



	AU	RP	RP + AU
Pdynamic	15.53	−7.53	−19.18
Pquiescent	0.42	−0.21	−0.53
Ptotal	5.12	−2.45	−6.42
Execution time	−82.55	6.04	−71.08
Energy	−81.66	3.44	−72.94

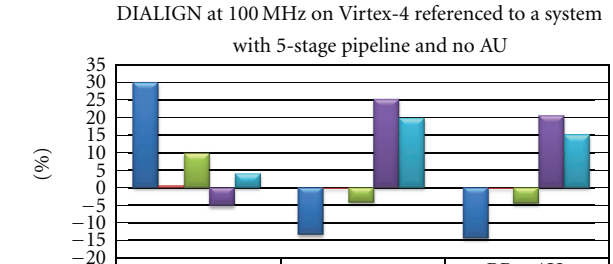
Figure 24: Impact of the MicroBlaze configurations for the NCC algorithm.

Table 6: Impact of the MicroBlaze configurations for the NCC algorithm at 100 MHz for V4FX100.

μB param	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
Default	438	949	1387	NA	316,4	438,85
AU	506	953	1458	5,12	55,2	80,48
RP	405	947	1353	−2,45	335,5	453,93
RP + AU	354	944	1298	−6,42	91,5	118,77

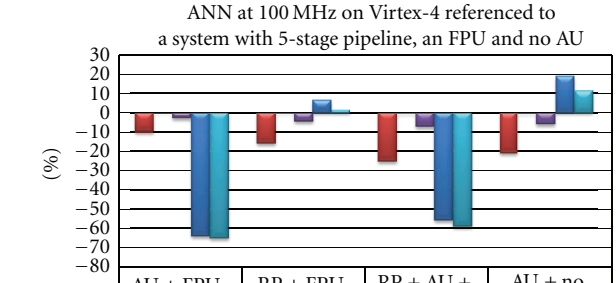
algorithms, a system with an AU and a 5-stage pipeline would be optimal from a performance and energy consumption perspective. If the power consumption needs to be reduced and some performance degradation is acceptable, then the AU + RP system would be a good choice.

In Figure 25 and Table 7, the impact to the performance and power consumption of the three different processor configurations compared to the reference system are presented for the DIALIGN algorithm. Adding an AU improves the execution time by 5% while increasing the overall power



	AU	RP	RP + AU
Pdynamic	30.37	−13.32	−14.25
Pquiescent	0.84	−0.32	−0.32
Ptotal	10.03	−4.36	−4.65
Execution time	−5.28	25.62	20.91
Energy	4.22	20.14	15.28

Figure 25: Impact of the MicroBlaze configurations for the DIALIGN algorithm.



	AU + FPU	RP + FPU	RP + AU + FPU	AU + no FPU
Pdynamic	−10.37	−16.38	−25.82	−21.41
Pquiescent	−0.23	−0.36	−0.56	−0.47
Ptotal	−3.06	−4.83	−7.61	−6.31
Execution time	−64.95	7.02	−56.64	19.21
Energy	−66.02	1.85	−59.94	11.69

Figure 26: Impact of the MicroBlaze configurations for the ANN algorithm.

Table 7: Impact of the MicroBlaze configurations for the DIALIGN algorithm at 100 MHz for V4FX100.

μB param	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
Default	428	948	1376	NA	829,8	1141,80
AU	558	956	1514	10,03	786	1190,00
RP	371	945	1316	−4,36	1042,4	1371,80
RP + AU	367	945	1312	−4,65	1003,3	1316,33

Table 8: Impact of the MicroBlaze configurations for the ANN algorithm at 100 MHz for V4FX100.

μB param	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
Default + FPU	365,85	944,95	1311	NA	2328	3052
AU + FPU	327,9	942,81	1271	−3,06	816	1037
RP + FPU	305,94	941,58	1248	−4,83	2491	3108
RP + AU + FPU	271,37	939,65	1211	−7,61	1009	1222
AU + no FPU	287,52	940,55	1228	−6,31	2775	3408

and energy consumption compared to the reference design by 10% and 4%. The reduction of the pipeline to 3 stages improves the total power consumption by 4%, but worsening the execution time by 25% and the energy consumption by 20%. The combination of AU + RP shows nearly the same impact as the RP system. Therefore, the reference system is the best choice.

In Figure 26 and Table 8, the impact to the performance and power consumption of the four different processor configurations compared to the reference system are presented for the ANN algorithm.

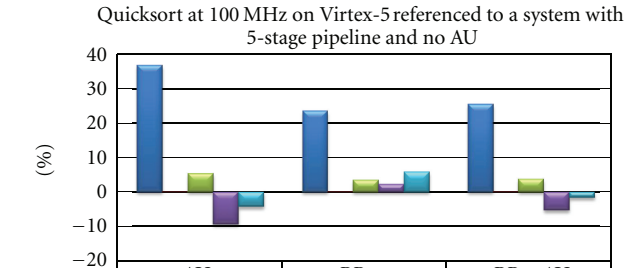
Adding an AU improves the execution time by 65% and reduces the overall power and energy consumption compared to the reference design by 3% and 66%. The reduction of the pipeline to 3 stages slightly improves the total power consumption by 4%, but worsening the execution time by 7%. The combination of AU + RP shows nearly the same impact as the AU system. Finally, the system with AU and without an FPU shows a slight improvement of the power consumption by 6% while the execution time is increased by 19%, and, also, the energy consumption is increased by over 11%. The reason for the longer execution time and, therefore, the higher energy consumption for the non-FPU system is due to the fact that the ANN uses floating point operations, which require much more clock cycles, if no FPU is provided. Therefore, the AU + FPU system is the best choice in terms of performance and energy consumption. If the power consumption is more important, then the RP + AU + FPU system would be the best choice for this algorithm.

Table 9: Impact of the MicroBlaze configurations for the Quicksort algorithm at 100 MHz for V5LX110T.

μB param	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
Default	211	1230	1441	NA	3,02	4,35
AU	289	1231	1520	5,48	2,74	4,16
RP	261	1231	1492	3,54	3,09	4,61
RP + AU	265	1231	1496	3,82	2,86	4,28

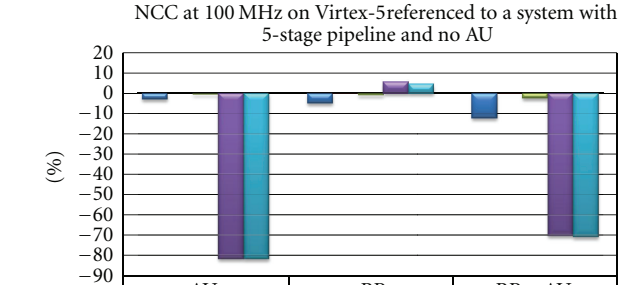
6.2.2. Virtex-5 LX110T FPGA. Figure 27 and Table 9 show the impact of the different configurations for the Quicksort algorithm. Due to the multiple comparators and the multiple branches in the algorithm, the combination of an arithmetic unit and a reduction of the pipeline stages is very beneficial in terms of execution time and energy consumption, but results in increased power consumption. If the power consumption is the critical factor, then the default system would be the best choice.

Figure 28 and Table 10 show the impact of the different configurations for the NCC algorithm. As this algorithm requires many arithmetic operations, the addition of an AU improves the overall execution time and the energy



	AU	RP	RP + AU
Pdynamic	36.97	23.7	25.59
Pquiescent	0.08	0.08	0.08
Ptotal	5.48	3.54	3.82
Execution time	−9.27	2.32	−5.3
Energy	−4.3	5.94	−1.68

Figure 27: Impact of the MicroBlaze configurations for the Quicksort algorithm.



	AU	RP	RP + AU
Pdynamic	−2.98	−4.97	−12.25
Pquiescent	0	−0.08	−0.08
Ptotal	−0.59	−0.98	−2.48
Execution time	−82.55	6.04	−71.08
Energy	−82.66	5	−71.8

Figure 28: Impact of the MicroBlaze configurations for the NCC algorithm.

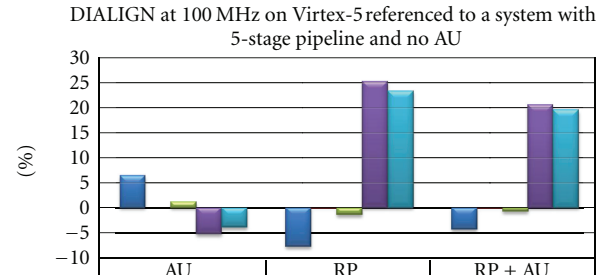
consumption by over 80% while the reduction of the pipeline stages results in a slight degradation. This degradation is due to the reason that the execution of arithmetic operations takes more clock cycles, if the pipeline is reduced. Therefore, for this and similar algorithms, a system with an AU and a 5-stage pipeline would be optimal from a performance and energy perspective. If the power consumption needs to be reduced and some performance degradation is acceptable, then the AU + RP system would be a good choice.

In Figure 29 and Table 11, the impact onto the performance and power consumption of the three different processor configurations compared to the reference system is presented for the DIALIGN algorithm.

Adding an AU improves the execution time and the energy consumption a little bit while slightly increasing the overall power consumption compared to the reference design. The reduction of the pipeline to 3 stages improves the total power consumption slightly, but worsening

Table 10: Impact of the MicroBlaze configurations for the NCC algorithm at 100 MHz for V5LX110T.

μB param	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
Default	302	1232	1534	NA	316,4	485,36
AU	293	1232	1525	−0,59	55,2	84,18
RP	287	1231	1519	−0,98	335,5	509,62
RP + AU	265	1231	1496	−2,48	91,5	136,884



	AU	RP	RP + AU
Pdynamic	6.69	−7.69	−4.35
Pquiescent	0	−0.08	−0.08
Ptotal	1.37	−1.5	−0.85
Execution time	−5.28	25.62	20.91
Energy	−3.98	23.73	19.88

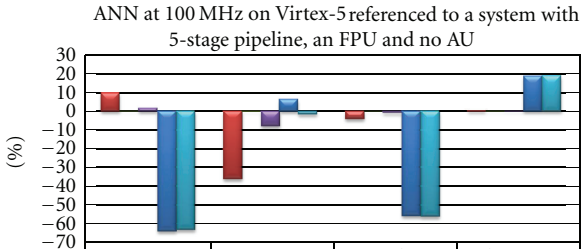
Figure 29: Impact of the MicroBlaze configurations for the DIALIGN algorithm.

Table 11: Impact of the MicroBlaze configurations for the DIALIGN algorithm at 100 MHz for V5LX110T.

μB param	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
Default	299	1232	1530	NA	829,8	1270
AU	319	1232	1551	1,37	786	1219
RP	276	1231	1507	−1,50	1042,4	1571
RP + AU	286	1231	1517	−0,85	1003,3	1522

the execution time by 25% and, therefore, the energy consumption by over 23%. The combination of AU + RP shows nearly the same impact as the RP system. Therefore, the AU system is the best choice for these kinds of algorithms.

Figure 30 and Table 12 show the impact onto the performance and power consumption of the four different processor configurations compared to the reference system are presented for the ANN algorithm. Adding an AU improves the execution time by 65% and reduces the overall energy consumption compared to the reference design by 64% while the power consumption is increased slightly by 2%. The reduction of the pipeline to 3 stages improves the total power consumption by 8% and the energy consumption by 1.69%, but worsening the execution time by 7%. The combination of AU + RP shows nearly the same impact as the AU system. Finally, the system with AU and without an FPU shows a slight increase of the power consumption by 0.23% while



	AU + FPU	RP + FPU	RP + AU + FPU	AU + no FPU
Pdynamic	10.37	−36.71	−4.25	1.05
Pquiescent	0.06	−0.21	−0.02	0.01
Ptotal	2.3	−8.13	−0.94	0.23
Execution time	−64.95	7.02	−56.64	19.21
Energy	−64.14	−1.69	−57.05	19.49

Figure 30: Impact of the MicroBlaze configurations for the ANN algorithm.

Table 12: Impact of the MicroBlaze configurations for the ANN algorithm at 100 MHz for V5LX110T.

μB param	$P_{Dynamic}$ (mW)	$P_{Quiescent}$ (mW)	P_{Total} (mW)	P_{Total} (%)	T_{exe} (ms)	Energy (mJ)
Default + FPU	341,75	1232,59	1574,34	NA	2328	3665
AU + FPU	377,18	1233,33	1610,51	2,30	816	1314
RP + FPU	216,3	1229,98	1446,28	−8,13	2491	3603
RP + AU + FPU	327,21	1232,29	1559,49	−0,94	1009	1574
AU + no FPU	345,34	1232,67	1578,01	0,23	2775	4379

Table 13: Quicksort power and energy consumption.

	Uniprocessor (100 MHz)	Dual_2 (80/50 MHz)	Dual_5 (95 MHz)
Execution time-ms	18,42	18,80	19,27
Total power-mW	1576,93	1475,56	1570,79
Total power %	NA	−6,43	−0,39
Total energy-mJ	29,05	27,74	30,27
Total energy %	NA	−4,51	+4,20

the execution time and the energy consumption are both increased by 19%. The reason for the longer execution time and the higher energy consumption for the non-FPU system is that the ANN uses floating point operations, which require much more clock cycles, if no FPU is provided. Therefore, the AU + FPU system is the best choice in terms of performance and energy consumption. If the power consumption is more important, then the RP + AU + FPU system would be the best choice for this algorithm.

6.3. Impact of the Task Distribution and the Frequency Scaling. To measure the impact onto the power and energy consumption, the algorithms were partitioned onto two MicroBlaze processors on the Virtex-4 FPGA. The frequency for the two processors was chosen in such a way that the execution time of the dual-processor design was as similar as possible to the reference system consisting of a single MicroBlaze running at 100 MHz. For all systems, the configurations of the processors were fixed to a 5-stage pipeline, an integer multiplier, and the pattern comparator. For the reconfigurable clock, the first version using reconfiguration was chosen.

Table 14: NCC power and energy consumption.

	Uniprocessor (100 MHz)	Dual_3 (54 MHz)	Dual_2 (87.5/50 MHz)
Execution time-ms	67,74	67,28	67,62
Total power-mW	1472,78	1477,20	1504,02
Total power %	NA	+0,30	+2,12
Total energy-mJ	99,77	99,39	101,70
Total energy %	NA	−0,38	+1,93

Table 15: DIALIGN power and energy consumption.

	Uniprocessor (100 MHz)	Dual_5 (50 MHz)	Dual_6 (50 MHz)
Execution time-ms	30,21	30,16	30,16
Total power-mW	1569,69	1631,62	1536,44
Total power %	NA	+3,95	−2,12
Total energy-mJ	47,42	49,21	46,34
Total energy %	NA	+3,77	−2,28

Table 13 shows the results for distributing the Quicksort algorithm on two processors instead of one. Two partitions were done. The first one is called Dual_2 (80/50 MHz), which means that the virtual-IO 2 was used, and μB0 was running at 80 MHz while μB1 was running at 50 MHz. The algorithm was so partitioned that μB0 receives the whole data to be sorted. It then divides the data into two parts and sends the second part to μB1. Both then sort their partition. μB0 forwards its sorted part of the list to μB1, which sends the final combined sorted list via the virtual-IO 2 to the host PC. With this partition, the overall power consumption could be reduced by 6.43% and the energy consumption by 4.51% compared to the single processor reference system.

The second partition, called Dual_5 (95 MHz), uses the virtual-IO 5 to send incoming data to both processors running at 95 MHz. μB0 searches the list for elements smaller than and μB1 searches the list for elements bigger than, the pivot. When one has found an element the position of this element is sent to the other processor. Both processor then update their lists by swapping the own-found element with the one the other processor has found. At the end, both processors have, as a result, a searched list. μB0 then sends its resulting list back to the host PC via the virtual-IO 5. The power consumption of this version is nearly the same as the reference system while the total execution time and, therefore, the energy consumption increase.

Table 14 shows the result for the partitioning of the NCC algorithm onto two processors. The first partitioning uses the virtual-IO 3 to partition the incoming image into two

Table 16: First approach for a developers guideline.

Algorithm classifiers	Example algorithm	Processor configuration	Task distribution
(i) Comparators (ii) Branch and bound	Quicksort	RP + AU	Dual_2 (80/50 MHz)
(i) Complex arithmetic	NCC	AU	Uniprocessor (100 MHz)
(i) Comparators (ii) Basic arithmetic	DIALIGN	Default (V4), AU (V5)	Dual_6 (50 MHz)
(i) Floating point (ii) Complex arithmetic	ANN	AU + FPU	Needs to be explored

overlapping tiles, one for each processor. The overlapping part is sent to both processors simultaneously. As the NCC is a window-based image processing algorithm, the boarder pixels between the two tiles are needed by both processors. Each of the processors runs at 54 MHz, which results in a similar execution time, and also in a similar total power and energy consumption as the reference design.

The second partition, called Dual_2 (87,5/50 MHz), uses virtual-IO 2 to send the whole image to μB0. μB0 runs at 87.5 MHz and calculates the complete numerator and the denominator. Then, it forwards both to μB1, which does the division and sends the results back to the virtual-IO 2. μB1 runs at 50 MHz. While the execution time is nearly the same, the overall power consumption is increased slightly by 2.12% and the energy consumption by 1.93%.

Table 15 shows the result for executing the DIALIGN algorithm with two processors. Two partitions were done. The first one is called Dual_5 (50 MHz) and uses virtual-IO 5 to send the incoming sequences to both processors running at 50 MHz. Each processor calculates half of the resulting score matrix. μB0 calculates on a row-based fashion all values above the main diagonal. μB1 calculates in a column-based fashion all values below the main diagonal. The scores on the main diagonal are calculated by both processors. After μB0 has finished calculating one row and μB1 one column respectively, they exchange the first score nearest to the main diagonal, as this score is needed by both processors for calculating the next row/column, respectively. While the execution time is nearly the same, the overall power consumption is increased by 3.95% and the energy consumption by 3.77%.

The second partition is called Dual_6 (50 MHz). It uses the virtual-IO 6 to send the sequences to the processors, which run both at 50 MHz. Here, a systolic array approach is used for executing the DIALIGN algorithm. μB1 then sends the final alignment and the score back to the host PC. With this partition, the overall power consumption could be reduced by 2.12% and the energy consumption by 2.28% compared to the single processor reference system.

6.4. First Approach for a Developers Guideline. Table 16 is a first approach for a developer's guideline based on the exploration results done so far.

This guideline will be extended, by exploring more types of algorithms, more different FPGA families, and more detailed explorations for the processor configurations.

7. Conclusions and Outlook

This paper reports the research and evaluation of different microprocessor parameterization, application, and data partitioning on FPGA-based processor systems. Two different FPGA families are explored: Xilinx Virtex-4 and Virtex-5 FPGAs. The results of the experiments show the impact of the different parameterization on the power dissipation and energy consumption as well as performance in relation to a set of selected applications. Depending on the application type, it can be seen that different parameter configurations, for example configuration of the processors and their frequencies, but also a good application partitioning, are essential for achieving an efficient tradeoff between performance and power constraints. The results can be used to guide developers which parameter set suits to a certain application scenario, as was shown in Table 16. One important aspect studied in this work is the energy consumption of the different designs. In the experiments performed, it is noticed that the correct choice of the microprocessor configuration can lead to an economy of up to 90% of the energy consumption. This is significant especially regarding embedded applications, which normally depend on batteries to the power supply. Furthermore, the results show that for the selected FPGAs, DFS without any scheme to reduce the voltage is poorly interesting in terms of energy consumption. Under this condition for reducing the energy consumption, the policy should be compute as fast as possible and with the appropriate processor configuration and then to shut the power down.

The vision is that more application scenarios will be analyzed in order to provide a broad overview of the parameter impact. It is envisioned to extend existing hardware benchmarks from different application domains in terms of a parameterization guideline also for further FPGA series from Xilinx.

In addition, the paper provides a tutorial for the estimation of the power consumption on a high level of abstraction, but with a high accuracy through postplace and route simulation. Therefore, other research in this area can be done and exchanged in the community.

Acknowledgments

The authors would like to thank Professor Alba Cristina M. A. De Melo and Jan Mendonca Correa for providing

them with their C code implementation of the DIALIGN algorithm.

References

[1] "Xilinx MicroBlaze Reference Guide," UG081 (v7.0), September 2006, http://www.xilinx.com/.

[2] D. Meintanis and I. Papaefstathiou, "Power consumption estimations vs measurements for FPGA-based security cores," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 433–437, Cancun, Mexico, December 2008.

[3] J. Becker, M. Huebner, and M. Ullmann, "Power estimation and power measurement of Xilinx virtex FPGAs: trade-offs and limitations," in *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI '03)*, Sao Paulo, Brazil, September 2003.

[4] K. Poon, A. Yan, and S. J. E. Wilton, "A flexible power model for FPGAs," in *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL '02)*, September 2002.

[5] F. N. Najm, "Transition density: a new measure of activity in digital circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 2, pp. 310–323, 1993.

[6] K. Weiss, C. Oetker, I. Katchan, T. Steckstor, and W. Rosenstiel, "Power estimation approach for SRAM-based FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '00)*, pp. 195–202, Monterey, Calif, USA, February 2000.

[7] V. Degalahal and T. Tuan, "Methodology for high level estimation of FPGA power consumption," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '05)*, Shanghai, China, January 2005.

[8] "Xilinx Power Estimator User Guide," UG440 (v3.0), June 2009, http://www.xilinx.com/.

[9] "Development System Reference Guide," v9.2i, Chapter 10 XPower, http://www.xilinx.com/.

[10] "Embedded System Tools Reference Manual," Embedded Development Kit, EDK 9.2i, UG111 (v9.2i), Chapter 3, September 2007, http://www.xilinx.com/.

[11] "Fast Simplex Link (FSL) Bus (v2.00a)," DS449 December 2005, http://www.xilinx.com/.

[12] D. Göhringer, J. Obie, M. Hübner, and J. Becker, "Impact of task distribution, processor configurations and dynamic clock frequency scaling on the power consumption of FPGA-based multiprocessors," in *Proceedings of the 5th International Workshop on Reconfigurable Communication Centric Systems-on-Chip (ReCoSoC '10)*, Karlsruhe, Germany, May 2010.

[13] "Virtex-4 FPGA Configuration User Guide," UG071 (v1.11), June 2009, http://www.xilinx.com/.

[14] "Virtex-4 FPGA User Guide," UG070 (v2.6), December 2008, http://www.xilinx.com/.

[15] C. A. R. Hoare, "Quicksort," *Computer Journal*, vol. 5, no. 1, pp. 10–15, 1962.

[16] A. Boukerche, J. M. Correa, A. C. M. Melo, and R. P. Jacobi, "A hardware accelerator for the fast retrieval of DIALIGN biological sequence alignments in linear space," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 808–821, 2010.

[17] R. Palacios and A. Gupta, "A system for processing handwritten bank checks automatically," *Image and Vision Computing*, vol. 26, no. 10, pp. 1297–1313, 2008.

[18] "Alpha-Data," http://www.alpha-data.com/.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 820454, 19 pages
doi:10.1155/2011/820454

Review Article

A Security Scheme for Dependable Key Insertion in Mobile Embedded Devices

Alexander Klimm, Benjamin Glas, Matthias Wachs, Sebastian Vogel, Klaus D. Müller-Glaser, and Jürgen Becker

Institute for Information Processing Technology, Karlsruhe Institute of Technology (KIT), 76021 Karlsruhe, Germany

Correspondence should be addressed to Alexander Klimm, klimm@kit.edu

Received 27 August 2010; Revised 5 February 2011; Accepted 10 February 2011

Academic Editor: Michael Hübner

Public Key Cryptography enables entity authentication protocols based on a platform's knowledge of other platforms' public key. This is particularly advantageous for embedded systems, such as FPGA platforms, with limited or none read-protected memory resources. For access control systems, an access token is authenticated by the mobile system. Only the public key of authorized tokens needs to be stored inside the mobile platform. At some point during the platform's lifetime, these might need to be updated in the field due to loss or damage of tokens. This paper proposes a holistic approach for an automotive access control system based on Public Key Cryptography. Next to a FPGA-based hardware architecture, we focus on a secure scheme for key flashing of public keys to highly mobile systems. The main goal of the proposed scheme is the minimization of online dependencies to Trusted Third Parties, Certification Authorities, or the like, to enable key flashing in remote locations with only minor technical infrastructure. Introducing trusted mediator devices, new tokens can be authorized and later their public key can be flashed into a mobile system on demand.

1. Introduction

Embedded systems in various safety critical application domains such as automotive, avionic, and medical care perform more and more complex tasks using distributed systems like networks of electronic control units (ECUs). Introducing Public Key Cryptography (PKC) to embedded systems provides essential benefits for the fabrication of electronic units needing to meet security requirements as well as for the logistics involved. Due to the nature of PKC, the number of keys that need to be stored in the individual platform is minimized. Only the private key of the platform itself needs to be stored secretly inside each entity—in contrast to symmetric crypto systems where a single secret key needs to be stored inside several different entities. In context of PKC, if one entity is compromised, the others remain unaffected.

Besides encrypting or signing of messages, PKC can be employed to control user access to a device via electronic tokens. Examples for this are Remote Keyless Entry (RKE) systems [1] in the automotive domain or Hilti's TPS technology [2]. These systems incorporate contactless electronic tokens that substitute classical mechanical keys. The owner or authorized user identifies himself to the user device (`UD`) by possession of the token. `UD` and token are linked. Only if a linked token is presented to `UD`, it is enabled or access to `UD` is granted. In order to present a token to `UD`, information has to be exchanged between the two. The communication channel is usually assumed to be insecure. To prevent the usage of a device or its accessibility by an unauthorized person, the authentication has to be performed in a secure manner.

Authentication schemes based on Public Key Cryptography such as the Needham-Schroeder protocol [3], Okamoto-Protocol [4], and Schnorr-Protocol [5] provide authentication procedures where no confidential data is transmitted. Secret keys are stored in the tokens only and not in `UD`, thus omitting the need for costly security measures in the `UD`. Only public keys have to be introduced into `UD` (see Section 2), which can usually only be done by the manufacturer (`OEM`) of `UD`. In real-world operation, the introduction of public keys is

done in the field where UD is not necessarily under the control of OEM and a live online connection to OEM may not be possible. PKC is computationally very expensive, especially when aiming for high security levels. Dedicated hardware can provide the necessary speed up of cryptographic operations. With the decreasing cost of FPGAs, these devices are introduced more and more into embedded systems and mass market products. Therefore, hardware accelerators can be made available in these cost sensitive systems by adding cryptographic computation blocks on FPGA.

We propose a system to introduce public keys into FPGA based user devices to pair them with a new token. The proposed key flashing method allows authorization of the flashing process by OEM. Additionally it can be carried out with UD in the field and with no active online connection to OEM while flashing a key into UD. Introduction or flashing of new keys to an embedded device can be seen as a special case of a software update. Latter focuses on protection of the intellectual property, interoperability, correctness, robustness, and security. Recent approaches for the automotive area have been developed, for example, in the german HIS [6, 7] or the EAST-EEA [8] project. A general approach considering security and multiple software providers is given in [9]. Nevertheless, general update approaches are focused on the protection of IP and the provider against unauthorized copying and less on the case that the system has to be especially protected against unwanted updates as in our key flashing scenario.

The remainder of this paper is structured as follows. In Section 2, we present the basic application scenario followed by a short introduction to public key cryptography in Section 3. Section 4 describes a high-speed architecture for cryptographic computations. The requirements for the keyflashing scenario are described in Section 5. Based on this, we propose our flashing concept in Section 6, followed by the according requirements (Section 6.3). Section 7 details the flashing protocol with a live online connection available and Section 8 the protocol with no online dependability. Implementation details of the prototypical flashing framework are given in Section 9. We conclude with a security analysis and an outlook to future work in Sections 10 and 11.

2. Application Scenario: Automotive Access Control Systems

The target application focused on in this work is foremost automotive access control system. They comprise an entity that acts as the verifier (an ECU within the car) and an entity that acts as a prover (the traditional car key). Traditionally, a standard car key serves the sole purpose of identifying the current owner of the key as the authenticated user of the car (*authentication by ownership*). This also holds true for electronic car keys. As depicted in Figure 1, access to the car is granted by unlocking the doors only if the correct car key (prover) is presented to the car. The same procedure can be employed to disable or enable the immobilizer of the car, allowing the car's engine to start or not.

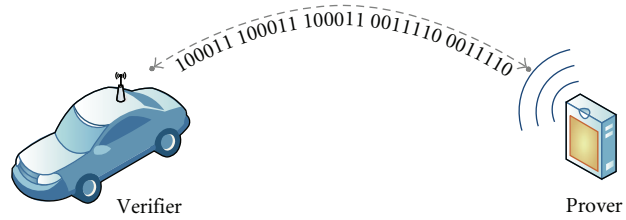


Figure 1: Access control: authentication scheme.

The automotive domain implies a very specific set of requirements. The industry is extremely cost driven, thus creating the need for very small hardware footprints. To comply with limited cost, OEMs tend to resort to cheap off-the-shelf components instead of specialized ASICs or complete systems-on-chip (SoC). Additionally a car's life cycle is about 10–15 years. Within this time span, all systems should work flawlessly.

Access control systems are a natural point of attack. Therefore, they need to offer very good security. To provide this, electronic car keys incorporate some kind of cryptographic algorithm. Raising security levels in this context can be achieved by adaption of the authentication protocol being used, enlarging key lengths, or substituting cryptographic primitives. All these measures tend to increase computation times. But all underlying computations and algorithms incorporated in access systems shall not be noticeable to the user of a car for best usability. Keeping the underlying hardware platform adaptable to varying interfaces and functionalities, it enables for integration of the same hardware components into a wide range of car keys for a multitude of different car models. With FPGAs dropping in cost over the last years, they also have been introduced more and more in cost driven industries such as the automotive domain. These devices are already being used in infotainment and multimedia devices. In addition to that, they can be used to provide dedicated hardware modules to accelerate cryptographic computations within user authentication in these systems. By using FPGA platforms for access control systems, they are adaptable over the lifetime of a car and offer some flexibility regarding changes in protocol and processing units.

In summary, we will regard the following application scenario: an access control system is applied to a mobile user device (UD); in our case, a vehicle is depicted in Figure 2. Through the access control system, the use of the UD can be restricted by allowing only the owner or authorized user access to the device. A transponder (TRK) serves as an electronic version of a mechanical key. TRK communicates to UD over a wireless communication channel. The user device accepts a limited number of transponders. If one of these is presented to the user device, it authenticates the transponder and the device is unlocked, thus granting access. Anyone possessing a valid TRK is considered an authorized user (OWN). This setup forms an authentication chain for usage of UD. An authorized user is authenticated through the possession of a valid TRK paired to the the UD. TRK in turn is authenticated by UD.

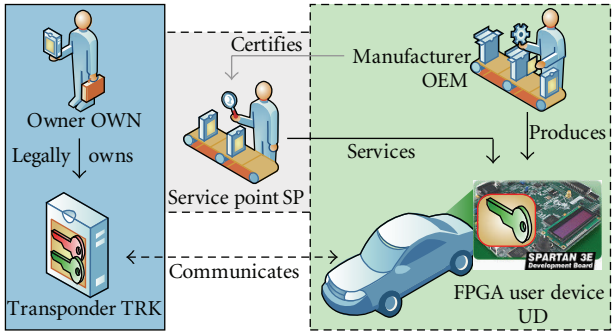


Figure 2: Entities and application scenario.

In automotive systems, authentication of a TRK can be achieved through a number of methods: rolling codes, symmetric codes, one-way-functions, or asymmetric codes. As analyzed in [10], there is a major disadvantage in using rolling codes and symmetric codes since secrets have to be stored within TRK as well as in UD, demanding for highly secure key management. One-way-functions such as cryptographic hash functions can circumvent this to some extent but demand for a substantial amount of secure storage. The most wide spread method for authentication in mobile devices is probably the usage of rolling codes (such as the KeeLoq [11] algorithm) due to easy implementation, followed by one-way-functions.

Asymmetric codes are very computationally expensive, although they provide extremely high security. With the advent of more and more computational power in embedded systems [12, 13], introducing such codes for user authentication is now feasible.

OEM is the manufacturer of UD. Due to the mobility of UD, it may be nowhere near OEM. Therefore, a service infrastructure has been established by OEM to repair, service, or replace a UD in the field. This infrastructure consists of a number of service points SP that are OEM certified. In the depicted example from the automotive domain, this would be a dealer or a car repair shop. SP is enabled by the OEM to carry out certain work on UD and acts in a way as a substitute for the OEM in the field.

In case of loss of a transponder, it is desirable to replace it, particularly if the user device itself is very costly or actually irreplaceable. Since the user device is mobile, linking a new transponder to a UD usually needs to be done in the field. This might include very secluded areas with minor to none communication infrastructure.

3. Basic PKC Functionalities

In 1976, Diffie and Hellman introduced the first PKC crypto system [14] for data encryption and confidential data transfer. Two different keys are used, one public (PK) and the other secret (SK). SK and corresponding PK are a fixed and unique keypair. It must be computational infeasible to deduce the secret key (SK) from the public key. With PK, a message M_p can be encrypted into M_c but not decrypted with the same key. This can only be done with knowledge of SK. If an entity Alice wants to transmit a message M_{Alice} to an entity Bob, it encrypts it with Bobs public key PK_{Bob}. Only Bob can retrieve the plain text from the encrypted message, by applying the appropriate decryption algorithm using his own secret key SK_{Bob}.

PKC can also be used to digitally sign a message. For this, a signature scheme is used that is usually different from the encryption scheme. When signing a message, the secret key is used and the signature can be verified using the according public key. In other words, if Bob wants to sign a message, he uses his own private key that is unique to him and solely known to himself. This key is used to sign a cryptographic hash value of the message M_{Bob}. The resulting value $\{\text{HASH}(M_{\text{Bob}})\}_{\text{sig}}$ is transmitted together with M_{Bob}. A receiver can validate the signature by using Bob's public key to retrieve $\text{HASH}(M_{\text{Bob}})$. From M_{Bob}, the receiver can compute the according hash value and compare it with the retrieved value. If both match, the signature has been validated. Since in the case of signature schemes the public key is often called verification key and the secret key is called signing key, we denote them accordingly VK and SK in the following.

4. Cryptographic Processing Entity

Computational efforts of cryptographic functionalities for PKC are very high and time consuming if carried out on today's standard platforms (i.e., microcontrollers) for embedded applications. Integrating security algorithms into FPGA platforms can provide high speed up of demanding PKC crypto systems such as *hyper elliptic curve cryptography* (HECC). By adding dedicated hardware modules for certain parts of a crypto algorithm, a substantial reduction of computation time can be achieved [15, 16].

In [16], an FPGA platform has been introduced which allows extremely fast authentication as proven by an experimental setup with two of these platforms. For this demonstrator, both platforms have been implemented on a Xilinx Spartan-3 XC3S2000 FPGA at 33 MHz. The communication channel in the setup is a wireless automotive transmitter [17] as is currently used in keyless go systems and is clocked with 412,5 kHz. The transceiver is connected to the FPGA system over SPI. Authentication of TRK via the Schnorr-protocol [5] in this setup lasts 120 ms including communication times over the wireless channel. To enable for even faster computation, we have developed a new, lean cryptographic core for Xilinx FPGA. It enables to carry out aforementioned mutual authentication within 82 ms.

Both platforms carry out calculations for public key cryptography based on hyper elliptic curves (HECC). They offer a higher security level than RSA while relying on relatively small key sizes of around 160 bit [18]. A detailed view on HECC and its underlying mathematics can be found in [19].

As shown in Figure 3, the automotive electronic control unit (ECU) comprises a MicroBlaze processor that handles arbitrary tasks necessary for running the car and is equipped with an appropriate interface such as CAN

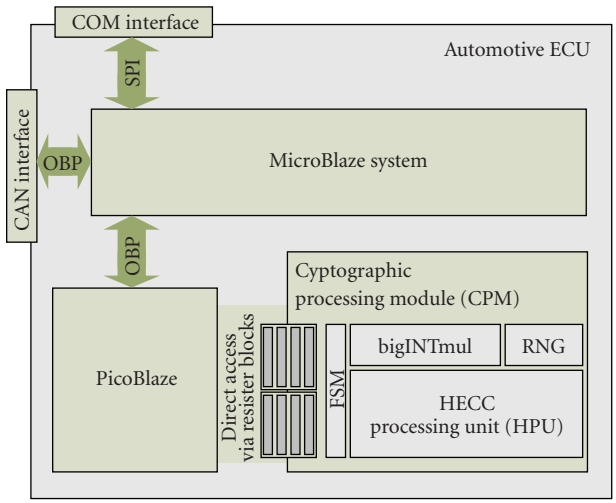


Figure 3: Overview of the system design.

to communicate with other ECUs residing in the vehicle. Additionally, a coprocessing unit composes of a PicoBlaze processor [20] and a *Cryptographic Processing Module* (CPM) is included. All cryptographic computations are done within this coprocessor. When no extensive tasks need to be run and only cryptographic functionality is needed, the coprocessor can also be run without the MicroBlaze. In this case, the PicoBlaze Controller is interfaced directly to the communication interface. This setup is very suitable for implementing a car key (`TRK`).

For HECC, three types of operations are essential (P_i denoting a point on a hyperelliptic curve and k, y, a, e, r are denoting integer values):

(i) calculation on a hyperelliptic curve ($k \cdot P$ and $P_1 + P_2$),

(ii) integer calculation with large operands ($y = a \cdot e + r$),

(iii) data exchange to/from the cryptographic unit.

Each arithmetic operation is assigned to a specialized hardware module to enable fast computation. At the same time, all cryptographic functionality is bound strictly to CPM, thus keeping all sensitive data on chip.

4.1. Cryptographic Processing Module. The *Cryptographic Processing Module* (CPM) is designed to efficiently compute $k \cdot P$ on a hyperelliptic curve, as well as integer multiplication with large operands. As depicted in Figure 4, the proposed architecture encompasses dedicated modules (HPU, bigINTmul) for these two operations and an additional module for generating random numbers (RNG).

A small finite state machine (FSM) is implemented for control flow of the calculations and to provide data exchange over the PicoBlaze processor. It controls all modules within the CPM directly. All arithmetic modules can work fully in parallel, allowing for concurrent operations within the protocol if necessary. A set of registers is provided for data exchange that can be accessed directly by PicoBlaze. Register e acts as input whereas y_i and the address space X of the CPM's internal memory *DataMem* are doubling as output registers. Some additional internal registers store cryptographic key material (Reg a_i) or random numbers (Reg r_i) and cannot be accessed from outside the CPM.

4.1.1. bigINTmul. A dedicated integer unit performing $y = a \cdot b + c$ on large operands is included in the CPM. Input to the multiplier are two operands a and b. The result $p = a \cdot b$ and operand c are then input to an adder stage calculating $y = p + c$. A sequential multiplier as depicted in Figure 5 is provided to execute a naive *shift&add* algorithm. p is accumulated by bitwise shifting of the bigger operand b, evaluating the least significant bit and adding a to the intermediate result if $b_0 = 1$, and then shifting p. If $b_0 = 0$, p is shifted without adding a.

In our use case, the two operands do not have the same bitlength since one input is the platforms secret key a_i and the other input is the challenge e.

4.1.2. HECC Processing Unit. The *HECC Processing Unit* (HPU) acts as a stand alone module for scalar multiplication $X_i = r_i \cdot P_i$ on a hyper elliptic curve. It comprises a dedicated arithmetic logical unit *dALU* for finite field arithmetic (*GF-operations*), internal memory *DataMem* for storage of intermediate values such as curve parameters and points P_i, and a control entity *HECC CTRL* connected to a program memory *pMEM*.

HECC CTRL in conjunction with *pMEM* implements the control flow of a dedicated algorithm for a scalar multiplication as a fixed sequence of GF-operations. The control flow is strongly optimized to execute a scalar multiplication in wMOF [21, 22]. To execute it, a highly specialized instruction set is implemented. An example of such an instruction is *shiftl_2* which shifts the content of the accumulator 2 bits to the left, an operation essential in wMOF [22]. The full instruction sequence of wMOF is stored in *pMEM*.

HPU is laid out as accumulator machine with harvard architecture. This enables to implement different data widths for *DataMem* and *pMEM* individually. This is particularly advantageous as we operate on galois fields GF(2^n), n being a big prime number, resulting in data words of n-bit length being stored in *DataMem*, while *pMEM* only stores minimal instruction codes.

Input to the HECC processing unit is a scalar r_i (bitlength of $r_i \leq l$) that is written into a dedicated register *rREG* of length l. Point P_i is a predetermined common point on a hyperelliptic curve and acts as a constant input decided upon during design time. Therefore, it is permanently stored in *DataMem* together with other curve parameters.

After storing r_1, HPU is triggered via the signal `control` to start a scalar multiplication. As soon as the result X representing the *commitment* is available, this is signaled by HPU over `trigger`. All r_i are random numbers generated by RNG. Therefore r_1 is loaded from the RNG module into HPU and $X = r_1 \cdot P_1$ is computed. Simultaneously, RNG generates a new random number r_2. After HPU signals the end of the current operation, r_2 can be loaded into *rREG* and HPU can calculate a new result $X = r_2 P_2$.

The dedicated arithmetic logical unit (*dALU*) can perform $u + v$, $u \cdot v \bmod p$ and u^2 with $u, v \in$ GF(2^n).

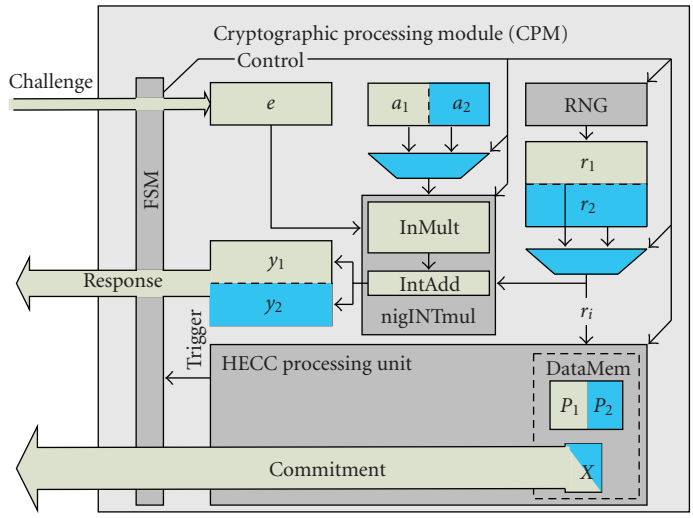


Figure 4: Cryptographic processing module (CPM).

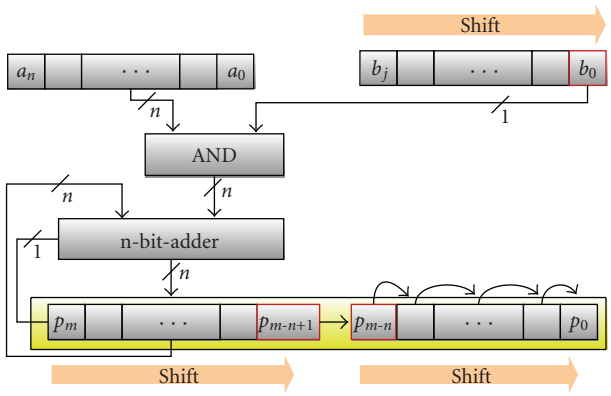


Figure 5: Sequential multiplier.

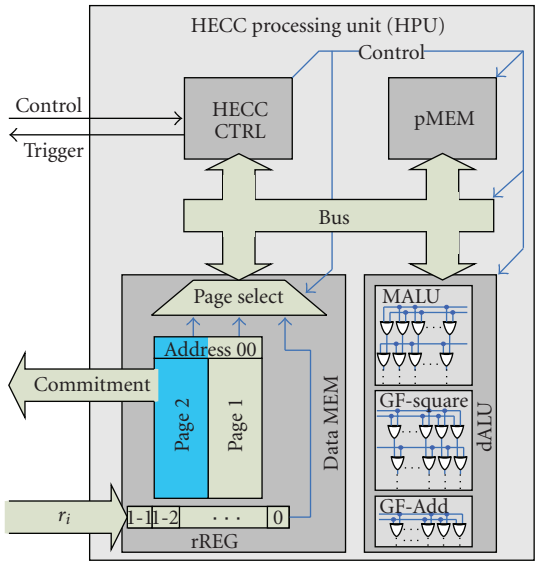


Figure 6: HECC processing unit.

On galois fields of genus 2, addition and subtraction can be implemented very efficiently by a bitwise *XOR* of the two operands u and v. Modular multiplication can be implemented as shown in Algorithm 1. It is performed as a *Shift&Add* algorithm with modular reduction in every step by adding the irreducible polynomial $P(x)$ if the intermediate result $T(x)$ exceeds the field size of GF(2^n). A single GF-multiplication $U \cdot V$ mod P will take n clock cycles since the size of the input word U is n-bit and U is examined bitwise (line 2–7 in Algorithm 1).

To speed up the multiplication a scalable GF-Multiplier (MALU) is integrated into the HPU. Instead of examining U a single bit at a time, a number d of neighboring bits in U are evaluated blockwise and in parallel. Algorithm 2 shows how d adder stages with modular reduction are fed with the operands U and V (line 3–6). This gives d individual intermediate results $T_d(x)$ which are then added and reduced to form the result $T(x)$ (line 6). This addition step is done over d cascading adders, so no additional clock cycle is necessary. The final result of a GF-multiplication is available after $\lfloor (n + d - 1)/d \rfloor$ clock cycles.

Speed up of the multiplication is achieved through the parallelization of additions, but hardware area increases as well. No additional instructions in *pMEM* are necessary, thus keeping this change in algorithm transparent to *HECC CTRL*.

Although u^2 can be calculated by feeding u into both inputs of *MALU*, a dedicated HW squarer requires 25% less computation time [23]. Because *HECADD* and *HECDBL* operations depend heavily on squaring [24], we included a GF-Square module in *dALU*.

4.2. Resource Usage and Timing Results. Table 1 gives a detailed view of the size of our system, and Table 2 shows the processing speed. To the best of our knowledge, no measurements of the execution time of Schnorr- and Okamoto-Authentication protocols based on HECC in the domain of

Requires: $U(x) = u_{n-1}x^{n-1} + \cdots + u_1x + u_0 \in \mathrm{GF}(2^n)$,
$V(x) = v_{n-1}x^{n-1} + \cdots + v_1x + u_0 \in \mathrm{GF}(2^n)$,
$P(x) = x^n + p_{n-1}x^{n-1} + \cdots + p_1x + p_0 \in \mathrm{GF}(2^{n+1})$
Ensure: $U(x)V(x) \bmod P(x)$
(1) $T(x) = t_nx^n + \cdots + t_1x + 0 \leftarrow 0$
(2) **for** $i = n - 1$ **to** 0 **do**
(3) $\quad m \leftarrow u_i \cdot V(x)$
(4) $\quad r \leftarrow t_n \cdot P(x)$
(5) $\quad T(x) \leftarrow (T(x) + m - r)x$
(6) $\quad i \leftarrow i - 1$
(7) **end for**
(8) **return** $T(x)/x$

Algorithm 1: GF multiplication of $U \cdot V \bmod P$ by adding n times.

Requires: $U(x) = u_{n-1}x^{n-1} + \cdots + u_1x + u_0 \in \mathrm{GF}(2^n)$,
$V(x) = v_{n-1}x^{n-1} + \cdots + v_1x + u_0 \in \mathrm{GF}(2^n)$,
$P(x) = x^n + p_{n-1}x^{n-1} + \cdots + p_1x + p_0 \in \mathrm{GF}(2^{n+1})$
Ensure: $U(x)V(x) \bmod P(x)$
(1) $T(x) = t_nx^n + \cdots + t_1x + 0 \leftarrow 0$
(2) **for** $i = n - 1$ **to** ≤ 0 **do**
(3) $\quad$ **for** $j = i$ **to** $i - d + 1$ **do**
(4) $\quad\quad m_j \leftarrow u_j \cdot V(x)$
(5) $\quad\quad r_j \leftarrow t_n \cdot P(x)$
(6) $\quad\quad T(x) \leftarrow (T(x) + m_j - r_j)x$
(7) $\quad$ **end for**
(8) $\quad i \leftarrow i - d$
(9) **end for**
(10) **returen** $T(x)/(x^{d-\lfloor (n+d-1) \bmod d \rfloor})$

Algorithm 2: GF multiplication of $U \cdot V \bmod P$ by adding $\lfloor (n + d - 1)/d \rfloor$ times.

embedded systems has been published. Because of this, we use the execution time of one scalar multiplication $k \cdot P$ as a benchmark in Table 3 to give a fair comparison of our architecture to other implementations.

As shown in Table 1, the deviation in size of the platform synthesized for executing a single scalar multiplication and the platform synthesized for execution two of this operation is less than 10%. The increase lies in the *Cryptographic Processing Module* (CPM) due to the increase in memory needed. An additional secret key a_2, as well as an additional random number r_2, needs to be stored, thus adding two registers. Also a 2nd memory page is needed, reflecting directly in the doubled number of BRAMs. These in turn require some additional slices for glue logic and enlarged multiplexers.

In a prototypical setup both the Schnorr- and Okamoto authentication protocols have been implemented. Table 2 shows that for both variants our architecture easily beats the real-time constraint of 180 ms, which is the average human reaction time [25].

When comparing our architecture with others (see Table 3), we compute a single $k \cdot P$ in roughly 8 ms. This is more than twice as fast as the platform no. 3 and no. 4. It also outperforms platform no. 1 which is one of the first full hardware implementations of a HECC scalar multiplication.

5. Pairing of Verifier/Prover Devices

With introduction of public-key-cryptography to automotive access control systems is advantageous for logistics. Less secret key material has to be handled. When integrating an authentication protocol such as Schnorr [5], no secret key resides in a vehicle (UD)—it only needs to be stored in the vehicle's transponder key token (TRK). Pairing of the two is done by storing a transponders public key in UD. In this paper, we mainly focus on this key flashing procedure for automotive entities and especially introduction of key material into UD during it's lifetime in the field.

Every UD has a number of public keys of transponders securely stored, thus establishing a "guest list" of legal TRKs. During production, at least two initial public keys of TRKs are written to the user device. This ensures that upon loss of one of the transponders, the remaining can be used to authenticate the owner. This initial operation certainly has to be secured against attacks to ensure that the "guest list" is not altered maliciously, otherwise illegal access to a UD might be granted.

As mentioned above, it is necessary to pair TRKs with a UD. This is achieved by flashing the public key of TRK into the user device, where the key is stored securely and is protected against unauthorized alteration. A number of initial TRKs are

Table 1: Resource usage.

		Okamoto				Schnorr			
		Slices	Slice FlipFlops	LUTs	BRAMs	Slices	Slice FlipFlops	LUTs	BRAMs
	Complete system	2651	2263	4323	8	2439	2026	3799	5
I	*PicoBlaze*	389	286	714	1	389	286	714	1
II	*CPM*	2275	1977	3585	0	2039	1740	3055	0
II-a	BigINTmul	265	392	201	0	265	392	201	0
II-b	HECC processing	1684	1082	2855	7	1682	1081	2759	4
II-b-i	*HECCCTRL*	61	32	108	0	61	32	108	0
II-b-ii	*dALU*	953	658	1611	0	953	658	1611	0
II-b-iii	*ProgramMem*	50	70	93	1	50	70	93	1
II-b-iv	*dbus*	98	0	162	0	98	0	162	0
II-b-v	*DataMem(singlepage)*	453	324	838	6	—	—	—	—
II-b-vi	*DataMem(doublepage)*	—	—	—	—	452	324	657	3

Table 2: Performance speed.

Operation	[μs]@50 MHz	
	Schnorr	Okamoto
Scalar multiplication $k \cdot P$		8 069
Complete protocol	81 621	132 858

paired during production in a secured environment by the OEM. Today pairing TRK to UD is done by introducing some kind of key material into UD, thus authorizing this token to use UD. Today's procedures either demand a live online connection to OEM or accept new TRKs if a "master" token (MTRK) is presented to the device [28, 29]. This means that such UD specific MTRK have to be stored very securely by SP and OEM has to fully entrust SP to do so. At the same time, there is no way to prevent TRKs to be flashed into a UD. Therefore, they have to be kept in secure, physical storage as well.

When employing asymmetric codes, these drawbacks are inexistent. Pairing procedures in this case depend mostly on a *trusted third party* (TTP) that generates key pairs and distributes them to the different entities. Because not only public key material is transferred but also secret key material this demands for fully encrypted end-to-end communication channels. Traditionally, this is done by establishing a mutual secret key between the two communication partners (i.e., via Diffie-Hellman key exchange [14]) and using symmetric ciphers to encrypt all data over the communication channel.

In our application scenario, we have the following main participants:

(i) a user device UD that may only be accessed or used by an authenticated user,

(ii) a human user OWN and he is authorized to access or use UD if he possesses a legitimate token,

(iii) a transponder key token TRK_{orig} originally linked to UD and a second token TRK_{new} that shall be flashed to UD additionally,

(iv) the manufacturer OEM that produces UD.

UD accepts a number of TRK to identify an authenticated user OWN of the UD. At least, two tokens are linked to a UD by storing the respective public keys VK_{TRK} inside the UD. The OEM is initially the only entity allowed to write public keys into any UD.

Solely, the public keys stored inside the UD shall be used for any authorization check of TRKs. The OEM's public key VK_{OEM} is stored in the UD as well.

OEM, TRK, and UD can communicate over any insecure medium, through defined communication interfaces.

5.1. Goals and Security Requirements. A new transponder TRK_{new} should be linked to UD to substitute an original token TRK_{orig} that has been lost or is defective. In the following, we will call the process of linking TRK_{new} to an UD *flashing*. Introduction of a TRK should be possible anytime in the complete life cycle of the UD. When flashing the UD it is probably nowhere near the OEM's location while introducing a TRK needs to be explicitly authorized by the OEM. Also should any TRK only be flashable into a single UD. Theft or unauthorized use of the UD resulting from improper pairing of a TRK needs to be prohibited. In addition, we demand that online connection of UD and OEM during the pairing procedure must not be imperative.

In summary, the protocol shall allow dependable authorized flashing under minimal assumptions while preventing unauthorized flashing reliably. Therefore, it has to guarantee the following properties, while assuming communication over an unsecured open channel.

(i) Correctness. In absence of an adversary, the protocol has to deliver the desired result, that is, after complete execution of the protocol, the flashing should be accomplished.

(ii) Authentication. The flashing should only be feasible if both OEM and OWN have been authenticated and have authorized the operation.

(iii) No online dependency. The protocol shall not rely on any live online connection to the OEM.

(iv) Secrecy. No confidential data like secret keys should be retrievable by an adversary.

Table 3: Duration of 1 scalar multiplication $k \cdot P$.

No.		Platform	Slices	f [MHz]	t [ms]	$t_{\text{normalized}}(f)$
1	Full Custom HW	[26]	$\gg$ 16000	45	20,2	1,03
2	8051 μC & CoPro	[18]	3781	12	2488	33,89
3	Microblaze & Coproc	[16]	1984	33	26,7	1
4	CrμP	[27]	1854 (+2379 for memory)	31,25	30,3	1,07
4	HECC processor	This work	2439	50	8,0693	0,46

5.2. Adversary Model. We assume an adversary $\mathcal{A}$ that is polynomially bounded in processing power and memory. $\mathcal{A}$ has access to all inter device communications, meaning he can eavesdrop, delete, delay, alter, replay, or insert any messages. We assume further that the adversary is attacking on software level without tampering with the participating devices. Without choosing particular instances of the cryptographic primitives, we assume that the signature scheme used is secure against existential forgery of signatures and regard the cryptographic hash function used as a random oracle.

6. Key Flashing Definitions and Requirements

The objective of the proposed key flashing protocol is the introduction of a public key $\mathtt{VK}_{\mathtt{TRK}}$ into UD. Main focus of it is the security aspect of the protocol itself, while it shall be usable under real world constraints as well. The protocol shall ensure the legitimacy of all entities involved as well as the security of the protocol itself to prevent misuse by a malicious attacker. Only after a correct, complete and successful flashing procedure, a new public key may be accepted and stored inside UD. If any error occurs during the flashing procedure, all previous steps in the protocol have to be revoked. All data resulting from these steps shall carry no information that can be exploited by an attacker.

Since TRKs gain their relevance only after successfully linking them to UD, they shall have no utility value before a successful key flashing procedure. This enables holding numerous TRKs in stock without an inherent need to restrict access to unflashed TRKs.

Two basic flashing scenarios are conceivable. One is that TRKs are flashed directly by the OEM, either during production or via an online connection as is addressed in Section 7. The second scenario is the flashing of TRKs through an authorized service point (SP) with no immediate online connection to the OEM (see Section 8).

6.1. Notations. For presentation of the protocols, we abstract from the specific algorithms and use abstract cryptographic primitives instead. Therefore, we introduce some assumptions, definitions, and notations.

Let H be a collision resistant cryptographic hash function of length k that maps any input of arbitrary bit length to an output of fixed bit length k. Application of H can be seen as taking a fingerprint of the input and is often used by signature systems. Nevertheless, our notion of a signature system given in Definition 1, abstracts from any implicitly used hash function.

Definition 1 (signature system). Let Σ_1, Σ_2 be finite alphabets. A *signature system SigSys* is a 7-tuple

$$SigSys = \left(\mathcal{M}, \mathcal{S}, \mathcal{SK}, \mathcal{VK}, f, \widetilde{S}, \widetilde{V}\right) \tag{1}$$

with

(1) $\mathcal{M}$ a nonempty set of messages $\emptyset \neq \mathcal{M} \subseteq \Sigma_1^*$ of arbitrary length over alphabet Σ_1,

(2) $\mathcal{S}$ a nonempty set of signatures $\mathcal{S} \subseteq \Sigma_2^*$,

(3) $\mathcal{SK}$ a nonempty set of signature keys,

(4) $\mathcal{VK}$ a nonempty set of verification keys,

(5) f a bijective function $f : \mathcal{SK} \to \mathcal{VK}$, mapping each signature key $\mathrm{SK} \in \mathcal{SK}$ on the respective verification key $f(\mathrm{SK}) = \mathrm{VK} \in \mathcal{VK}$, and we define a set $\mathcal{K} \subset \mathcal{SK} \times \mathcal{VK}$ of key pairs by $\mathcal{K} = \{(\mathrm{SK}, \mathrm{VK}) \in \mathcal{SK} \times \mathcal{VK} \mid f(\mathrm{SK}) = \mathrm{VK}\}$,

(6) $\widetilde{S} : \mathcal{M} \times \mathcal{SK} \to \mathcal{S}$ a signature function,

(7) $\widetilde{V} : \mathcal{M} \times \mathcal{S} \times \mathcal{VK} \to \{0, 1\}$ a verification function with the property, that for $\mathrm{SK} \in \mathcal{SK}, \mathrm{VK} \in \mathcal{VK}$, $M, M' \in \mathcal{M}$ holds:

$$\widetilde{V}\left(M, \widetilde{S}(M', \mathrm{SK}), \mathrm{VK}\right) = 1 \iff f(\mathrm{SK}) = \mathrm{VK},\ M' = M. \tag{2}$$

To ease readability and presentation, we introduce a shortened notation. We define the signed message

$$\mathtt{Sig}_{\mathrm{SK}}(M) = \left(M, \widetilde{S}(M, \mathrm{SK})\right) \tag{3}$$

as the message M together with the respective signature. Let $\mathcal{SM}$ be the set of all signed messages. In the following, we use an extended signature function

$$\begin{aligned} S :& \mathcal{M} \times \mathcal{SK} \longrightarrow \mathcal{SM}, \\ & (M, \mathrm{SK}) \longmapsto \mathtt{Sig}_{\mathrm{SK}}(M), \end{aligned} \tag{4}$$

and the respective extended verification function

$$\begin{aligned} V :& \mathcal{SM} \times \mathcal{VK} \longrightarrow \{0, 1\}, \\ & (\mathtt{Sig}_{\mathrm{SK}}(M), \mathrm{VK}) \longmapsto \widetilde{V}\left(M, \widetilde{S}(M, \mathrm{SK}), \mathrm{VK}\right). \end{aligned} \tag{5}$$

For a tuple (M, S'), where either the signature or the message is altered, we define

$$V((M, S'), \mathrm{VK}) = \widetilde{V}(M, S', \mathrm{VK}). \tag{6}$$

Furthermore, the tuple $(\mathrm{SK}_X, \mathrm{VK}_X) \in \mathcal{K}$ denotes the key pair of entity X.

6.2. Entities. In addition to the entities introduced in Section 5 (UD, OWN, TRK, and OEM), we use three additional participants, namely the transponder manufacturer TRKM, a service point SP and an employee SPE of this service point conducting the flashing procedure. In the following, an overview of the entities involved as well as their required properties and abilities is given.

6.2.1. OEM: Manufacturer. The OEM manufactures the UD and delivers it to OWN. OWN issued the corresponding TRKs linked to the UD. All UDs are obviously known to the OEM. Furthermore, TRKM and all SP and the respective public verification keys are known to the OEM. We regard the entity OEM as a trusted central server with database functionality. OEM can store data, sign data with $\mathtt{SK}_{\mathtt{OEM}}$, and send data. It possesses all cryptographic abilities for PKC based authentication schemes and can thereby authenticate communication partners.

6.2.2. TRK: Transponder. TRK possesses a keypair ($\mathtt{VK}_{\mathtt{TRK}}$, $\mathtt{SK}_{\mathtt{TRK}}$) for PKC functionality. It is generated inside TRK to ensure that the secret key $\mathtt{SK}_{\mathtt{TRK}}$ is known solely to TRK. Read access to $\mathtt{VK}_{\mathtt{TRK}}$ is granted to any entity over a communication interface. As TRKs can be manufactured by a supplier TRKM that has been certified by OEM, the $\mathtt{VK}_{\mathtt{TRK}}$ is signed by TRKM after generation and stored in TRK as a certificate. TRK possesses cryptographic primitives for PKC-based authentication schemes on prover's side and can thereby be authenticated by communication partners.

6.2.3. TRKM: Transponder Manufacturer. TRKM is a supplier for TRKs that has been certified by OEM and manufactures TRKs that fulfill OEM's requirements. Certification of TRKM is bound to the fulfillment of the conditions of manufacturing, defined by OEM and is enforced through appropriate legal contracting. TRKM possesses cryptographic abilities to sign the public key $\mathtt{VK}_{\mathtt{TRK}}$ of TRK. These signed keys act as certificates guaranteeing the origin of the respective TRK as well as the compliance of TRKM to all of OEM's manufacturing policies.

6.2.4. UD: User Device. UD is enabled only when a linked TRK is presented by authenticating the TRK via a PKC authentication scheme. All linked TRKs' public keys $\mathtt{VK}_{\mathtt{TRK}}$ are stored in UD. Additionally, the public key of the OEM $\mathtt{VK}_{\mathtt{OEM}}$ is stored in UD and cannot be erased or altered in any way. UD grants read access to all stored public keys. Write access to the memory location of $\mathtt{VK}_{\mathtt{TRK}}$ is only granted in the context of the proposed key flashing scheme. UD possesses all cryptographic abilities for PKC-based authentication schemes and can thereby authenticate communication partners.

6.2.5. OWN: Legal User. OWN is the legal user of UD and can prove this by possession of a linked $\mathtt{TRK}_{\mathrm{orig}}$.

6.2.6. SP: Service Point. SP is a service point in the field such as a wholesaler or workshop, certified by the OEM. For the protocol, SP is considered to be a computer terminal at the respective institution. The terminal and access to it is secured by appropriate means as in standard PC practice. SP can communicate to the OEM as well as to UD. In addition, it is able to read the $\mathtt{VK}_{\mathtt{TRK}}$ of any TRK.

Furthermore, SP constitutes a trusted platform meaning that it always behaves in the expected manner for the flashing procedure and accommodates a trusted module responsible for:

(i) storage of OEM-authorized keymaterial of TRKs,

(ii) key management of TRK keys,

(iii) secure counter.

SP possesses cryptographic primitives for PKC-based authentication schemes on prover's and verifier's side and can thereby be authenticated by communication partners, while it can also actively authenticate communication partners.

6.2.7. SPE: Employee of Service Point. SPE is a physical person that is operating SP and has to be authenticated prior to a flashing procedure to prevent misuse of the system. At the same time, SPE is regarded as a potential attacker of the flashing operation so that the protocol has to have a certain robustness against a compromised SPE. Access control of SPE to SP is enforced via password or similar. SPE is responsible for the system setup for the flashing application consisting of establishing the communication links of UD, SP, TRK, and OEM if needed.

UD, $\mathtt{TRK}_{\mathrm{new}}$, and SP are under control of SPE, and the communication links to UD, $\mathtt{TRK}_{\mathrm{orig}}$, $\mathtt{TRK}_{\mathrm{new}}$, SP, and OEM can be eavesdropped, but the trusted module cannot be penetrated.

6.3. Flashing Policies. In order to meet the goals defined in Section 5, some additional requirements must be met as follows:

(1) legitimation of the flashing procedure by OWN,

(2) legitimation of $\mathtt{TRK}_{\mathrm{new}}$ by a certified TRKM,

(3) only an OEM-authorized TRK may be flashed,

(4) any single TRK may only be flashed into a single UD.

Legitimation of a flashing procedure is achieved if the legal owner of OWN has commissioned and approved a flashing procedure. Legal ownership of a UD is proven by OWN through possession of a valid TRK that is already activated in UD. If no such TRK is available, legal documents proving the ownership have to be presented to OEM. Such a document could be a deed of ownership, for example. If OWN cannot prove his legal ownership, it is mandatory to prohibit flashing in order to prevent usage of an illegally acquired UD with an unauthorized TRK.

Legitimation of TRK is first achieved by adhering to all manufacturing policies posed by OEM. This is guaranteed by TRKM which in turn certifies the manufactured TRKs by signing their respective public keys $\mathtt{VK}_{\mathtt{TRK}}$. A main requirement for a legitimate TRK is its uniqueness and

accordingly the uniqueness of its cryptographic key material. Otherwise two identical TRKs, linked to two separate UDs would automatically be able to access both UDs. Additionally it has to be ensured and guaranteed that no secret key material of TRK is available to any entity other than to TRK itself.

Authorization of a legitimate TRK to be flashed is handled through OEM. OEM verifies the identity of each individual TRK to be flashed. Identification of a TRK can be achieved directly via checking VK_{TRK}. The identities are then stored by OEM. Only TRKs that have their identities checked and stored this way are considered to be authorized by OEM. Prove of authorization can be given by OEM through a signature $Sig_{OEM}(VK_{TRK})$.

7. Key Flashing Protocol without Mediator

The most direct flashing scenario is depicted in Figure 7, where the key flashing into a UD is done directly through the OEM. This scenario is valid during production of the UD or if the OWN is not able to legitimize the procedure through the possession of a second TRK as is needed in the standard flashing scenario as described later in Section 8. Therefore the legitimation of the flashing procedure is done implicitly through the OEM by checking the legal credentials of OWN (i.e., billing receipts, legal records, etc.). Only if sufficient proof of ownership is presented to the OEM, the flashing procedure is carried out.

The following entities are involved in the key flashing protocol:

(i) manufacturer OEM,

(ii) user device UD,

(iii) transponder TRK_{new}.

As shown in Algorithm 3, the direct flashing has two requirements. It is mandatory that UD has stored an immutable VK_{OEM}. This enables UD to verify the correctness of the OEM's signature later in the flashing protocol. A mandatory requirement for carrying out the flashing procedure is that the OEM has verified that the commissioning of the flashing procedure has been done by the legal owner of UD.

In a first step, OEM reads out the public key of the TRK to be flashed (TRK_{Bob}). The manufacturer TRKM of TRK_{Bob} has certified the key by signing it ($Sig_{SK_{TRKM}}(VK_{TRK_{new}})$). The OEM then checks if TRKM has fulfilled all legal obligations to be considered as a trusted manufacturer. If this is the case, the OEM checks if $VK_{TRK_{new}}$ is already stored in its internal database and if it has been already flashed to a UD or not. Only if $VK_{TRK_{new}}$ is a fresh key, it is stored in the OEM's database and the protocol is continued.

The second step consists of the OEM triggering the start of the flashing procedure. UD authenticates the OEM by means of an appropriate public key authentication protocol, referring to the internally stored VK_{OEM}. The protocol can only be passed successfully by the OEM since knowledge of the signing key SK_{OEM} is mandatory for the entity being authenticated. Subsequently, UD sends its self-signed verification key $Sig_{SK_{UD}}(VK_{UD})$ to the OEM.

The signature is then verified by the OEM in order to ensure that the transmitted verification key VK_{UD} has not been tampered with. Subsequently, the OEM binds $VK_{TRK_{new}}$ to VK_{UD} by composing an adequate data packet and signing it as a whole. This is then transmitted back to UD (Step 4 in Algorithm 3).

UD verifies the correctness of the packet by checking the OEM's signature in Step 5. After that, UD verifies the correct binding of data packet to its own identity by inspecting the VK'_{UD} including the data packet received. Only if the received VK'_{UD} is identical to his own verification key VK_{UD}, UD will accept $VK_{TRK_{new}}$ included in the received data packet as a valid, legitimate, and authorized transponder key to be flashed. UD stores $VK_{TRK_{new}}$ into internal protected memory and sends an acknowledge message back to the OEM. Storing $VK_{TRK_{new}}$ in UD turns TRK_{new} into an activated transponder $VK_{TRK_{orig}}$ linked to UD. OEM logs the successful conclusion of the protocol and annotates the $VK_{TRK_{new}}$ (now $VK_{TRK_{orig}}$) accordingly to exempt it from future flashing attempts.

8. Key Flashing Protocol with Mediator

The procedure as outlined in Section 7 demands a live online connection of UD to OEM. To fully comply with the flashing requirements introduced in Section 6, this online connection shall not be mandatory for the complete protocol. Therefore, an additional entity is introduced, a trusted Service Point SP that substitutes for the OEM in the field for introducing a new VK_{TRK} into a UD when no direct online connection to OEM is possible. The mediator and its properties are detailed in Section 8.1. The flashing procedure including a mediator comprises the following steps (see also Figure 8):

(i) delegation of trust to SP,

(ii) authorization of TRK_{new} by OEM,

(iii) introducing an authorized TRK_{new} into an UD.

The first two steps form an initialization phase to enable an SP to substitute for the OEM while flashing a new TRK into a UD. This two-step initialization will be detailed in Section 8.3. These steps form the first phase of the flashing process and can be done in advance without UD and OWN but need a communication link to OEM.

The last of the three steps, the actual flashing process of a new VK_{TRK} into an UD, do no longer depend on any direct interaction with OEM. Details will be given in Section 8.4.

8.1. Mediator. Because a mediator has to partly replace the OEM during the flashing protocol and UD only allows TRKs to be flashed through a trustworthy source—namely the OEM—the mediator has to be enabled to act as a trustworthy entity [30]. For this, the OEM has to delegate trust to a SP, in order to enable UD to entrust SP enough to accept a flashing request from it. It has to be ensured that the security of the overall flashing protocol is not weakened. Every mediator (SP) is evaluated by the OEM for its trustworthiness. Assessment factors can also include nontechnical aspects such as political and cultural environment, legal issues, or business models.

Requirements
(i) UD has knowledge of VK_{OEM}
(ii) OEM has ensured, that the legal owner OWN of UD has commissioned the flashing procedure
Protocol
Step 1: For a new TRK_{new} to be flashed OEM reads out the respective certified verification key $Sig_{SK_{TRKM}}(VK_{TRK_{new}})$ and verifies that VK_{TRKM} is in the internal database of trusted transponder manufacturers.
Step 2: OEM contacts UD and is authenticated using a PKC authentication protocol. UD sends $Sig_{SK_{UD}}(VK_{UD})$ to OEM.
Step 3: OEM verifies that $V(Sig_{SK_{UD}}(VK_{UD}), VK_{UD}) \stackrel{!}{=} 1$
Step 4: OEM sends $Sig_{SK_{OEM}}(VK_{TRK_{new}}, VK_{UD})$ to UD.
Step 5: UD verifies that $V(Sig_{SK_{OEM}}(VK_{TRK_{new}}, VK'_{UD}), VK_{OEM}) \stackrel{!}{=} 1$.
Step 6: UD verifies that $VK'_{UD} \stackrel{!}{=} VK_{UD}$. Then the new transponder TRK_{new} can be activated. The protocol is completed by sending a DONE-message to OEM.

Algorithm 3: Direct flashing protocol (with online connection to OEM).

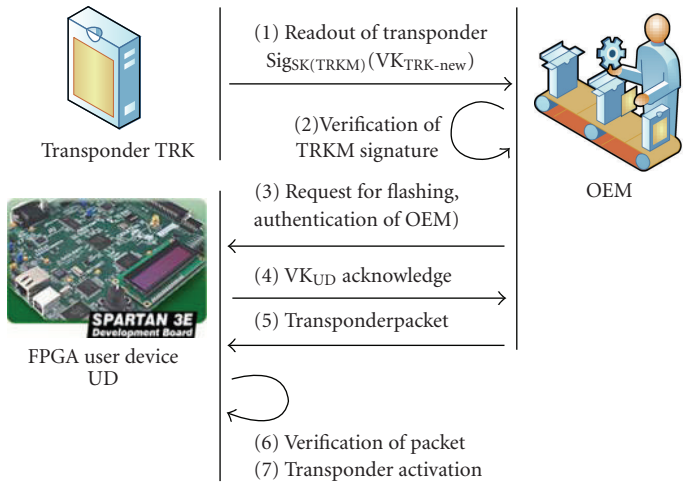


Figure 7: Key flashing without mediator.

Based on this evaluation, trust credentials (see Section 8.2) are issued to each individual SP.

To restrict access to the flashing capability of SP, as might be necessary in order to comply with the flashing policies of OEM a separate authentication of employees (SPE) working at SP is suggested. Such authorization of SPE can be done, for example, via a password (knowledge) or by biometrical identification (physical property).

8.2. Service Point as Trusted Platform. An SP constitutes a trusted platform as defined in [30] meaning that SP always acts as specified at any point in the protocol. At the same time, it needs to act reliably in order to enforce trust policies. Typically, an SP might reside in a hostile environment and can be accessible to malicious attackers. Therefore, some minimal functionalities of SP must be inherently secure and are encapsulated in a *Trusted Zone* (see Figure 9) as follows:

(i) generation of trust key pairs,

(ii) storage of private keys (SK_{SP}^{TD} and SK_{SP}),

(iii) signature generation,

(iv) enforcement of Trust Policy.

For all key pairs that are generated to be used as temporary trust keys, it has to be ensured that a SK_{SP}^{TD} is never communicated to another entity. Also, it has to be ensured that SK_{SP}^{TD} cannot be deducted from VK_{SP}^{TD}. This can be ensured with a proper key generation algorithm. Signing of messages has to be secure in order to prevent manipulation of signed data packets. This means that any signing operation is always done with the proper signing key residing in SP.

The main point of the trusted platform is the enforcement of trust policies. SP is issued a temporal trust key pair $(SK_{SP}^{TD}, VK_{SP}^{TD})$ as will be described in Section 8.3. This key pair expires after a timepoint T or after a certain amount of flashing procedures N. Expiration is enforced from within the *Trusted Zone* with a secure unforgeable counter that is keeping track of the number of flashing cycles. As soon as the counter value reaches N, the trust key pair is fully deleted and the counter is reset. SP also compares its system

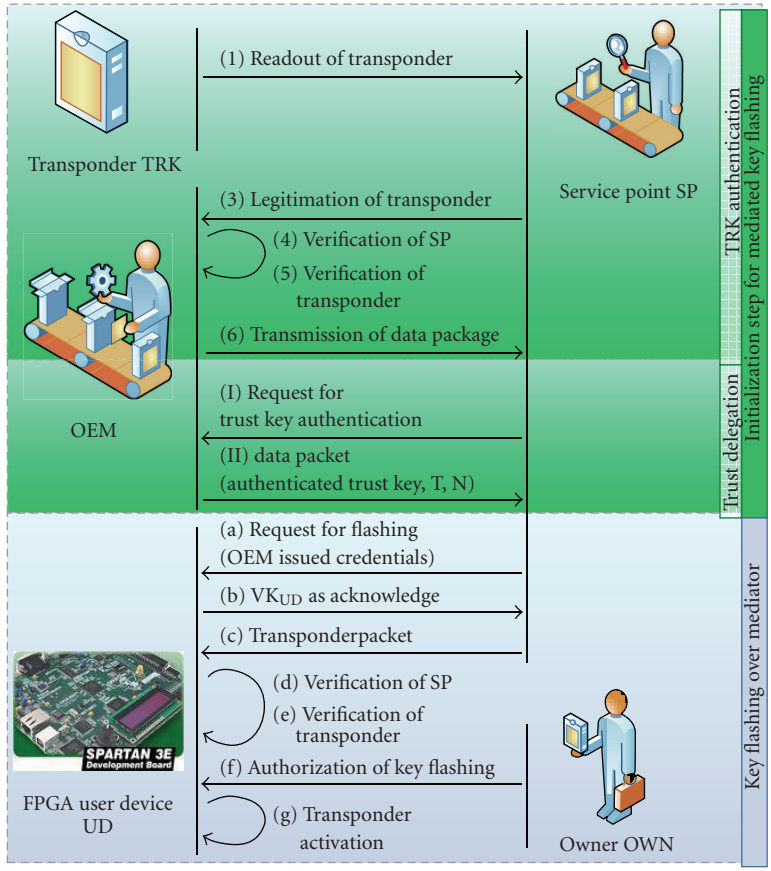


Figure 8: Flashing scheme.

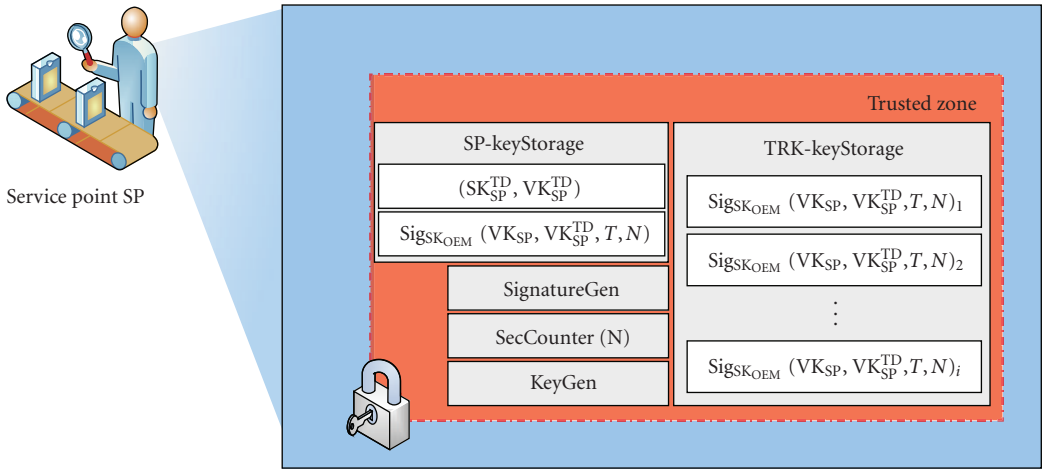


Figure 9: Service point as trusted platform.

time T_{SP} to T. If $T_{SP} \geq T$ the trust key pair is also fully deleted.

8.3. Trust Delegation and TRK_{new} Authorization. To be able to perform a key flashing procedure without an active link to OEM, a local representative has to be empowered by the OEM to perform the flashing, assuming that UD trusts only the OEM to flash legit keys. This is done by presenting a credential to UD accounting that flashing is authorized by OEM. The exchange of this credential is denoted in the following as *trust delegation*.

Algorithm 4 shows the protocol for instantiating a mediator that can flash a TRK into a UD. Steps I.1 to I.4 detail the trust delegation to SP. First of all, SPE is authenticated by SP

Requirements

(i) OEM has knowledge of VK_{SP} and VK_{TRKM}

Protocol

Step I.1: SPE presents his credential $CRED_{SPE}$ and SP authenticates SPE. After that SP is activated and communication to OEM is enabled.

Step I.2: SP creates a new key pair $(SK_{SP}^{TD}, VK_{SP}^{TD})$ and sends its ID together with the created verification key VK_{SP}^{TD} as a signed request $[Sig_{SK_{SP}}(VK_{SP}^{TD}), VK_{SP}]$ for a trust credential to OEM.

Step I.3: OEM verifies that SP and the respective verification key VK_{SP} is listed in the internal database of trusted mediators and that $V(Sig_{SK_{SP}}(VK_{SP}^{TD}), VK_{SP}) \stackrel{!}{=} 1$. In this case OEM creates a trust delegation credential $Sig_{SK_{OEM}}(VK_{SP}^{TD}, VK_{SP}, T, N)$ bound to SP with timestamp T and number of granted transactions N and sends it to SP.

Step I.4: SP receives $Sig_{SK_{OEM}}(VK_{SP}, VK_{SP}^{TD}, T, N)$ and stores it in the trusted storage. This step completes the trust delegation for flashing.

Step II.1: For a number of TRK_{new} to be flashed, SP reads out the respective certified verification keys $Sig_{SK_{TRKM}}(VK_{TRK_{new}})$ and sends $Sig_{SK_{SP}}(Sig_{SK_{TRKM}}(VK_{TRK_{new}}))$ to OEM.

Step II.2: OEM verifies that VK_{SP} and VK_{TRKM} are in the internal database of trusted peers and that $V(Sig_{SK_{SP}}(Sig_{SK_{TRKM}}(VK_{TRK_{new}})), VK_{SP}) \stackrel{!}{=} 1$ and $V(Sig_{SK_{TRKM}}(VK_{TRK_{new}}), VK_{TRKM}) \stackrel{!}{=} 1$. Afterwards OEM creates $Sig_{SK_{OEM}}(VK_{TRK_{new}}, VK_{SP})$ and sends it to SP.

Step II.3: SP receives $Sig_{SK_{OEM}}(VK_{TRK_{new}}, VK_{SP})$ and stores it in the trusted storage. This step completes the activation of the transponder TRK_{new} for flashing over SP.

Algorithm 4: Initialization step for mediated key flashing.

to prevent SP abusive operations. Afterwards, SP can connect to OEM and request a trust credential (Step I.1).

A trust credential consists of a cryptographic temporal trust key pair $(SK_{SP}^{TD}, VK_{SP}^{TD})$ with the public key VK_{SP}^{TD} being signed by OEM. Therefore, SP creates a fresh trust key pair. From this pair, SP sends the fresh verification key VK_{SP}^{TD} as well as its ID as a signed data packet to the OEM, while the secret key SK_{SP}^{TD} never leaves SP. The ID is the standard verification key VK_{SP} of the service point (see Algorithm 4 Step I.2). For ease of implementation, this can be replaced with a unique identifier that can then be matched by the OEM to the appropriate VK_{SP} stored in an internal database.

Using VK_{SP} OEM verifies the correctness of the data packet received. If SP is considered a trustworthy entity, that is, if it adheres to all of OEMs policies, OEM issues a trust credential. As shown in Step I.3 (Algorithm 4) it comprises the verification key of the trust key pair, the standard verification key (VK_{SP}) of SP, as well as a timestamp T, and a maximum transaction number N in an OEM signed packet. Through inclusion of VK_{SP}, OEM binds the trust key VK_{SP}^{TD} to SP. Later, this binding will be verified by UD (see Section 8.4). A trust key pair is only valid for a limited time and limited number of flashing operations after which it is deleted by SP. Step I.4 concludes the trust delegation phase with SP storing the OEM-signed packet in trusted storage for later use in the actual flashing procedure (Section 8.4).

In order to flash a TRK_{new}, the transponder needs to be authorized by OEM. The second part (Steps II.1–Step II.3) of the protocol shown in Algorithm 4 accomplishes this for a single TRK_{new}. If more than one TRK shall be set up for flashing, this part of the protocol is rerun for each additional TRK_{new}. SP reads out the verification key of TRK that previously has been certified and signed by TRKM. SP send this key as a self-signed message as shown in Step II.1 to OEM for authorization. In Step II.2, OEM verifies both signatures and ensures that TRKM as well as SP are trusted peers. If this is the case, OEM forms a data packet comprising the public key of TRK_{new} as well as the standard public key of SP and sends it as a signed message to SP. This message effectively binds TRK_{new} to SP, thus enabling solely SP to flash TRK_{new} into UD later on. The second part of the protocol in Algorithm 4 is finalized in Step II.3 by SP storing the received, signed message in its trusted module.

Only a limited number of authorized TRKs can be stored at any given point in time. As soon as a TRK has been authorized by the OEM, physical access to the TRK needs to be controlled. The authorization process of TRKs is the only step that demands a data connection between SP and OEM. This does not necessarily need to be an online connection since data could also be transported via data carriers such as CDs, memory sticks, or the like.

8.4. Flashing of TRK. The actual flashing of a TRK_{new} to a given UD is shown in Algorithm 5. It demands a valid new transponder TRK_{new} and authorization by OEM and OWN. Former either directly or delegated to SP using the credential introduced above, latter done by presenting a valid and linked TRK_{orig} assumed to be solely accessible by OWN. If an online connection to OEM is available, the protocol can be performed by UD and OEM directly as described in Section 7, with SP only relaying communication.

If SP has to act as an offline mediator, the initialization protocol (Algorithm 4) has had to be successfully completed. From there on, the flashing protocol commences as shown in Algorithm 5 with SP contacting UD and sending the trust credentials that SP has received from OEM (Step 1). In Step 2,

Requirements

(i) The initialization protocol has been completed successfully.

(ii) UD has knowledge of $\mathrm{VK}_{\mathtt{OEM}}$.

(iii) SP has a valid trust key pair and has not reached the maximum quota N of allowed flashing procedures.

Protocol

Step 1: SP contacts UD and sends $\mathtt{Sig}_{\mathrm{SK}_{\mathtt{OEM}}}(\mathrm{VK}_{\mathtt{SP}}, \mathrm{VK}_{\mathtt{SP}}^{\mathrm{TD}}, T, N)$ to UD.

Step 2: UD verifies $V(\mathtt{Sig}_{\mathrm{SK}_{\mathtt{OEM}}}(\mathrm{VK}_{\mathtt{SP}}, \mathrm{VK}_{\mathtt{SP}}^{\mathrm{TD}}, T, N), \mathrm{VK}_{\mathtt{OEM}}) \stackrel{!}{=} 1$ and sends back $\mathtt{Sig}_{\mathrm{SK}_{\mathtt{UD}}}(\mathrm{VK}_{\mathtt{UD}}, \mathrm{VK}_{\mathtt{SP}})$ as an acknowledge.

Step 3: OWN authorizes the start of a key flashing procedure by presenting a valid $\mathtt{TRK}_{\mathrm{orig}}$. UD authenticates $\mathtt{TRK}_{\mathrm{orig}}$ using the internally stored $\mathrm{VK}_{\mathtt{TRK}_{\mathrm{orig}}}$ and a PKC authentication protocol.

Step 4: SP sends the certified new key package $\mathtt{Sig}_{\mathrm{SK}_{\mathtt{SP}}^{\mathrm{TD}}}(\mathtt{Sig}_{\mathrm{SK}_{\mathtt{OEM}}}(\mathrm{VK}_{\mathtt{TRK}_{\mathrm{new}}}, \mathrm{VK}_{\mathtt{SP}}), \mathrm{VK}_{\mathtt{UD}})$ to UD.

Step 5: UD verifies that $V(\mathtt{Sig}_{\mathrm{SK}_{\mathtt{SP}}^{\mathrm{TD}}}(\mathrm{S}_{\mathrm{SK}_{\mathtt{OEM}}}(\mathrm{VK}_{\mathtt{TRK}_{\mathrm{new}}}, \mathrm{VK}_{\mathtt{SP}}), \mathrm{VK}_{\mathtt{UD}}), \mathrm{VK}_{\mathtt{SP}}^{\mathrm{TD}}) \stackrel{!}{=} 1$ and $V(\mathtt{Sig}_{\mathrm{SK}_{\mathtt{OEM}}}(\mathrm{VK}_{\mathtt{TRK}_{\mathrm{new}}}, \mathrm{VK}_{\mathtt{SP}}), \mathrm{VK}_{\mathtt{UD}}, \mathrm{VK}_{\mathtt{OEM}}) \stackrel{!}{=} 1$. Then the new transponder $\mathtt{TRK}_{\mathrm{new}}$ can be activated: $\mathrm{TRK}_{\mathrm{new}} \mapsto \mathrm{TRK}_{\mathrm{orig}}$. The protocol is completed by sending a DONE-message to SP.

Algorithm 5: Indirect flashing protocol over a trusted mediator (no online connection to OEM).

UD verifies these credentials using the OEM's verification key that is already embedded in UD (Section 6.2.4). The verification key of the trusted key pair $\mathrm{VK}_{\mathtt{SP}}^{\mathrm{TD}}$ is stored temporarily for use in Step 6. As an acknowledge message, UD sends a self-signed packet to SP that includes its own public key as well as the standard public key of SP.

Since OWN has to authorize the flashing procedure he presents his credential in form of an original $\mathtt{TRK}_{\mathrm{orig}}$ already linked to UD. UD authenticates $\mathtt{TRK}_{\mathrm{orig}}$ by means of a public key authentication protocol (Step 3). Only if $\mathtt{TRK}_{\mathrm{orig}}$ has been successfully authenticated UD will accept a $\mathtt{TRK}_{\mathrm{new}}$ being flashed into UD. In Step 4, SP sends the authorized data packet for the TRK to be flashed that it has received from the OEM during the second phase of the initialization procedure (Section 8.3). It is annotated with UD's public key $\mathrm{VK}_{\mathtt{UD}}$ and additionally signed by SP using the trust key pair. That way, the $\mathtt{TRK}_{\mathrm{new}}$ is bound to UD. To finalize the protocol, UD enforces the flashing policies in Step 5. First it verifies if the signature of SP is correct and has used the trust key pair. If that is the case, it verifies the correctness of the OEM's signature on the data packet for $\mathtt{TRK}_{\mathrm{new}}$. Since this data packet has been bound to UD, UD verifies if its own public key has been used to annotate $\mathrm{VK}_{\mathtt{TRK}_{\mathrm{new}}}$. Only a correctly annotated $\mathrm{VK}_{\mathtt{TRK}_{\mathrm{new}}}$ will be accepted, all others are dismissed and will not be stored into UD.

In the case of successful verification, UD accepts the new token $\mathtt{TRK}_{\mathrm{new}}$ and adds $\mathrm{VK}_{\mathtt{TRK}_{\mathrm{new}}}$ to its internal list of linked tokens thus transforming $\mathtt{TRK}_{\mathrm{new}}$ into a $\mathtt{TRK}_{\mathrm{orig}}$. The correct and full flashing is then reported back to SP. Subsequently, SP will log this as a successful flashing procedure and decrement its internal counter for allowed flashing processes.

8.5. Entity Requirements. Regarding the proposed flashing protocols certain requirements for the entities' functionalities have to be satisfied. An overview is given in Table 4. Data management is one of the key requirements in the protocol in the sense that public key data needs to be stored. Secure storage for delegated trust has some additional requirements such as intrusion detection to protect data from being altered in any way. At the same time, it is mandatory that this data is always changed correctly as demanded by the protocol. Also, the OEM's public key needs to be firmly embedded into the entities and must not be altered in any way. Otherwise, the OEM cannot be identified correctly within the proposed protocols.

Table 4: Entity requirements.

	OEM	SP	UD	TRK
Initiate communication	•	•		
Acknowledge communication	•		•	
Generation of keypairs	•	•		•
Signature generation	•	•	•	
Signature verification	•	•	•	
Datamanagement for suppliers	•			
Datamanagement for user devices	•			
Datamanagement for service points	•			
Datamanagement for TRKs		•	•	
Secure storage for delegated trust		•		
Knowledge of OEM's public key		•	•	

9. Implementation

The protocol has been implemented as a proof of concept in a prototypical setup based on a network of standard PCs representing OEM and SP (see Figure 10). Furthermore, Digilent *Spartan3E Starter Boards* with a Xilinx XC3S500 FPGA represent TRKs and UDs. TRK, SP, and UD have to be connected when flashing the key. The OEM connection needs to be established anytime prior to the flashing according to the proposed protocol and is connected via TCP/IP to the SP.

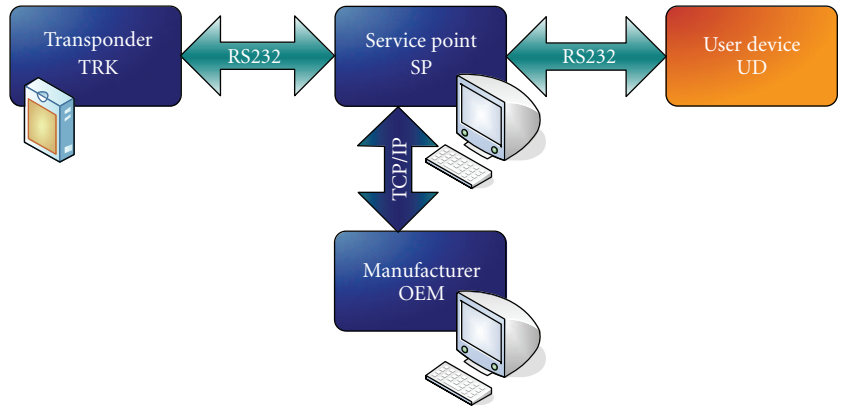


Figure 10: Component interaction.

Table 5: Parameters for RSA-System.

Key length	1024 Bit
Exponent	$2^{16} + 1$ (65537)
Padding scheme	PKCS#1 v1.5
Signature scheme	PKCS#1 v1.5
Hashing scheme used for signing	SHA1

All other communication is done over RS232 interfaces that are available both on PC and the FPGA boards. These can be substituted for other communication structures if needed, that is, wireless transmitters.

9.1. Choice of Cryptographic Primitives. The proposed key flashing concept demands asymmetric encryption and a cryptographic hash function. RSA [31] is chosen for encryption and signing, SHA1 [32] for hash functionality. Both schemes are today's standard and have not been broken yet, but can be substituted in our implementation for more secure schemes such as HECC if needed. RSA as well as SHA1 implementations are freely available as software and hardware modules for numerous platforms. RSA parameters used in the prototype are given in Table 5.

All signatures in our context are SHA1-hash values of data that has been encrypted according to the signing scheme PKCS#1 v1.5 [33]. Such a signature has a length of 128 Byte when using a key length of 1024 bit and hash values of 160 bit length.

9.2. OEM/Service Point-Software Platform. Both components `OEM` and `SP` have been implemented on a standard PC in software under the .NET framework version 2.0 [34] using C#. The .NET framework provides the Berkeley Socket-interface for communication over the PC's serial interface. It also includes the `Cryptography`-namespace providing all needed cryptographic primitives including hash functions and a random number generator that are based on the FIPS-140-1 [35] certified Windows CryptoAPI. The software is modularized to enable easy exchange of functional blocks and seamless replacement of algorithms. Software modules communicate only over defined interfaces to enable full functional encapsulation. For ease of usage, a graphical user interface (GUI) is included as well in both entities.

9.3. Transponder/UserDevice—FPGA platform. The targeted user device is an FPGA. To ease reuse of functionalities the exemplary `TRK` has been implemented on FPGA as well, but can also be integrated into a smart card or RFID chip as long as the appropriate cryptographic primitives are provided.

In the prototypical setup, we used a MicroBlaze-based ECU (see Figure 3) for both `UD` and `TRK`. We omitted the coprocessor and implemented all functionality on the MicroBlaze including cryptographic functions. Hardware peripherals such as an LCD controller have been integrated for debugging purposes. To enable handling of big numbers, as are used in the cryptographic functions of the protocol, the libraries `libtommath` [36] and `libtomcrypt` [37] are used. Only necessary components have been extracted from those libraries and are integrated into `TRK` and `UD`.

9.4. Resource Usage. The resource usage of the components `OEM` and `SP` are very similar, since almost identical functional software blocks are used in both. Table 6 gives an exemplary overview of the lines of code of the `OEM` implementation. The memory footprint of the compiled `OEM` implementation is 129 KB (139 KB for the `SP` implementation). At start up, 15400 KB of main memory is used. The execution times for RSA- and SHA1-operations were measured on a PC (2 GHz, 1024 MB RAM) and are all in the range of milliseconds.

Resource usage of the FPGA-based components `UD` and `TRK` are given in Table 7. By implementing all functionality on a MicroBlaze softcore, the hardware usage is quite moderate. On the other hand, the software footprint is 295 KB for the `UD` implementation, due to the nonoptimized memory usage of the crypto library.

Shown in Table 8 are the execution times of the diverse protocol instances. The duration of parts of the protocol that are based solely on `OEM` and `SP` is in the area of few milliseconds. As soon as mobile devices (`UD`, `TRK`) process parts of the protocol, speed is declining since all crypto operations are currently carried out on an embedded

Table 6: Properties of OEM component.

Module	Lines of code	Percentage
Main application	1234	41.77
GUI	264	8.94
Cryptography	385	13.03
Interaction	383	12.97
Communication	545	18.45
Data management	143	4.84
Total	2954	100

Table 7: FPGA resources.

Slices	1.791 of 4.656 (38%)
Slices: FlipFlops uses	1.590 of 9.312 (17%)
Slices: LUTs used	1.941 of 9.312 (20%)
BlockRAMs used	16 of 20 (80%)
Equivalent logic cells	1.135.468
Minimal clock period	18,777 ns
Maximum clock frequency	53,257 MHz

Table 8: Protocol execution times.

Protocol instance	Duration (min:sec.ms)
ReadOut of transponder	01:32.000
Mutual authentication of UD and TRK	03:14.000
Direct keyflashing	
Keyflashing to transponder by OEM	23:50.000
Keyflashing by servicePoint	
Delegation of trust OEM to SP	00:00.350
Transponderdelegation	00:00.250
Keyflashing to transponder by SP	12:43.000

microcontroller. Main factor here is the RSA decryption operation. With appropriate hardware support, choice of parameters, cryptosystem, and substantial speedups can be achieved as shown in [16].

10. Security Analysis

Looking at the security of the proposed concept some points can be identified where security relies on policies and implementing rules while other issues are covered by design.

Using PKC primitives and trusted computing approaches, the protocol ensures confidentiality of secret keys and mutual authentication of SP and OEM, OWN and UD, SP and UD, SP, and SPE. Due to the necessity of online independence, there are some assumptions that have to be made to guarantee security. This is mainly the trustworthiness of the SP in combination with the physical protection of any authorized TRK_{new} and all TRK_{orig}.

If these assumptions are broken, for example, by theft of authorized TRK, the corresponding SP and the SPE password, unauthorized flashing may be possible. As countermeasures, the usage of the protocol can be adapted to dilute effects of such events. So, the number of allowed authorized TRK should be as low as possible and the SP should be implemented using trusted components and based on a trusted platform. Secrets should be especially protected against misuse by a physical attacker.

There certainly is a tradeoff between security and usability of the flashing scheme, since the protocol has been designed for real-world implementation.

10.1. Security of Direct Flashing Scenario. In the flashing scenario with no mediator (Section 7), an illegal flashing of TRK is not possible. The flashing procedure is authorized through the OEM directly. Only the OEM is considered trustworthy enough to accept flashing commands from. By verifying the signature on a TRK's key, it can be checked if a certain TRK has been manufactured by a certified supplier TRKM or not. Certification policy ensures that such a TRK has a unique ID, unique cryptographic keys, and secret key material is solely known to the TRK itself and is nowhere else available or reproducible.

Verifying the signature of UD and the mutual authentication phase of OEM and UD, OEM can be sure that a UD is targeted in the flashing procedure that has been manufactured by OEM. In turn, UD can be sure that its communication partner is the OEM.

Binding the key material to be flashed to a dedicated UD by incorporating a mutual signature of the key material enforces that a certain transponder is flashed only into a single UD. Also, it can be enforced that the packet containing the VK_{TRK} is only used for a single flashing procedure, thus countermeasuring replay attacks. Neither unrecognized mutation of dedicated parts of this communication packet is possible, nor forging the signatures on data, due to the security assumptions of the cryptographic primitives. No confidential information is included in any communication packet. By activating the VK_{TRK} inside the UD only after a successful transmission without errors, it is ensured that the TRK has no previous utility value. Readingout the TRK's VK_{TRK} has no benefit to an attacker. Therefore, a TRK has not to be stored away safely before linking it to a UD. Loss of an unlinked TRK does also not lead to any security issues.

If all entities involved in the flashing procedure adhere to the protocol, abolish all data resulting from a disrupted flashing procedure, and implement all cryptographic primitives securely, an attacker is not able to carry out an illegal flashing procedure.

10.2. Security of Mediator Flashing Scenario. In the flashing scenario involving a mediator as described in Section 8, the flashing procedure is legitimated through OWN directly by presenting a second TRK that is already linked to UD during the last phase of the flashing process. A UD receiving the final communication packet carrying the VK_{TRK} to be flashed can verify if the sender of the packet is legit and trustworthy by checking the signatures of the OEM as well as checking the trust credentials of the SP. Therefore, the UD directly enforces the policy that only certified parties may flash a VK_{TRK} by dismissing any received VK_{TRK} as soon as a invalid signature is detected.

Usage of SP through a SPE is restricted and protected by appropriate access control, so no malicious outsider can flash a VK_{TRK}. Since trust has to be redelegated after a certain time or amount of flashing procedures, it is not possible to haphazardly flash TRKs.

No sensitive data is transmitted during the flashing protocol that might compromise the system's security. It seems that the data packet sent by OEM to SP might be highly sensitive, but even if an attacker were able to access the keypair forming the delegation of trust, it would not be possible to authenticate new TRK's with the OEM since this demands knowledge of SK_{SP} known only to SP itself. TRK's already authenticated by the OEM can also not be flashed into a UD, since it will be impossible for an attacker to correctly sign the data packet containing the TRK's public key, because the signing key SK_{SP}^{TD} is also known solely by SP.

In this second flashing scenario, it is again ensured that the TRK has no previous utility value before finally linking it to a UD in the final step of the protocol. The loss of of an unlinked TRK does not lead to any security issues as long as it has not been authorized by the OEM. Therefore all unauthorized TRK do not pose a security risk. Authorized TRKs do need to be stored away securely since they can be flashed into a UD, but only with the SP that is linked to the TRK. As before, no attacker may be able to carry out an illegal flashing procedure as long as all entities involved in the flashing procedure adhere to the protocol, abolish all data resulting from a disrupted flashing procedure, and implement all cryptographic primitives securely.

10.3. Potential Risks. Although the technical aspects of the flashing protocols can be secured against manipulation and tampering, there are still some risks involved resulting from nontechnical aspects. A malicious insider such as a SPE might be able to gain access to the SP, an authenticated TRK_{new} as well as a UD and the corresponding TRK_{orig}. Only if SPE has access to all aforementioned entities, then it is possible for him to flash TRK_{new} into UD, unknown to the legal owner OWN. Although such malicious misbehavior of SPE cannot be prohibited, it can at least be traced by logging all activity inside the SP.

A similar risk is faulty implementations of security primitives that are used in the protocol leading to a leak of secret cryptographic keys, thus enabling an attacker to impersonate an entity. The two main concerns regarding security leaks lie in the nontechnical aspects of the flashing protocol through mediators.

10.3.1. Social Engineering. Since the flashing of keys involves human interaction, this can offer an entry point for an attacker using social engineering [38–40]. A conceivable entry point is the SPE. If an attacker is able to extract the credentials from SPE, he can gain access to SP and therefore flash TRKs into any user device of OEM. This is a widely known issue in security systems in general that can only be countermeasured by proper training of SPE to enhance security awareness. Any misuse of the system through an SPE can be tracked if a secure log of all activities is provided within the trusted part of SP. As soon as misuse is detected, the trust delegation to SP can be revoked by OEM through not reissuing a trust keypair. Therefore, damage can be limited to flashing of the TRK_{new} already prepared for introduction into a UD.

Since the flashing scheme demands for authentication of the procedure through OWN, it is necessary to ensure the security of the second TRK_{orig} to be presented at the final stage of the process. If the second TRK_{orig} is in possession of an attacker and additionally the attacker has access to SP, he is able to flash an additional TRK into one UD. This attack is limited to a single UD, thus representing a fairly small risk, which on the other hand can easily be countermeasured.

10.3.2. SP Theft Scenario. If an SP carrying a valid trust key falls into the hands of a malicious attacker, the credential of SPE must also be known to the attacker in order to use SP to flash TRKs. Additionally, it is mandatory that the attacker also has a TRK_{new} in his possession that has already been certified by OEM. Additionally, TRK_{new} has to be bound to the stolen SP. If no such TRK_{new} is available, the system is not compromised. Even in case of such an aggressive attack the risk to the overall system is minimized, since only a very limited number of TRKs is flashable and a trust revocation can be carried out.

The risk level for such a scenario can be adapted through appropriate policies based on risk assessment through OEM. The most aggressive policy is not to allow flashing through a mediator. A minimal risk policy is to only have a single TRK on location that can be flashed into a UD. Only after it has been flashed successfully, does the OEM authenticate another TRK. While providing higher security, such restrictive policies will naturally inhibit usability.

11. Conclusions and Future Work

Access control systems are an important part in many systems such as vehicles or expensive machinery. Authentication protocols based on Public Key Cryptography offer advantages in logistics and key handling. These protocols are computationally very expensive but can be accelerated in hardware. In this paper, we presented a high-speed crypto architecture based on FPGA for fast authentication protocols based on HECC. Exploiting the properties of PKC, we introduced a scheme for pairing user devices (UD) and transponder tokens (TRK) by flashing public keys into UD. Compared to current key flashing procedures, our proposed protocol eliminates some logistic issues. TRKs do not have to be physically stored securely in SP any more. Also, shipment of TRKs does not have to be secured physically. Today OEM has to fully trust an SP to flash TRKs in the field. With our protocol the amount of trust in an SP can now be reduced to allow for risk management.

Security of the system is guaranteed by appropriate policy enforcement and usage of secure cryptographic primitives. No online connection is mandatory for linking a new transponder (TRK) for user authentication to a user device (UD). This makes it very practical for scenarios if TRK's have

to be replaced in the field with no intact communication infrastructure. It is applicable for a variety of embedded systems that need to implement and enforce access or usage restrictions in the field. We have shown the portability of the concept to non-FPGA platforms by implementing the protocol as a proof-of-concept using a combination of PC-based and FGPA-based protocol participants.

Flashing speed is of utmost importance in real-world implementation. To make allowance for a real world integration of the proposed flashing schemes, optimization is needed regarding usage and speed of the computational units involved. In the current prototype, the MicroBlaze processor has been used for simplicity and to show that the protocol can already be easily deployed in microprocessor driven embedded system such as the automotive domain. With the coprocessing unit in Section 7, very short computation times are achieved, and on a public key cryptosystem with a high security level than RSA. Adapting this system to the complete flashing scheme is target of future work and promises dramatic acceleration of the entire key flashing procedure.

One crucial point is the protection of the TRK's public key stored in the UD against physical attackers. Means to countermeasure attacks that might alter stored keys on a physical level need to be investigated in the future.

References

[1] H. Wallentowitz and K. Reif, *Handbuch Kraftfahrzeugelektronik: Grundlagen, Komponenten, Systeme, Anwendungen*, Vieweg, Wiesbaden, Germany, 2006.

[2] Hilti Corporation, "Electronic theft protection".

[3] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computer," *Communications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[4] T. Okamoto, "Provably secure and practical identification schemes and corresponding signature schemes," in *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '93)*, pp. 31–53, Springer, Santa Barbara, Calif, USA, 1993.

[5] C. P. Schnorr, "Efficient identification and signatures for smart cards," in *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '89)*, pp. 239–252, Springer, Santa Barbara, Calif, USA, August 1989.

[6] HIS Security Module Specification v1.1, Herstellerinitiative Software (HIS), 2006, http://www.automotive-his.de/.

[7] HIS-Presentation 2004-05, Herstellerinitiative Software (HIS), 2005, http://www.automotive-his.de/.

[8] G. de Boer, P. Engel, and W. Praefcke, "Generic remote software update for vehicle ecus using a telematics device as a gateway," *Advanced Microsystems for Automotive Applications*, pp. 371–380, 2005.

[9] A. Adelsbach, U. Huber, and A.-R. Sadeghi, "Secure software delivery and installation in embedded systems," in *Proceedings of the 1st International Conference on Information Security, Practice and Experience (ISPEC '05)*, R. H. Deng, Ed., vol. 3439 of *Lecture Notes in Computer Science*, pp. 255–267, April 2005.

[10] G. Dr-Ing and H. Brinkmeyer, "Authentikationsverfahren für fahrzeuganwendungen," *VDI-Berichte*, no. 1287, pp. 819–833, 1996.

[11] Microchip, "Keeloq authentication products," http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2074.

[12] B. Drisch and T. Zeggel, "Unterstützende Hardware-sicherheitsmodule für Automotive-anwendungen: voraussetzungen für die sichere umsetzung kryptographischer verfahren in fahrzeug-steuergeräten," *VDI Berichte*, no. 2016, pp. 147–156, 2007.

[13] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, "Public-key cryptography on the top of a needle," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '07)*, pp. 1831–1834, May 2007.

[14] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[15] A. Klimm, O. Sander, J. Becker, and S. Subileau, "A hardware/software codesign of a co-processor for real-time hyperelliptic curve cryptography on a spartan3 fpga," in *Proceedings of the 21st International Conference on Architecture of Computing Systems (ARCS '08)*, U. Brinkschulte, T. Ungerer, C. Hochberger, and R. G. Spallek, Eds., vol. 4934 of *Lecture Notes in Computer Science*, pp. 188–201, Springer, 2008.

[16] A. Klimm, O. Sander, and J. Becker, "A microblaze specific coprocessor for real-time hyperelliptic curve cryptography on Xilinx FPGAs," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, pp. 1–8, IEEE Computer Society, Rome, Italy, 2009.

[17] Atmel Corporation, "Ata5811/5812, uhf ask/fsk transceiver," 2006.

[18] L. Batina, D. Hwang, A. Hodjat, B. Preneel, and I. Verbauwhede, "Hardware/software co-design for Hyperelliptic Curve Cryptography (HECC) on the 8051 μP," in *Proceedings of the 7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '05)*, Lecture Notes in Computer Science, pp. 106–118, September 2005.

[19] H. Cohen, G. Frey, and R. Avanzi, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Chapman & Hall/CRC, Boca Raton, Fla, USA, 2006.

[20] Xilinx, "Picoblaze 8-bit embedded microcontroller user guide," 2005.

[21] K. Okeya, K. Schmidt-Samoa, C. Spahn, and T. Takagi, "Signed binary representations revisited," in *Proceedings of the 24th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '04)*, pp. 123–139, Santa Barbara, Calif, USA, 2004.

[22] R. Fan, *On the efficiency analysis of wNAF and wMOF*, Diploma thesis, September 2005, Supervised by Professor Dr. Tsuyoshi Takagi.

[23] S. K. Jain, L. Song, and K. K. Parhi, "Efficient semisystolic architectures for finite-field arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 1, pp. 101–113, 1998.

[24] M. W. Zuccherato, "An elementary introduction to hyperelliptic curves," Tech. Rep. CORR 96, University of Waterloo, Ontario, Canada, 1996.

[25] P. Engel and G. Hildebrandt, "Die rhythmischen schwankungen der reaktionszeit beim Menschen," *Psychological Research*, vol. 32, no. 4, pp. 324–336, 1969.

[26] N. Boston, T. Clancy, Y. Liow, and J. Webster, "Genus two hyperelliptic curve coprocessor," in *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 400–414, Springer, 2002.

[27] A. Klimm, M. Haas, O. Sander, and J. Becker, "A flexible integrated cryptoprocessor for authentication protocols based on hyperelliptic curve cryptography," in *Proceedings of the International Symposium on System-on-Chip (SoC '10)*, Tampere, Finland, September 2010.

[28] A. Weigl, K.-E. Weiss, C. Schroff et al., "Vehicle security device," Patent EP0 925 209, 2001, http://www.freepatentsonline.com/EP0925209B1.html.

[29] M. Hirozawa, A. Okamitsu, K. Adachi, and H. Tagawa, "Antivehicle-thief apparatus and code setting method of the apparatus," Patent EP0 695 675, 1999, http://www.freepatentsonline.com/EP0695675B1.html.

[30] Trusted Computing Group, TPM main specification v1.2.

[31] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[32] "National Institute of Standards and Technology (NIST): FIPS-180-2, Secure Hash Standard (SHS)," 2002, http://www.itl.nist.gov/fipspubs.

[33] "RSA Laboratories Inc: RSA Cryptograpy Standard PKCS No.1," http://www.rsa.com/ .

[34] MSDN, ".net framework class library—rsacryptoser-viceprovider class," http://msdn.microsoft.com/en-us/library/system.security.cryptography.

[35] D. L. Evans, K. H. Brown, A. Director, and W. M. Director, "Fips 140-1: security requirements for cryptographic modules," Category Computer Security, Gaithersburg, Md, USA, 1994.

[36] T. S. Denis, "Libtommath," http://math.libtomcrypt.com/.

[37] T. S. Denis, "Libtomcrypt," http://libtomcrypt.com/.

[38] S. Schumacher, "Adminsalbtraum," vol. 7, pp. 11–13, 2009, http://grundschutz.info/fileadmin/kundenbereich/Dokumente/Grundschutz 7-2009 11 13.pdf.

[39] S. Schumacher, "Admins albtraum," vol. 8, pp. 8-9, 2009, http://grundschutz.info/fileadmin/kundenbereich/Dokumente/Grundschutz 8-2009 8 9.pdf.

[40] S. Schumacher, "Admins albtraum," vol. 10/11, pp. 21-22, 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 952937, 9 pages
doi:10.1155/2011/952937

Research Article

Evaluation of the Reconfiguration of the Data Acquisition System for 3D USCT

Matthias Birk,[1] Clemens Hagner,[1] Matthias Balzer,[1] Nicole V. Ruiter,[1] Michael Hübner,[2] and Jürgen Becker[2]

[1] *Institute for Data Processing and Electronics, Karlsruhe Institute of Technology, P.O. Box 3640, 76021 Karlsruhe, Germany*
[2] *Institute for Information Processing Technology, Karlsruhe Institute of Technology, P.O. Box 6980, 76049 Karlsruhe, Germany*

Correspondence should be addressed to Matthias Birk, matthias.birk@kit.edu

Received 27 August 2010; Accepted 10 February 2011

Academic Editor: Gilles Sassatelli

As today's standard screening methods often fail to diagnose breast cancer before metastases have developed, an earlier breast cancer diagnosis is still a major challenge. To improve this situation, we are currently developing a fully three-dimensional ultrasound computer tomography (3D USCT) system, promising high-quality volume images of the breast. For obtaining these images, a time-consuming reconstruction has to be performed. As this is currently done on a PC, parallel processing in reconfigurable hardware could accelerate both signal and image processing. In this work, we investigated the suitability of an existing data acquisition (DAQ) system for further computation tasks. The reconfiguration features of the embedded FPGAs have been exploited to enhance the systems functionality. We have adapted the DAQ system to allow for bidirectional communication and to provide an overall process control. Our results show that the studied system can be applied for data processing.

1. Introduction

Breast cancer is the most common type of cancer among women in Europe and North America. Unfortunately, in today's standard screening methods, breast cancer is often initially diagnosed after metastases have already developed [1]. The presence of metastases decreases the survival probability of the patient significantly. Thus, early breast cancer diagnosis is still a major challenge.

A more sensitive imaging method could allow for detection in an earlier state and thus enhance the survival probability. With this ultimate goal, we are researching and developing a three-dimensional ultrasound computer tomography (3D USCT) system for early breast cancer diagnosis [2]. This method promises reproducible volume images of the female breast fully in 3D.

Our initial measurements of clinical breast phantoms using the first 3D prototype showed promising results [3, 4] and led to a new and optimized aperture setup [5], currently built and shown in Figure 1. It is equipped with over 2000 ultrasound transducers. Further virtual positions of the ultrasound transducers are created by rotational and translational movements of the complete sensor aperture.

In USCT, the interaction of unfocused ultrasonic waves with an imaged object is recorded from many different angles. During a measurement, the emitters sequentially send an ultrasonic wave front which interacts with the breast tissue and is recorded by the receivers as a pressure variation over time. These data sets, also called A-Scans, are sampled and stored for all possible emitter-receiver-combinations, resulting for our setup in over 3.5 million data sets and 20 GByte of raw data.

For acquisition of the A-Scan data during the measurement procedure, we use a massively parallel, FPGA-based data acquisition (DAQ) system. After completion, the recorded A-Scans are transferred to an attached computer workstation for the time-consuming image reconstruction. We exploit the pressure over time information in the A-Scans by a synthetic aperture focusing technique (SAFT) approach [6].

The necessary time for a volume reconstruction varies and strongly depends on both the desired image resolution

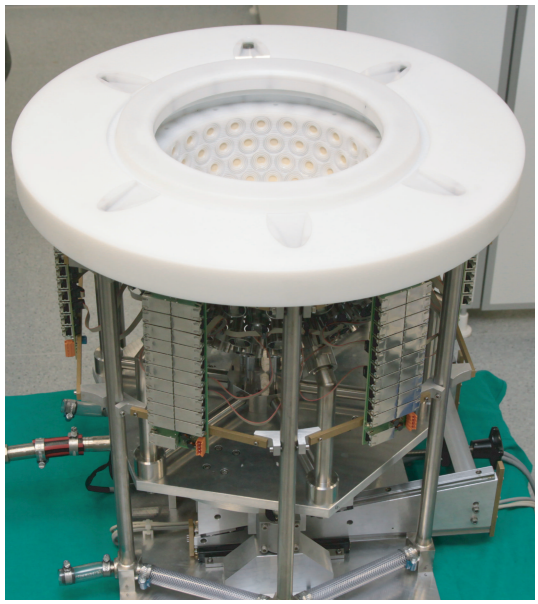


Figure 1: Image of the semiellipsoidal aperture of the new 3D USCT II. It is equipped with 628 ultrasound senders and 1413 receivers, grouped into 157 transducer array systems, mounted at the inner surface of the measurement basin.

and quality. However, our designated configuration for clinical application takes about 8 hours computed on an up to date PC. As we consider 5 minutes as an acceptable practical limit, the applied reconstruction algorithms need a significant acceleration of at least a factor 100 to be clinically relevant.

A promising approach to accelerate image reconstruction is parallel processing in reconfigurable hardware. In this work, we investigated the applicability of the DAQ system for further data processing tasks by a reconfiguration of the embedded FPGAs. As an exemplary processing sequence, we used the first step of the 3D USCT image reconstruction, which operates directly on the acquired A-Scans.

2. Related Work

The majority of processing algorithms which are applied in medical imaging methods feature an enormous compute and data intensity as well as a high degree of parallelism. Consequently, the application of parallel processors and further parallel accelerator architectures is investigated by many research groups. Special interest has recently been put on general purpose graphics processing units (GPUs), for example, [7–11], and the STI Cell Processer, for example, [11–13]. As our existing DAQ system is composed of a large number of FPGAs, which would be idle during the image reconstruction, the processing capabilities of such a system are here investigated.

Only a minority of medical imaging projects use FPGAs for computation. Most of them focus on well-established methods, like the X-ray-based Computed Tomography (CT) or Magnetic Resonance Imaging (MRI), which both differ from the algorithms used for 3D USCT. In the following, a few examples are presented.

In [14], the most computational intensive part of the volume reconstruction in CT, that is, the filtering and backprojection step, is ported onto a reconfigurable platform. The authors chose an accelerator board with nine FPGAs as the target architecture and yield an acceleration factor of over 20, in comparison with the software-based approach. Although the reconfiguration capability of the FPGAs is mentioned in this work, it is not used within the application. Similar approaches targeting the backprojection algorithm in CT have been shown in [15–17], all of which producing a significant acceleration.

Furthermore, [18] describes a basic reconfigurable 16 channel front-end for magnetic resonance imaging (MRI) on one FPGA. By means of the created partial dynamic reconfiguration framework, the authors replace the data acquisition module after the actual measurement took place by various processing units and therefore the chip resources are reused for image reconstruction. As this framework is created as a proof-of-concept, an acceleration factor is not given. The presented reconfiguration framework is very interesting as it follows the same basic principle as our work; however, the performed partial reconfiguration is not possible with the Altera FPGAs embedded in our DAQ system.

Closest to our application is a novel ultrasound based method presented in [19]. Jensen et al. show an FPGA-based data acquisition and processing system for ultrasound synthetic aperture imaging. The overall system is composed of 320 FPGAs, distributed over 64 identical boards, and is able to process 1024 ultrasound signals in parallel. However, the embedded FPGAs are statically configured and a dynamic reconfiguration of the FPGAs is not considered.

In summary, to the best of our knowledge, no reconfigurable computing system based on the system-wide FPGA reconfiguration for SAFT-based medical image reconstruction has been introduced.

3. Data Acquisition System

The data acquisition (DAQ) system has been developed as a common platform for multiproject usage, for example, in the Pierre Auger Observatory [20], the Karlsruhe Tritium Neutrino Project [21] and has also been adapted to the needs of 3D USCT. The DAQ system is described in detail in the following subsections.

3.1. Setup and Functionality. In the USCT configuration, the DAQ system consists of 21 expansion boards: one second level card (SLC) and 20 identical first level cards (FLC). The complete system fits into one 19″ crate, which is depicted in Figure 2. The SLC is positioned in the middle between 10 FLCs to the right and left.

The DAQ system holds 81 Altera Cyclone II FPGAs. Table 1 gives an overview of the FPGAs' device features. In total, up to 480 receiver signals can be acquired concurrently by assigning 24 channels to each FLC. This results in a receiver multiplex-factor of three for the acquisition of all

Figure 2: Image of the DAQ system in the USCT configuration. It is composed of one Second-Level Card (SLC) for measurement control and communication management (middle slot) and 20 First-Level Cards (FLC) for parallel sensor signal acquisition and data storage.

Table 1: FPGA device elements [22].

Element	Number per FPGA
Logic elements	33 216
Embedded multipliers (9 bit)	70
Total memory bits	483 840

possible emitter-receiver combinations in the new 3D USCT prototype.

The SLC controls the overall measurement procedure. It triggers the emission of ultrasound pulses and handles data transfers to the attached reconstruction PC. It is equipped with one Cyclone II FPGA and a processor module (Intel CPU, 1 GHz, 256 MB RAM) running a Linux operating system. Communication with the attached PC is either possible via Fast Ethernet or an USB interface. For communication between SLC and the FLCs within the DAQ system, a custom backplane bus is used.

3.2. First Level Card. An FLC consists of an analogue and a digital part. However, only the digital part will be considered throughout this paper. A block diagram of this part is given in Figure 3. In addition to three 8-fold ADCs for digitization of the 24-assigned receiver channels, an FLC is equipped with four Cyclone II FPGAs: we use one FPGA as a local control instance (Control FPGA, Cntrl FPGA). It handles communication and data transfer to the other FPGAs on board and to the SLC via backplane bus. We employ the other three FPGAs for actual signal acquisition. Each of these is fed by one ADC and thus processes eight receiver channels in parallel.

There are two different types of memory modules as intermediate storage for the acquired A-Scans on-board available: each Comp FPGA is connected to a distinct static RAM module (QDRII, 2 MB each) and the control FPGA is attached to a dynamic RAM module (DDRII, 2 GB), summing up to a system capacity of over 40 GB.

There are two separate means of communication between the control FPGA and the computing FPGAs (see Figure 3): a slow local bus with a width of 32 bit (Local Bus, 80 MB/s) and 8-bit wide direct data links (Fast Links, 240 MB/s per computing FPGA). Additionally, there are several connections for synchronization between the control FPGA and the computing FPGAs on each board.

4. Methodology

As outlined in Section 1, 3D USCT promises high-quality volumetric images of the female breast and therefore has a high potential in cancer diagnosis. However, it requires time-consuming image reconstruction steps, limiting the method's general applicability.

To achieve a clinical relevance of 3D USCT, that is, application in clinical routine, image reconstruction has to be accelerated by at least a factor of 100. A promising approach to reduce overall computation time is parallel processing of reconstruction algorithms in reconfigurable hardware.

In the current design, we use the DAQ system only to control the measurement procedure and to acquire the ultrasound receiver signals. In this evaluation, we investigated the utilization of the FPGAs in the DAQ system for further processing tasks. Due to limited FPGA recourses, the full set of the necessary processing algorithms, that is, for data acquisition, signal processing, and image reconstruction, cannot be configured statically onto the FPGAs at the same time.

Therefore, a reconfiguration of the FPGAs is necessary to switch between different configurations, enabling signal acquisition and further processing on the same hardware system. As the DAQ system has not been designed for further processing purposes, the scope of this work was to identify its capabilities as well as architectural limitations in this regard.

Only a reconfiguration of the FPGAs on the FLCs has been investigated since these hold the vast majority of FPGAs within the complete system. Therefore, only these cards and their data-flow are considered in the following sections. Furthermore, an interaction or data exchange between different FLCs has not been considered in this study.

The hardware setup of an FLC is given in Section 2 and shown in Figure 3. The detailed data-flow of an FLC in conventional operation mode is shown in Figure 4. As 24-receiver channels are processed per FLC, the signals are split into groups of eight. Every group is digitized in one ADC and fed into one computing FPGA. Within an FPGA, the signals are digitally filtered and averaged with previous recordings of the same emitter-receiver combination by means of the attached QDR memory. Note that during DAQ, the same emitter-receiver combination is sampled multiple times in order to improve the signal-to-noise ratio. Finally, the measurement data is transmitted via fast data links to the control FPGA, where it is stored in DDRII memory. After the complete measurement is finished, the resulting data is

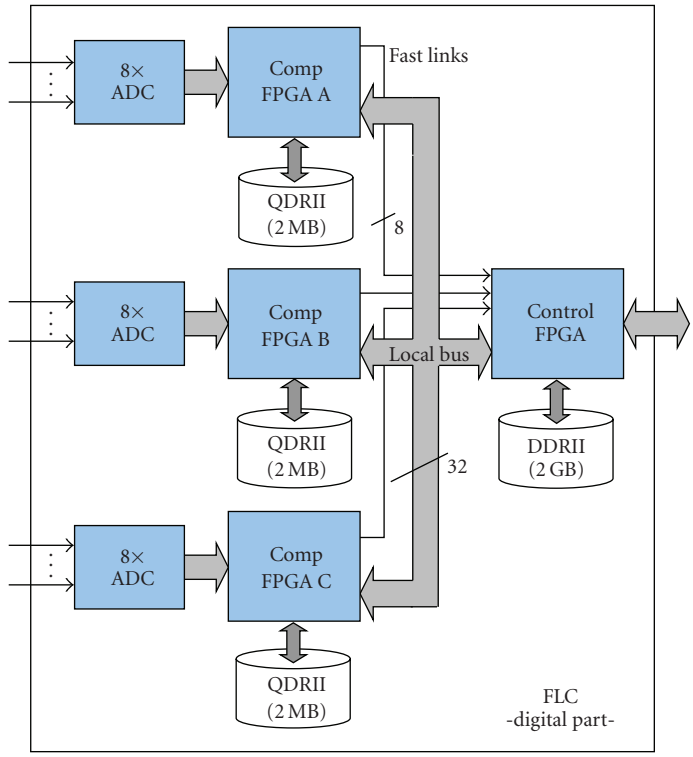


Figure 3: Block diagram of the digital part of an FLC in the 3D USCT DAQ system. It is equipped with four Altera Cyclone II FPGAs, one for local control (control FPGA, Cntr FPGA) and three for signal acquisition (computing FPGAs, Comp FPGA). Each Comp FPGA is fed by an 8-fold ADC and is attached to a 2 MB QDR static RAM module. The Cntrl FPGA is attached to a 2 GB DDRII dynamic RAM. Communication on each board is either possible by the slower local bus (Local Bus, 80 MB/s) or by fast data links (Fast Link, 240 MB/s).

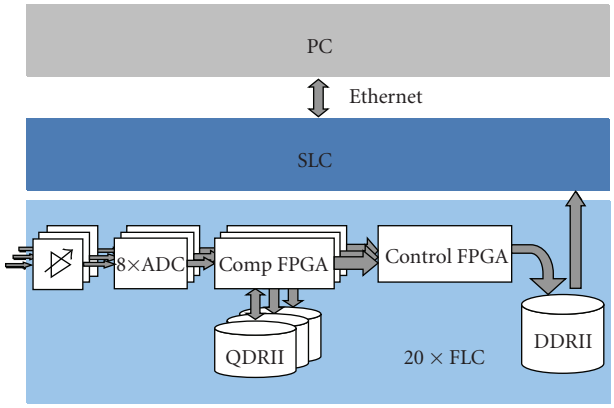


Figure 4: Detailed data-flow of one FLC during the conventional acquisition mode: Every FLC processes 24 receiver channels in parallel, whereas a group of eight signals is digitized in a single ADC. The digital signals are filtered and averaged in the computing FPGAs. Finally, the signals are transmitted to the Control FPGA and stored in DDRII memory.

transferred via backplane bus to the SLC and further to the attached PC for signal processing and image reconstruction.

In this work, we reconfigured the FPGAs after completion of a measurement cycle, that is, when the data is stored in DDRII memory, and thus switched from conventional acquisition to data processing mode. As depicted in Figure 5, instead of transmitting the data sets via SLC to the attached PC, we loaded them back to QDR II and subsequently processed them in the computing FPGAs. After completion, we transmitted the resulting data back to the control FPGA and again stored it in DDRII memory. For providing this reconfiguration methodology, we had to perform the following tasks:

(i) preventing data loss during reconfiguration,

(ii) establishing communication and synchronization,

(iii) implementing bidirectional communication interfaces.

4.1. Preventing Data Loss during Reconfiguration. Our DAQ system is built up of Altera Cyclone II FPGAs, which do not allow a partial reconfiguration [22]. Therefore, we had to reconfigure the complete FPGA chip. To prevent a loss of measurement data during the reconfiguration cycle, all data has to be stored outside the FPGAs in on-board memory, that is, QDRII or DDRII memory.

The QDRII is static memory so that stored data is not corrupted during reconfiguration of the FPGAs. However, only the larger memory (DDRII) is capable of holding all the data sets recorded per FLC. This dynamic memory module needs a periodic refresh cycle to hold stored data. On the FLC, the control FPGA is responsible for triggering

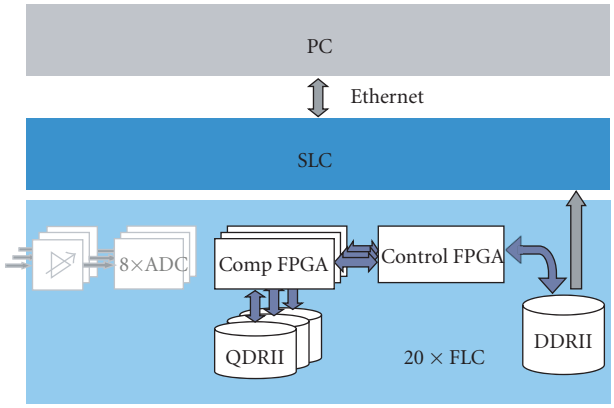


Figure 5: Detailed data-flow of one FLC during the newly created processing mode: as the data sets were previously stored in DDRII memory, we transferred them back to QDRII memory and processed them in the computing FPGAs. Finally, we stored the resulting data again in DDRII memory.

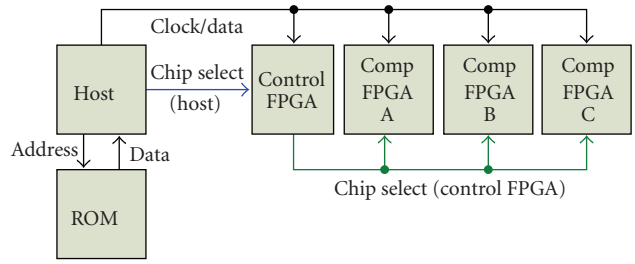


Figure 6: Passive serial configuration of the FPGAs on an FLC at system startup time: firstly, the control FPGA and after that the computing FPGAs are configured in parallel with configuration data from an embedded ROM.

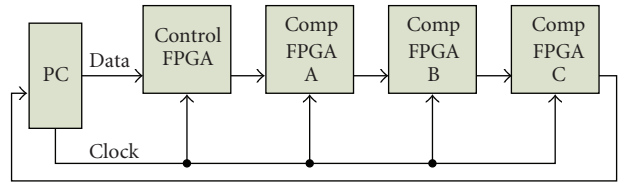


Figure 7: JTAG chain for reconfiguration of the FPGAs on an FLC. In JTAG reconfiguration mode, each FPGA can be selected separately for a reconfiguration, whereas deselected FPGAs remain in normal operation mode. Reconfiguration of different FPGAs within the chain is sequential.

these refresh cycles. During a reconfiguration, this FPGA would not able to perform this task. Since a refresh interval of the dynamic memory module is in the order of a few microseconds and a reconfiguration of the control FPGA takes even in the fastest mode about 100 ms [22], we had to ensure that it is not reconfigured; otherwise, data in the DDRII memory would be lost. Due to this requirement, we were only able to reconfigure the three computing FPGAs during operation.

At a normal startup of the DAQ system, all FPGAs on an FLC are configured via passive serial mode [22] with configuration data provided by an embedded ROM. As depicted in Figure 6, first the control FPGA and then all three computing FPGAs are configured in parallel in this mode. Constraint by the FLC printed circuit board, in current hardware setup we were not able to exclude the control FPGA from a configuration in passive serial mode. Therefore, in order to address and reconfigure each FPGA on the FLC separately, we had to use the JTAG configuration mode [22]. The JTAG chain through all four FPGAs is shown in Figure 7: each FPGA within the chain is configured sequentially with configuration data from an external programmer (PC).

4.2. Communication and Synchronization. Another important task in establishing the described reconfiguration methodology was to organize both communication and control on the FLC and furthermore, the synchronization of the parallel processing on the computing FPGAs.

As described in Section 2, there are two means of communication between the computing FPGAs and the control FPGA (see Figure 3): the slow local bus (Local Bus) and fast direct data links (Fast Links). In conventional DAQ operation mode, measurement data is transmitted only in the direction from the computing FPGAs to the control FPGA. Due to operational constraints in the FPGA pins, which are used for the fast links, this connection can only be operated in the above mentioned sense, that is, unidirectional. Thus, in the created processing mode, we had to rely on the slower local bus for data transfer since only this connection allows a bidirectional communication. The complete communication infrastructure is shown in Figure 8.

As the control FPGA is not reconfigured during operation, we had to statically configure it to handle data transfers appropriately in each system state, that is, DAQ and processing mode. Therefore, it must be able to determine the current state. As also depicted in Figure 8, we used a single on-board spare connection (*conf_state*) for that purpose, and therefore connected *conf_state* to all four FPGAs. In addition, we established process control and synchronization by further point-to-point links. Thus, each computing FPGA can be addressed and selected directly by the control FPGA: the respective *chip_select* signal triggers processing in a computing FPGA and completion of processing is indicated to the control FPGA by the *busy* signal.

4.3. Communication Interfaces. A further task was structuring communication and memory interfaces in the computing FPGAs. As a result, we created modular interfaces for transmitting data over the Local Bus (communication I/F) and storing data in QDRII memory (memory I/F). Figure 9 shows a block diagram of these modules on the computing FPGAs. This modular design allows a simple exchange of algorithmic modules without the need to change further elements.

The communication interface is managed by the control FPGA. It performs data transfers via Local Bus during processing or via fast data links during DAQ mode. The

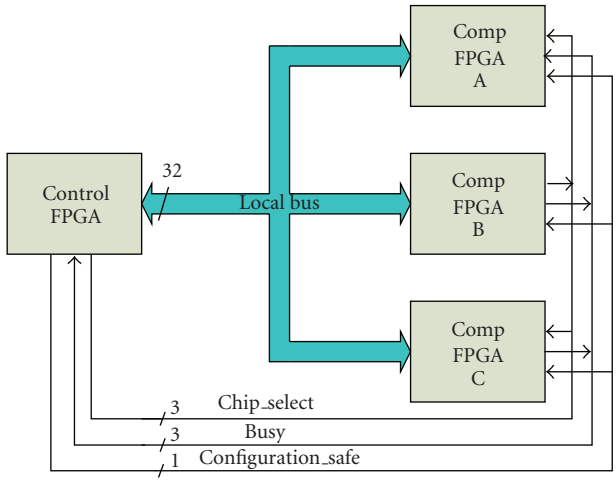


Figure 8: Communication structure during processing mode on an FLC: bidirectional data transfer is only possible via the slower Local Bus (80 MB/s). Separate point-to-point links (*chip_select* & *busy*) are used for control and synchronization of parallel processes. A further single point-to-point link is connected to all four FPGAs, indicating the current system state, that is, DAQ or processing mode.

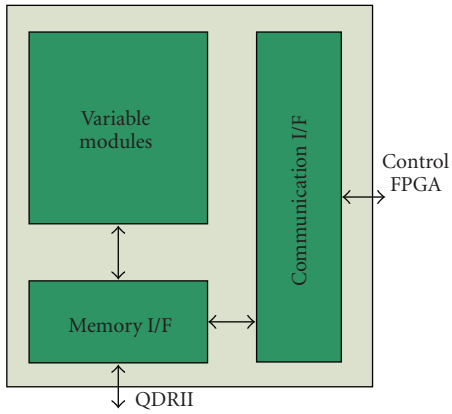


Figure 9: Block diagram of a computing FPGA: generic interfaces for communication with the control FPGA (communication I/F) via Local Bus or Fast Links and access to QDRII memory (memory I/F). The variable algorithmic part is also indicated.

memory interface handles accesses to the QDRII memory. We can be either access it by the control FPGA via Local Bus or from the algorithmic modules. In the current configuration, an algorithmic module only interacts with the memory interface and thus, only processes data which has already been stored in QDRII memory.

In order to ensure a seamless data transfer over the Local Bus, we supplemented the respective memory interface in the control FPGA by a buffered access mode to the DDRII memory. When we initialize a data transfer, enough data words are preloaded into a buffer so that the transmission is not interrupted during a refresh cycle.

5. Experimental

We tested the reconfigurable computing system by acquisition of test pulses, followed by the reconfiguration of the computing FPGAs and an exemplary data processing. The used pulse was in the same frequency range as regular measurement data and was handled like a normal data set (A-Scan). Our main goals of this test were to determine the required transfer time per data set over the Local Bus and the reconfiguration times per FPGA and per FLC.

5.1. Test Setup. For functional validation and performance measurements, we used a reduced setup of our DAQ system, which contains the SLC and only one FLC. However, as our exemplary processing sequence operates only locally on the FLC and does not include communication or interactions with other FLCs, this setup allows us an extrapolation of both reconfiguration and processing times for a fully equipped DAQ system.

5.2. Detailed Test Procedure. We tested the system as follows: at system startup, we loaded the initial DAQ configuration into the FPGAs as outlined in Section 4.1. Afterwards, we applied the test pulse at the inputs of the ADCs on the FLC. In DAQ mode, the digitized pulse is finally stored in DDRII memory on the FLC.

The further detailed procedure is indicated in Figure 10. After our manual reconfiguration of the computing FPGAs via JTAG, we transferred the first set of A-Scans to a computing FPGA (FPGA A) via Local Bus, stored them in its attached QDRII memory and subsequently processed them. While data in this FPGA is being processed, we supplied the other two computing FPGAs (FPGA B and FPGA C) with their first sets of A-Scans and started processing on these FPGAs. After processing is completed in FPGA A, we transmitted the resulting A-Scan data back to DDRII memory and sent further unprocessed A-Scans to this FPGA. We repeatedly applied this scheme until all A-Scans had been processed.

5.3. Examplary Processing Sequence. We used for this test a basic version of the so-called adapted matched filtering [23]. This processing sequence is applied as the first step of the 3D USCT image reconstruction and operates directly on the raw A-Scans in order to improve the signal-to-noise ratio, resulting in an enhanced image contrast. All processing steps are performed separately and independently from each other on all acquired A-Scans.

Initially, each A-Scan consists of 3000 time discrete samples with a width of 16 bit. Due to resource limitations, we had to retain this signal width throughout the processing and thus had to reduce the bit width after each computational step. The complete processing chain as implemented on the computing FPGAs and executed in this test is depicted in Figure 11.

Firstly, we read the raw A-Scan from QDRII memory and wrote it into embedded memory blocks within the FPGA. Then, we correlated this A-Scan with the matched filter [24]

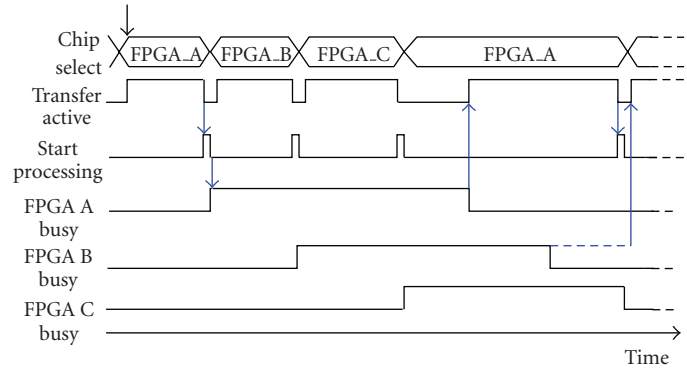


Figure 10: Test procedure after DAQ and manual reconfiguration of the Comp FPGAs via JTAG. Firstly, we supplied computing FPGA A with a set of A-Scans and started processing on this FPGA. While processing is underway, we initiated data transfer and processing on the Comp FPGAs B and C. After completion of processing on FPGA A, we transferred the resulting data back to DDRII memory and loaded further unprocessed A-Scans. This scheme is repeatedly applied.

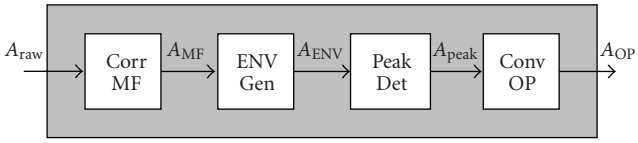


Figure 11: Complete processing chain of the adapted matched filtering [19] as implemented on the computing FPGAs. Firstly, we correlated the raw A-Scan with the matched filter signal (CorrMF). Then, we generated the absolute signal envelope (EnvGen) and in the following reduced it to its local maxima (PeakDet). Finally, the intermediate signal is convoluted with an optimal pulse for reflectivity imaging (ConvOP).

(CorrMF). This is the expected wave form at the receivers and was estimated by previous empty measurements. The correlation kernel consists of 64 samples of 12-bit integers. We performed this calculation by means of embedded multipliers [22].

In the next step, we generated the envelope of the resulting signal (EnvGen) by firstly creating the absolute value signal and then applying an adapted cascaded integrated comb (CIC) filter [25]. Subsequently, we reduced this intermediate signal to its local maxima (PeakDet), that is, we retained the signal values at local peaks, whereas all other samples within the A-Scan are set to zero. We did this in a streamed manner by a direct comparison of every sample with both its immediate predecessor and successor.

Finally, we convoluted this signal with an optimal pulse for reflectivity imaging as illustrated in [26]. The convolution kernel consists of 17 samples of 12-bit integers. Again, we performed the calculation by means of the embedded multipliers. After completion of the described processing steps, we wrote the resulting processed A-Scans back to the attached QDRII memory module.

5.4. Results. A JTAG configuration of a single computing FPGA requires 1.8 s, resulting in a reconfiguration time of 5.4 s for one FLC, when only the three computing FPGAs are configured in the JTAG chain. A reconfiguration of all 60 computing FPGAs, distributed over the 20 FLCs in the complete DAQ system, would take up to 2 minutes by building up a JTAG chain through all FPGAs. The determined JTAG reconfiguration times are illustrated in Table 2.

Table 2: JTAG reconfiguration times.

Procedure	Required time
Reconf. of one computing FPGA	1.8 s
Reconf. of one FLC	5.4 s
Extrapolated reconf. of the DAQ system	~2 min

Table 3: Computing FPGA occupation.

Components	Configuration		
	Data acquisition	Data processing	Comm & mem I/F only
Logic elements	17%	19%	4%
Embedded multipliers	97%	91%	0%
Memory bits	<1%	68%	0%

The transfer of one data set via Local Bus in either direction, that is, from control FPGA to computing FPGA or vice versa, takes 75 μs. Usage of the Local Bus is limited to one computing FPGA at a time and the same bus is used for data transfer to and from all three computing FPGAs.

Table 3 outlines the occupation of the computing FPGA during the test procedure. The extensive use of embedded multipliers in DAQ mode, which are required due to the hard real-time constraints, states a clear demand for the established reconfiguration methodology. Furthermore, the implemented communication and memory interfaces occupy only 4% of the device's logic elements.

The implemented adapted matched filtering takes 450 μs per A-Scan on a single computing FPGA. By employing the entangled processing method in Figure 10, the processing time of a set of three A-Scans equals 750 μs on one FLC. As this processing sequence does not include any interactions with other FLCs, the extrapolated processing time per A-Scan

on the complete DAQ system (20 FLCs) results in to 12.5 μs. The current computation in Software on an Intel Core i7-920 at 2.67 GHz takes in Matlab about 840 μs and for a multithreaded C implementation using 8 threads (hyperthreading), compiled with gcc 4.4 with −O3 flag, about 40 μs per A-Scan.

6. Discussion and Conclusions

In this paper, we presented a feasible concept of a reconfigurable computing system based on an existing DAQ system for 3D USCT. As the main result, we showed the possibility of reusing this system for data processing.

The main goals were to analyze the DAQ system's characteristics, to determine limitations and derive implementations strategies. Thus, the performance comparison of our exemplary processing sequence may be misleading. The algorithmic implementation could be further optimized in both computational strategy and quality of results. Nevertheless, already in this basic version, we obtained an acceleration of about factor three if we compare the pure computational time with a multithreaded C implementation on an up to date CPU. However, when comparing the total processing time for a full set of A-Scans (~3.5 million) and taking the necessary reconfiguration of the complete DAQ system into account, the achieved speed-up almost balances out. On the other hand, File −I/O from and to hard disk has not been considered in the software-based processing time, which also contributes substantially and degrades the overall performance by a factor of two.

The main drawback of the current system is the long reconfiguration time. Thus, the reconfiguration cycles impact the total processing time significantly. To which extent this constraint will restrict the applicability of the presented method needs further investigation. However, the reconfiguration time can be reduced by a factor of 20 by separate JTAG chains for each FLC and a concurrent reconfiguration.

Likewise, due to the slow bidirectional data transfer over the Local Bus, the achievable performance during the processing phase is also limited. This issue could be improved by a modified communication scheme where data from the computing FPGAs to the control FPGA is transferred via Fast Links and the Local Bus is only used for the opposite direction from the control FPGA to the computing FPGAs. As the Fast Links have a much larger data rate, this would accelerate the overall data transfer by a factor two with the Local Bus being the limiting factor.

Assuming the applied data parallel processing strategy, that is, each computing FPGA performs the same computation on a different data set, a high efficiency can be reached if the following condition holds: the parallelized processing time per A-Scan on a computing FPGA has to be longer than 450 μs, which is six times the transfer time of a single data set. In this context, parallelized time is the processing time per A-Scan on one computing FPGA divided by the number of concurrently processed A-Scans on this FPGA. Then, the transfer times to and from all three FPGAs could be hidden.

If this processing strategy is not feasible for a given algorithm or it requires communication between different FLCs other effects come into play. These may limit scalability and need further consideration.

7. Outlook

For future work, two obvious aspects have been derived in the last section. Namely, reducing data transfer time by a modified communication scheme in the processing phase and reducing reconfiguration time by parallel JTAG chains.

A further task will be porting more processing algorithms to the DAQ system in order to evaluate the established reconfiguration ability in the real application. This also includes the operation of the full DAQ system with all 20 FLCs. So far, we did not consider a direct communication between the computing FPGAs on an FLC as well as an interaction of different FLCs in general. This will open up manifold implementation strategies for algorithmic modules besides the applied data parallel processing scheme, but also increases the implementation effort.

In the long term, a redesign of the DAQ system will focus on enhanced processing capabilities. This includes high-speed data transfer as well as research in heterogeneous computing concepts by combining FPGAs with further processing elements like GPUs or multicore CPUs.

Acknowledgments

The authors acknowledge support by Deutsche Forschungsgemeinschaft and Open Access Publishing Fund of Karlsruhe Institute of Technology.

References

[1] D. van Fournier, H. J. H.-W. Anton, and G. Bastert, "Breast cancer screening," in *Cancer Diagnosis: Early Detection*, P. Bannasch, Ed., pp. 78–87, Springer, Berlin, Germany, 1992.

[2] H. Gemmeke and N. V. Ruiter, "3D ultrasound computer tomography for medical imaging," *Nuclear Instruments and Methods in Physics Research A*, vol. 580, no. 2, pp. 1057–1065, 2007.

[3] N. V. Ruiter, G. F. Schwarzenberg, M. Zapf, and H. Gemmeke, "Conclusions from an experimental 3D ultrasound computer tomograph," in *Proceedings of IEEE Nuclear Science Symposium Conference Record (NSS/MIC '08)*, pp. 4502–4509, October 2008.

[4] N. V. Ruiter, G. F. Schwarzenberg, M. Zapf, A. Menshikov, and H. Gemmeke, "Results of an experimental study for 3D ultrasound CT," in *Proceedings of NAG-DAGA International Conference on Acoustics*, 2009.

[5] G. F. Schwarzenberg, M. Zapf, and N. V. Ruiter, "Aperture optimization for 3D ultrasound computer tomography," in *Proceedings of IEEE Ultrasonics Symposium (IUS '07)*, pp. 1820–1823, October 2007.

[6] S. R. Doctor, T. E. Hall, and L. D. Reid, "SAFT—the evolution of a signal processing technology for ultrasonic testing," *NDT International*, vol. 19, no. 3, pp. 163–167, 1986.

[7] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the Common Unified

Device Architecture (CUDA)," in *Proceedings of IEEE Nuclear Science Symposium Conference*, vol. 6, pp. 4464–4466, 2007.

[8] K. Mueller, F. Xu, and N. Neophytou, "Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?" in *Computational Imaging V*, vol. 6498 of *Proceedings of the SPIE*, San Jose, Calif, USA, January 2007.

[9] S. S. Stone, J. P. Haldar, S. C. Tsao, W. M. W. Hwu, B. P. Sutton, and Z. P. Liang, "Accelerating advanced MRI reconstructions on GPUs," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1307–1318, 2008.

[10] T. Schiwietz, T.-C. Chang, P. Speier, and R. Westermann, "MR image reconstruction using the GPU," in *Medical Imaging 2006: Physics of Medical Imaging*, vol. 6142 of *Proceedings of SPIE*, February 2006.

[11] S. Van Der Maar, K. J. Batenburg, and J. Sijbers, "Experiences with cell-BE and GPU for tomography," in *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS '09)*, vol. 5657 of *Lecture Notes in Computer Science*, pp. 298–307, July 2009.

[12] M. Knaup, S. Steckmann, O. Bockenbach, and M. Kachelrieß, "Tomographic image reconstruction using the Cell Broadband Engine (CBE) general purpose hardware," in *Computational Imaging V*, vol. 6498 of *Proceedings of SPIE*, January 2007.

[13] H. Scherl, M. Koerner, H. Hofmann, W. Eckert, M. Kowarschik, and J. Hornegger, "Implementation of the FDK algorithm for cone-beam CT on the cell broadband engine architecture," in *Medical Imaging 2007: Physics of Medical Imaging*, vol. 6510 of *Proceedings of SPIE*, February 2007.

[14] B. Heigl and M. Kowarschik, "High-speed reconstruction for C-arm computed tomography," in *Proceedings of the 9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, 2007.

[15] N. Sorokin, "Parallel backprojector for cone-beam Computer Tomography," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 175–180, December 2008.

[16] S. Coric, M. Leeser, E. Miller, and M. Trepanier, "Parallel-beam backprojection: an FPGA implementation optimized for medical imaging," in *Proceedings of the 10th ACM International Symposium on Field-Programmable Gate Arrays (FPGA '02)*, pp. 217–226, February 2002.

[17] N. Gac, S. Mancini, and M. Desvignes, "Hardware/Software 2D-3D backprojection on a SoPC Platform," in *Proceedings of the ACM Symposium on Applied Computing*, pp. 222–228, April 2006.

[18] I. L. Dalal and F. L. Fontaine, "A reconfigurable FPGA-based 16-channel front-end for MRI," in *Proceedings of the Asilomar Conference on Signals, Systems and Computers*, pp. 1860–1864, 2006.

[19] J. A. Jensen, M. Hansen, B. G. Tomov, S. I. Nikolov, and H. Holten-Lund, "System architecture of an experimental synthetic aperture real-time ultrasound system," in *Proceedings of IEEE Ultrasonics Symposium (IUS '07)*, pp. 636–640, October 2007.

[20] H. Gemmeke et al., "First measurements with the auger fluorescence detector data acquisition system," in *Proceedings of the 27th International Cosmic Ray Conference*, 2001.

[21] A. Kopmann, T. Bergmann, H. Gemmeke et al., "FPGA-based DAQ system for multi-channel detectors," in *Proceedings of IEEE Nuclear Science Symposium (NSS/MIC '08)*, pp. 3186–3190, October 2008.

[22] Altera Corporation, *Cyclone II Device Handbook*, Altera, 2007.

[23] N. V. Ruiter, G. F. Schwarzenberg, M. Zapf, and H. Gemmeke, "Improvement of 3D ultrasound computer tomography images by signal pre-processing," in *Proceedings of IEEE International Ultrasonics Symposium (IUS '08)*, pp. 852–855, November 2008.

[24] D. O. North, "An Analysis of the factors which determine signal/noise discrimination in pulsed-carrier systems," *Proceedings of the IEEE*, vol. 51, no. 7, pp. 1016–1027, 1963.

[25] E. B. Hogenauer, "An economical class of digital filters for decimation and interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 2, pp. 155–162, 1981.

[26] S. J. Norton and M. Linzer, "Ultrasonic reflectivity tomography: reconstruction with circular transducer arrays," *Ultrasonic Imaging*, vol. 1, no. 2, pp. 154–184, 1979.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 156946, 15 pages
doi:10.1155/2011/156946

Research Article

Dynamic Application Model for Scheduling with Uncertainty on Reconfigurable Architectures

Ismail Ktata,[1,2] Fakhreddine Ghaffari,[1] Bertrand Granado,[1] and Mohamed Abid[2]

[1] *ETIS Laboratory, CNRS UMR8051, University of Cergy-Pontoise, ENSEA, 6 avenue du Ponceau 95014 Cergy-Pontoise, France*
[2] *Computer & Embedded Systems Laboratory (CES), University of Sfax, ENIS, 3038 Sfax, Tunisia*

Correspondence should be addressed to Ismail Ktata, ismail.ktata@ensea.fr

Received 26 August 2010; Revised 16 December 2010; Accepted 10 February 2011

Academic Editor: Michael Hübner

Applications executed on embedded systems require dynamicity and flexibility according to user and environment needs. Dynamically reconfigurable architecture could satisfy these requirements but needs efficient mechanisms to be managed efficiently. In this paper, we propose a dedicated application modeling technique that helps to establish a predictive scheduling approach to manage a dynamically reconfigurable architecture named OLLAF. OLLAF is designed to support an operating system that deals with complex embedded applications. This model will be used for a predictive scheduling based on an early estimation of our application dynamicity. A vision system of a mobile robot application has been used to validate the presented model and scheduling approach. We have demonstrated that with our modeling we can realize an efficient predictive scheduling on a robot vision application with a mean error of 6.5%.

1. Introduction

Embedded systems are everywhere in our automobiles, robots, planes, satellites, boats, industrial control systems, and so forth. An important feature of such systems is to be reactive, as they continuously react with their environment at a rate imposed by this later itself. They receive inputs, process these stimuli, and produce the outputs, known as reactions. In general, the inputs are not known until they appear, and the systems must react dynamically to these stimuli. DRAs (dynamically reconfigurable architectures) are good candidate to implement an embedded application as they could be in perfect adequacy with its dynamic behavior.

In this paper, we are interested in the modeling and scheduling of soft real-time applications that could be embedded on DRA. Soft real time constraint means that the application may tolerate lateness in its deadline and could respond with decreased quality of service (loss of frames in video, e.g.). One of the particularities of the considered applications is their uncertainty: we cannot have complete information on the characteristics of the processing. These characteristics, latency, functionality, resources need, and so forth, depend on the environment inputs and are known as the application is processed (e.g., in computer vision, object recognition process depends on the number of objects present in the image or video sequence, lighting or color, viewing direction, size/shape, etc.). One of the solutions to minimize the effect of the dynamicity of the application is to realize a predictive scheduling that takes into account the characteristics of the application. In order to realize this predictive scheduling we need to exhibit the dynamicity of the application. In this paper, we propose an original dynamic application model that could be used to do predictive scheduling. We present also a complete and original predictive scheduling based on our dynamic applications model.

The remainder of our paper is structured as follows: Section 2 introduces the context and the problematic. Section 3 describes the related works on modeling techniques as well as the scheduling approaches for dynamic systems. Our new proposed method of dynamic application modeling is presented in Section 4 and compared to other models. Section 5 outlines the predictive scheduling technique. Section 6 presents a real case example of a vision system on

a mobile robot and the validation of our model and our predictive scheduling. The last section concludes the paper with a summary of achievements.

2. Context and Problem Definition

Today, the growing complexity of soft real-time applications represents important challenges due to their dynamic behavior and uncertainties which could happen at runtime [1]. To overcome these problems, designers tend to use DRAs that are well suited to deal with the dynamism of applications and allow better compromise between cost, flexibility and performance [2]. In particular, fine grained dynamically reconfigurable architectures (FGDRAs), as a kind of DRAs, can be adapted to any application with a great optimality. However, this type of architecture makes the applications design very complex [3], especially with the lack of suitable and efficient tools. This complexity could be abstracted at runtime by providing an operating system which abstracts the lower level of the system [4]. This operating system has to be able to respond rapidly to events. In the case of soft real-time application with dynamic behavior, the goal is to meet some quality of service requirements. This goal can be achieved by an operating system with a suitable predictive scheduling approach.

In order to realize an efficient predictive scheduling of an application, an operating system needs to know the behavior of this application, in particular the part where the dynamicity can be efficiently exploited on a DRA.

In this paper, we focus on two major problems to realize an efficient scheduling on an FGDRA.

(a) The modeling of the application that should exhibit its dynamical aspects and must allow the expression of its constraints, in particular real-time constraints.

(b) The run-time performance of the predictive scheduling algorithm.

The different items of a scheduling problem are the tasks, the constraints, the resources, and the objective function. Many resolution strategies have been proposed in literature [5]. These methods usually assume that execution times can be modeled with deterministic values. They use an off-line schedule that gives an explicit idea of what should be done. Unfortunately, in real environments, the probability of a precomputed schedule to be executed exactly as planned is low [6]. This is due to not only variations, but also to a lot of data that are only previsions or estimations. It is then necessary to deal with uncertainty or flexibility in the process data. Hence, a significant on-line reformulation of the problem and the solving methods are needed in order to facilitate the incorporation of this uncertainty and imprecision in scheduling [7].

Uncertainty in scheduling may arise from many sources [8]:

(a) the release time of tasks can be variable, even unexpected;

(b) new unexpected tasks may occur. We named such a task a hazardous task;

(c) cancellation or modification of existing tasks;

(d) resources may become unavailable;

(e) tasks assignments: if a task could be done on different resources (identical or not), the choice of this resource can be changed. This flexibility is necessary if such a resource becomes unusable or less usable than others;

(f) the ability to change execution mode: this mode includes the approval or disapproval of preemption, whenever a task could be resumed or not, the overlap between tasks, changing the range of a task, changing the number of resources needed for a task, and so forth.

In order to describe these features, a new model description is needed. It should be a little sensitive to data uncertainties and variations and adaptable to the possible disturbances.

3. Related Works

3.1. Modeling Methods. In order to efficiently implement an application, it should be described by a suitable model showing significant system characteristics of geometry, information, and dynamism. The latter is a crucial application characteristic as it permits to represent how an application behaves and changes states over time. Moreover, dynamic modeling can cover different application domains from the very general to the very specific [9]. Model types have different presentations, as shown in Figure 1; some are text-based using symbols while others have associated diagrams.

(a) Graphical models use a diagram technique with named symbols that represent processes, lines that connect the symbols and show relationships, in addition to various other graphical notations representing constraints (Figures 1(a), 1(b), and 1(d)).

(b) Textual models typically use standardized keywords accompanied by parameters (Figure 1(c)).

In addition, some models have static form, whereas others have natural dynamics during model execution as in Figure 1(a). The solid circle (a token) moves through the network and represents the execution behavior of the application.

In the domain of embedded systems, a large number of modeling languages have been proposed [10–12], including extensions to finite state machines, data flow graphs, communicating processes, and Petri nets. In this section, we present main models of computation for real-time applications reported in the literature.

3.1.1. Finite State Machines. The finite state machine (FSM) representation is probably the most well-known model used for describing control systems. However, one of the disadvantages of FSMs is the exponential growth of the number of states to be explicitly captured as the system complexity rises, making the model increasingly difficult to

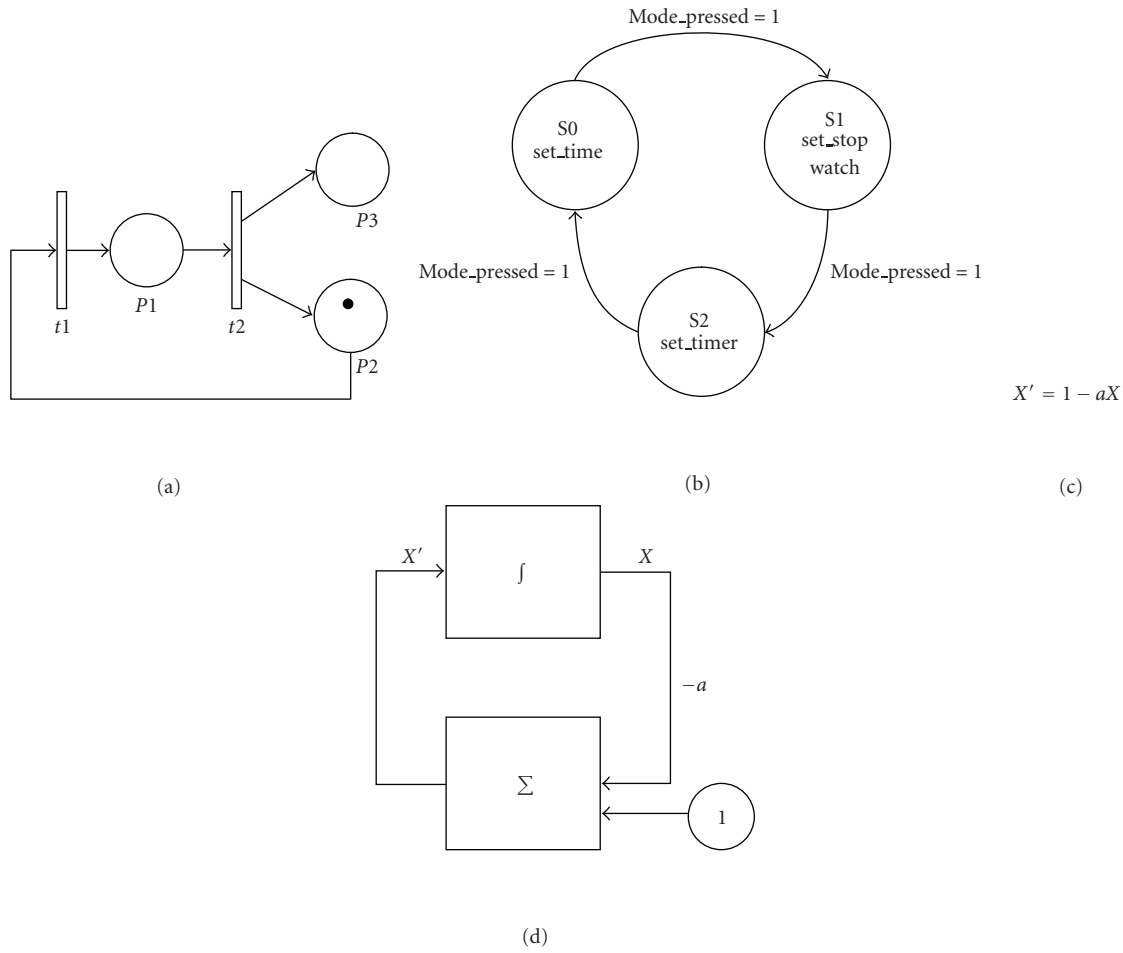


Figure 1: Four types of dynamic system models. (a) Petri net. (b) Finite state machine. (c) Ordinary differential equation. (d) Functional block model.

visualize and analyze [13]. For dynamic systems, the FSM representation is not appropriate because the only way to model it is to create all the states that represent the dynamic behavior of the application. It is then unthinkable to use it as the number of states could be prohibitive.

3.1.2. Data-Flow Graph. A data-flow graph (DFG) is a set of compute nodes connected by directed links representing the flow of data. It is very popular for modeling data-dominated systems. It is represented by a directed graph whose nodes describe the processing and the arcs show the partial order followed by the data. However, the conventional model is inadequate for the systems control unit representation [14]. It does not provide information about the ordering of processes for the scheduler. It is therefore inappropriate to model dynamic applications.

3.1.3. Program Evaluation and Review Technique (PERT). PERT is a model that was originally introduced in 1958 by the US Navy for its polaris weapon system. Since then, PERT has spread rapidly throughout almost all industries. Tasks are represented by arcs with an associated number presenting their duration. Between arcs, there are circles marking events of beginning or end of tasks (Figure 2). Each node (i) contains three characteristics: event's number (event number i), an earliest start time (R_i) on which an event can be expected to take place, and a latest finish time (D_i) on which an event can take place without extending the completion date of the application. The difference between the latest finish time and the earliest start time is called slack time. A task, from node A to node B, is considered critical if the difference between the latest finish time of B and the earliest start time of A is equal to the execution time of the task. All critical tasks form the critical path, which is the path on which no task should be delayed, so that the whole application would not be delayed.

An interesting feature of PERT model is that it allows predicting the minimum and maximum duration of the application based on a known tasks execution time [15]. For each event, PERT indicates earliest time when a task can start/finish and latest time when a task can start/finish. The earliest and latest times are considered as random variables. To estimate them, the scheduler refers to the schedule of event occurrences that were established at the beginning of the application. The scheduler would have at

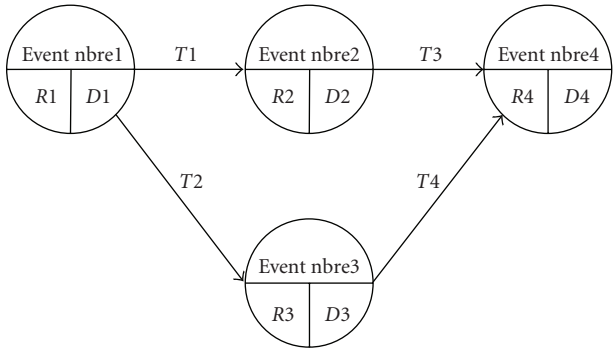


Figure 2: PERT graph.

his disposal a large volume of historical data from which to make its estimates. Obviously, the more historical data is available, the more reliable the estimate is. The calculations of critical paths and slack times were based on these best estimates. This model seems interesting as it can describe a dynamic duration of a task, but it cannot model other dynamic features, for instance, the variable number of tasks or resources.

3.1.4. Petri Net. Petri net (PN) is a modeling formalism which combines a well-defined mathematical theory with a graphical representation of the dynamic behavior of systems [9]. Petri net is a 5-tuple PN $= (P, T, F, W, M_0)$ where P is a finite set of places which represent the status of the system before or after the execution of a transition. T is a finite set of transitions which represent tasks. F is a set of arcs (flow relation). $W : F \rightarrow \{1, 2, 3, \ldots\}$ is a weight function. M_0 is the initial marking. However, though Petri net is well established for the design of static systems, it lacks support for dynamically modifiable systems [16]. In fact, the PN structure presents only the static properties of a system while the dynamic one results from PN execution which requires the use of tokens or markings (denoted by dots) associated with places [17]. The conventional model suffers from good specification of complex systems such as lack of the notion of time which is an essential factor in embedded applications and lack of hierarchical composition [18]. Therefore, several formalisms have independently been proposed in different contexts in order to overcome the problems cited above, such as introducing the concepts of hierarchy, time, and valued tokens. Timed PNs are those with places or transitions that have time durations in their activities. Stochastic PNs include the ability to model randomness in a situation and also allow for time as an element in the PN. Colored PNs enable the user and designer to witness the changes in places and transitions through the application of color-specific tokens, and movement through the system can be represented through the changes in colors [9].

None of the methodologies mentioned provides sufficient support for systems which include dynamic features. Dynamic creation of tasks, for instance, is not supported. In [18], authors proposed an extension of high-level Petri net model [19] in order to capture dynamically modifiable embedded systems. They coupled that model with graph transformation techniques and used a double push-out approach which consists of the replacement of a Petri net by another Petri net after firing of transitions. This approach allows modeling dynamic tasks creation but not variable execution time nor variable number of needed resources. As we can see there is no model to exhibit dynamic features of an application. Next section will present related works about scheduling under uncertainty.

3.2. Scheduling under Uncertainty. In literature, most of the works have been focused on finding optimal or near-optimal predictive schedules for simple scheduling models with respect to various criteria assuming that all problem characteristics are deterministic. However, many embedded systems operate in dynamic environments, frequently subject to various real-time events and several sorts of perturbations, such as random task releases, resources failure, task cancellation, and execution time changes. Therefore, dynamic scheduling is of great importance for the successful implementation of real-world scheduling applications. In general, there are two main approaches dealing with uncertainty in a scheduling environment according to phases in which uncertainties are taken into account [8].

(a) Proactive scheduling approach aims at building a robust baseline schedule, that is, protected as much as possible against disruptions during schedule execution. It takes into account uncertainties only in design phase (offline). Hence, it constructs predictive schedule based on statistical and estimated values for all parameters, thus implicitly assuming that this schedule will be executed exactly as planned. However, this could become infeasible during the execution due to the dynamic environment, where unexpected events continually occur. Moreover, overestimations may lead to a schedulability test failure and overutilization of resources. Therefore, in this case, a reactive approach may be more appropriate [8].

(b) Instead of anticipating future uncertainties, reactive scheduling takes decisions in real time when some unexpected events occur. A reference deterministic scheduling, determined offline, is sometimes used and reoptimized. In general, reactive methods may be more appropriate for high degrees of uncertainty, or when information about the uncertainty is not available.

A combination of the advantages of both precedent approaches is called proactive-reactive scheduling. This hybrid method implies a combination of a proactive strategy for generating a protected baseline schedule with a reactive strategy to resolve the schedule infeasibilities caused by the disturbances that occur during schedule execution. Hence, this scheduling/rescheduling method permits to take into account uncertainties all over the execution process and ensures better performance [20, 21]. For rescheduling, the literature provided two main strategies: schedule repair and

complete rescheduling. The first strategy is most used as it takes less time and preserves the system stability [22].

Dynamic scheduling techniques are widely studied in information and industrial systems and are quite different depending on the nature of the problem and the type of disturbance considered: resources failure, the variation of the tasks duration and the fact that new tasks can occur, and so forth. The mainly used methods are dispatching rules, heuristics, metaheuristics, and artificial intelligence techniques [23]. In [24], authors considered a scheduling problem where some tasks (called "uncertain tasks") may need to be repeated several times to satisfy the design criteria. They used an optimization methodology based on stochastic dynamic programming. In [25, 26], scheduling problem with uncertain resource availabilities was encountered. Authors used proactive-reactive strategies and metaheuristic algorithm that combines genetic algorithms. Another uncertainty case, which is uncertain tasks duration, had been studied in [27, 28]. Since their complexity in implementation and calculation time, metaheuristic techniques are more easily to be applied offline. On the other hand, when it is about an online context, priority-based scheduling (dispatching rules and list scheduling) is more rapid to have reasonable solutions. However, priority-based scheduling is inefficient and not appropriate for systems where the required scheduling behavior changes during runtime [29]. Therefore, it is necessary, in our case, to combine different techniques together to endow the scheduling approach with the required flexibility and robustness. For permanent tasks with deterministic features, priority-based algorithms will be practical, while for hazardous tasks, the scheduler should occur to a prediction service based on heuristic techniques. Taking the case of tasks with uncertain execution time, the predictive algorithm must take advantage of dynamic characteristics of the application and attribute, for each task, a dynamic online estimated value. This will permit to schedule the different hardware tasks (T_i) and allocate its needed resources of the DRA based on online estimated values.

There are basically two approaches for execution time prediction [30].

(a) Static approaches: these methods do not need a code execution on real hardware or on a simulator but rather rely on code structure and possible control-flow paths analysis to compute upper bounds of execution times [31]. A given code analysis technique is typically limited to a specific code type or a limited class of architectures. Thus, these methods are not very applicable to DRA. In addition, these offline analysis methods do not take into account changes in the processed data on each tasks' activation. Therefore, online changes in execution time will not be considered to avoid schedulability test failure, or too much resources uses.

(b) Measurement-based approaches: these methods execute the task on the real target hardware or hardware emulator, for some set of inputs. They then take the measured execution times and derive the maximal and minimal observed execution times on their distribution. These methods have complexity and safety problems [32]. These measurement-based methods require maintaining a history of execution time values of all tasks forming the application. During tasks execution, execution time is measured, and this measurement is subsequently added to the set of previous observations in order to improve the precision. Thus, as the number of observations increases, the estimates produced by a statistical algorithm will be improved [33]. These methods have the advantages that they are able to compensate for data input parameters (such as the problem size) and do not need any direct knowledge of the data characteristics, internal design of the code, or the considered architecture.

In measurement approaches, generally the execution time estimation problem is considered as a regression problem. They use regression algorithms to compute estimates from the set of previous observations. In literature, there are two classes of regression techniques: parametric techniques and nonparametric techniques. In general, parametric techniques require the definition of a form that describes the execution time (Y) of a task as a function (e.g., polynomial regression model) of a parameter X (e.g,, X is the problem size, or number of objects). A popular parametric technique for solving this type of problem is the least squares method. These techniques need offline computation of the function coefficients. However, for flexible and dynamic applications, it is difficult to make any assumptions on the functional form. Parametric techniques are then not well suited to this problem. Nonparametric techniques are a better choice. Nonparametric regression techniques (also called nonparametric estimators or smoothing techniques) are considered to be data driven, since the estimate depends only upon the set of previous observations, and not on any assumptions about $Y(X)$ function. All nonparametric regression techniques compute $Y(X)$ using a variation of the equation

$$Y(X) = \frac{1}{n}\sum_{i=1}^{n} W_i(X) \cdot C_i, \tag{1}$$

where $W_i(X)$ is a weighting function. $Y(X)$ is a weighted average of the execution time values C_i, of the n previous observations. The weight function $W_i(X)$ typically assigns higher weights to observations close to the parameter X, and lower weights to observations farther away from X. This is illustrated in Figure 3. A popular nonparametric technique is k-nearest neighbor (k-NN) algorithm [34].

For nonparametric methods (like k-NN), the main problem is that, especially in embedded systems, they are more complex in implementation (need historic observation, distance calculation).

However, in contrast with a static property of a task, the estimated parameters are not only dependent on the code to be executed by the task (and its possible set of parameters) but also on the system's state and the environment at the

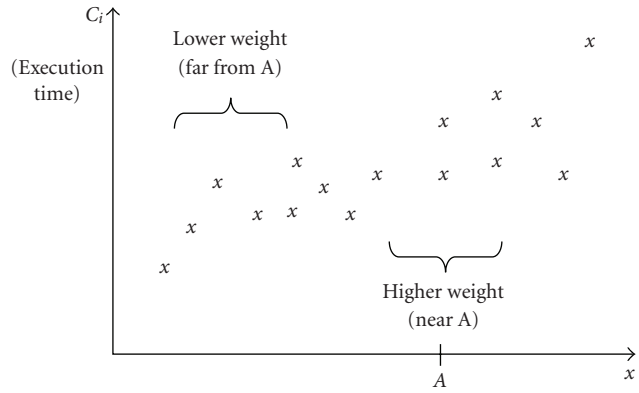


Figure 3: Weights assigning to previous observations.

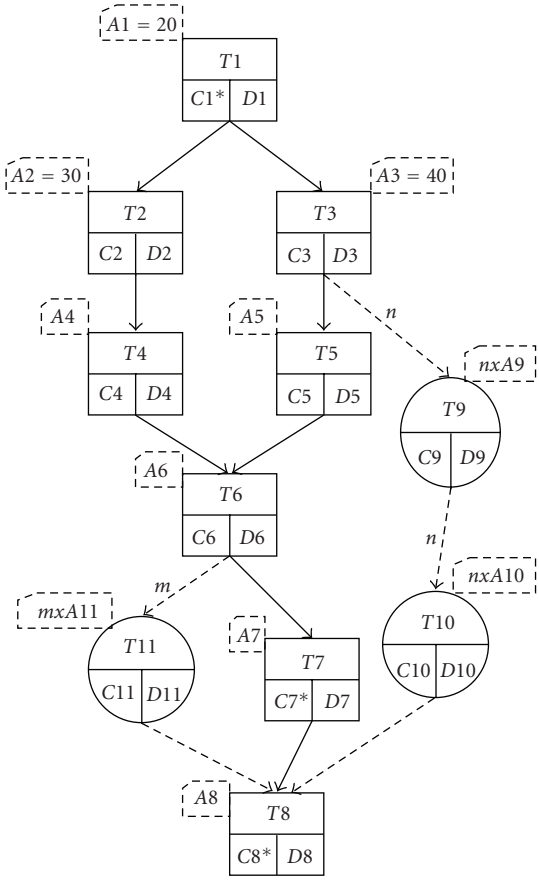


Figure 4: New model for dynamic application.

time (t) when is executed. This means that the estimated parameters are probabilistic values, and they may change over the lifetime of the system. In order to predict those parameters for the next instance (t) of a task T, it should be a good approximation to look at the most recent executions of instances of T. This is the typical case of some real time applications such as image and video processing, where tasks features variations may be correlated with some parameters like keypoints for edge extracting or interesting points for particular processing.

The next section deals with our new modeling method for dynamic applications. The model will permit to the scheduler to take into account the dynamic features and so to help in its predictions.

4. Dynamic Application Modeling

4.1. Proposed Method. As we mentioned in Section 3, to realize a predictive scheduling, we must exhibit the dynamic characteristics of an application. We consider three cases of dynamicity in applications.

(a) The number of tasks is not fixed. It may change from iteration to another.

(b) The tasks execution time may change too.

(c) The number of needed resources for tasks execution is variable. In addition, the number of available resources may decrease after a failure occurs.

For these cases, the goal is to develop a robust scheduling method, that is, little sensible to data uncertainties and variations between theory and practice. To represent all those constraints in the same model, we have developed a graphical model. In this model, the graph is composed of two forms of nodes. We make use of the example shown in Figure 4 in order to illustrate the different definitions corresponding to our model. The first type of nodes refers to permanent tasks which are known in advance and which are always executed during the whole lifecycle execution of the application. This is the case of the set $\{T1, T2, T3, T4, T5, T6, T7, T8\}$. The second type, corresponding to the set $\{T9, T10, T11\}$, is for hazardous tasks which may be executed in some period but not in others. Each task of the proposed model is characterized by the following four parameters:

(a) T_i for the task's number,

(b) C_i for the execution time, and C_i^* for random execution time,

(c) D_i for the deadline,

(d) (A) is the number of needed resources. It may be multiplied by a variable resource factor (n) that represents n hazardous tasks,

(e) plain arc between two nodes represents the precedence between two tasks,

(f) dashed arc between two nodes represents the precedence between several nodes indicated by n. If $n = 0$, then the initial node has no successor.

In our model, to represent tasks with variable execution time, we have been inspired by PERT model presented in Section 3.1. We replace the earliest start time (R_i) in PERT by the execution time (C_i) as it would be changed over execution. We kept the deadline D_i as it will be useful to calculate the makespan (i.e., the length of the schedule). In the example of Figure 4, tasks with variable execution time are $\{T1, T7, T8\}$. This will be noticed by the use of asterisk.

To represent the uncertainty of the number of tasks instances, for example in Figure 4, task $T11$ will be executed

m times, we introduce a valued arc by the number of instances. In order to show that in some cases the task will not be executed (there is no need for that processing), we present the task with circled node (e.g., $T9$, $T10$, and $T11$). Hence, the number of instances is defined as integer, and if the number $m = 0$, then task $T11$ will not occur. The number m will depend on the previous executions and the actual input data to be processed. Thus, m will be recalculated, after each period, based on its previous values.

To be executed, hardware tasks need resources like certain amount of silicon area. This feature is indicated in the labels over each tasks node. For permanent tasks (e.g., $T1$ in Figure 4), the label contains one parameter, that is, $A_1 = 20$ indicating the volume of needed resources for $T1$. For hazardous tasks, and to execute multiple instances of its processes, the volume of resources will be multiplied by the number of instances. For $T9$, the volume of needed resources is $n \times A_9$ of the whole hardware architecture.

The arcs represent the dependencies between tasks. For permanent tasks, arcs are represented with solid lines, while uncertain dependencies are represented by dashed lines.

4.2. Comparison of Models. Compared with other models (Section 3.1), our proposed technique presents several advantages. For uncertain number of occurring tasks, the data flow graph (DFG model) does not contain information about the number of instances. During execution of the application, every task represented by the nodes of DFG is executed once in each iteration [35]. Only when all nodes have finished their executions, a new iteration can start. So to model this, we need to represent n nodes of the same task, which increases the size of the model (see Figure 5(b)). In our model, information about uncertain number of instances of a same task to be executed is noted by the circle form of the task and the number above its arc. For PN model, (see Figure 5(a)), arcs could be labeled with their weights where a k-weighted arc can be interpreted as the set of k parallel arcs [36]. But, from its definition (Section 3.1), weights are positive integers, so PN cannot present a fictive arc with nonfiring transition representing a task that may not be executed in some iterations. In Figure 5(a), if $T9$ and $T10$ are not executed, then n should be null, which is impossible from PN definition. In addition, to fire $T8$, all input places should have at least one token, which will be not possible if $T10$ or $T11$ was not executed (fired).

Petri net does not provide any timing information, that is, mandatory for determining minimum application completion time, latest starting time for an activity which will not delay the system, and so on. The only important aspect of time is the partial ordering of transitions. For example, it presents variable tasks duration with a set of consequent transitions for each task which will complicate the model (see Figure 6). The addition of timing information might provide a powerful new feature for Petri nets but may be difficult to implement in a manner consistent with the basic philosophy of Petri nets research [37]. For resource representation, PN represents this feature by an added place with a fixed number of tokens. To begin execution, a task removes a token from the resource place and puts it back in the end of its execution. However, this model is inadequate in our case since the number of available resources may change over the execution.

Therefore, the use of conventional modeling methods is not effective for dynamic applications. With PN and DFG models (Figure 5), there is no distinction between permanent tasks and hazardous ones (that may not be executed), nor an explicit notion of time (as variable execution time of some tasks). PERT technique enables to present tasks temporal features in order to identify the minimum time needed to complete the total project. However, it lacks functional properties in estimation and does not assume resources constraints (considered as unlimited) [15].

The main advantage of our method is the possibility to represent several dynamic features of real-time applications with the minimum of nodes and thus in a simple formalism. We can bring out three main characteristics of this model:

(a) the distinction between the static execution and the dynamic execution;

(b) the tasks whose execution time is variable (presented by the asterisk);

(c) the volume, for each task, of needed resources for its execution. In each iteration, and depending on the available resources, schedule is able to decide which ready tasks could be executed on the device.

5. Predictive Scheduling

5.1. OLLAF Architecture. Our goal is an efficient management of dynamically reconfigurable architectures; more precisely, as case study, we target the OLLAF architecture. OLLAF, as presented in [4], is an original FGDRA specifically designed to enhance the efficiency of OS (operating system) services necessary to manage such architecture. In particular, OLLAF can efficiency support the context switching service of an OS. From the global view (Figure 7), OLLAF has a reconfigurable logic core organized in columns. Each column can be reconfigured separately and offer the same set of services. A task uses an integer number of columns and can be moved from one column to another without any change on the configuration data [4]. Each column provides a hardware configuration manager (HCM) and a local cache memory (LCM). The reconfigurable logic core uses a double memory plan. With this topology, the context of a task can be shifted in while the previous task is still running and shifted out while the next one is already running. The effective task switching context overhead is then taken down to one clock cycle. To obtain such a rapid context switching service, the configurations of the tasks have to be placed in advance in the LCM of the corresponding columns. In the first version of OLLAF, those memories can store 3 configurations and 3 task contexts.

In OLLAF, as an OS is purely a control process it is implemented on a microprocessor denoted by (HW Sup + HW RTK + CCR) in Figure 7. The OS must manage the context switching and then needs to take into account

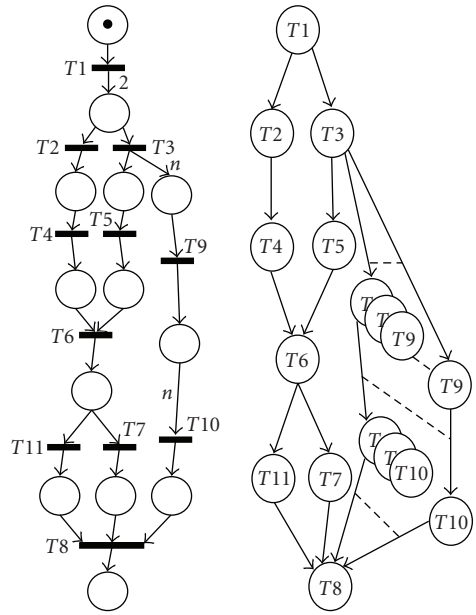


Figure 5: (a) A Petri net representation of the example of Figure 4. (b) A DFG representation of the example of Figure 4.

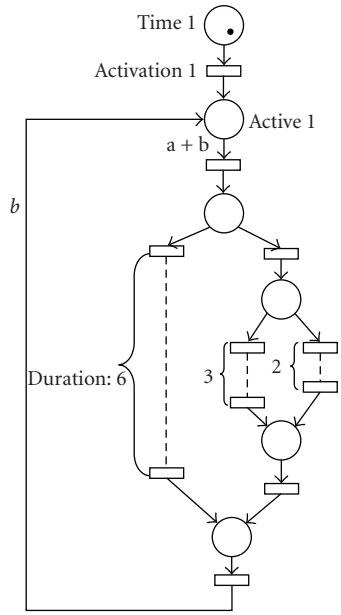


Figure 6: PN model for variable tasks duration.

a prefetch task configuration on the LCM of the columns where it might most probably be placed. It is a conditional process because OLLAF is targeted to execute dynamic application. In [4], authors showed some case studies which demonstrate that the OLLAF architecture can perform a greater efficiency than the one performed using a traditional commercial FPGA. Our goal is then to make a dynamic and predictable scheduling to better manage a dynamic real-time application executed on such architecture. To realize this, we propose to use as the input of the scheduler an application

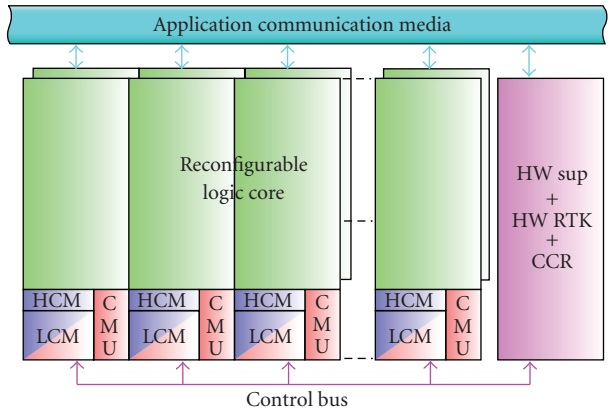


Figure 7: Global view of OLLAF architecture.

represented by our dynamic graph modeling described in Section 4.

5.2. Scheduling Flow. We can represent our scheduler flow as in Figure 8. It takes as inputs on one hand a dynamic graph modeling of an application and on the other hand a description of a DRA, here a model of OLLAF FGDRA as we focus on it. For validate purpose of our scheduling, OLLAF is modeled as a set of columns. Based on these descriptions, our scheduler makes an online dynamic scheduling based on prediction process. The parts which are outlined with dashed lines, visible in Figure 8, present the prediction process which will make online decisions based on precedent observations of variable parameters and taking into account the application constraints.

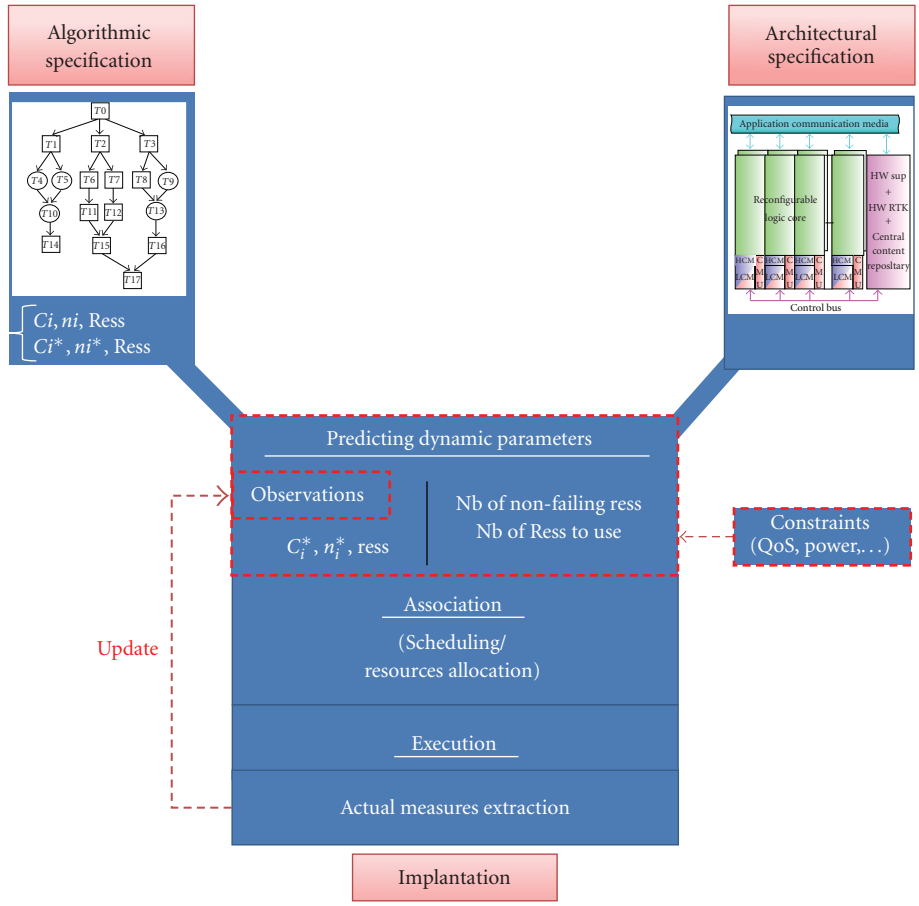


Figure 8: Scheduling flow on OLLAF architecture.

With our model the scheduler is able to distinct between two parts of the application: one containing all the permanent tasks and other containing hazardous tasks, as well as, in permanent tasks, those that have a variable execution time. Permanent tasks will be scheduled respecting their precedence constraints (topological order). Their configuration data will be prefetched, as much as possible, on the columns making a maximum use of the whole of the architecture. This ensures minimizing the number of configuration data transfers and tasks relocation, and thus the time of saving and restoring process. The principle of our scheduler is to realize an initial scheduling where all dynamic features are not taken into account (all dynamic parameters are equal to zero). Then an execution is done and dynamic parameters are updated. Based on these new parameters, a rescheduling is done that takes into account dynamic features. The update of dynamic parameters is computed with a prediction technique. We use a least laxity first (LLF) policy [38] because it permits a dynamic priority assumption depending on the laxity which varies according to the execution time. If there is equality between some tasks, task which has maximum execution time will have priority to be launched before the others.

The scheduler will proceed the rescheduling with a minimum effect on performance. This reschedule must rely on rapid algorithms based on a simple scheduling technique, so that it can perform online execution with no overhead. From the ready list, LLF algorithm determines the tasks that can be executed on the reconfigurable device. Tasks with the higher priorities will be placed first until the area of device is fully occupied. If there are not enough hardware resources available, the last recently used (LRU) strategy is used to select which tasks' configuration data will be removed. During runtime, effective values of dynamic parameters are measured and stored. These values are used to update dynamic parameters with an approximation function. Such function can be, for a parameter, a mean of all its measured values, or a maximum value of all the measured values of a more complex function. In next section, we present four used functions for predicting dynamic parameter values in a robotic application.

6. Experimental Results

6.1. Robotic Application Benchmark. As an illustration of the modeling approach outlined in previous section, consider an image-processing application of a visual system embedded in a mobile robot. This application illustrates the modeling of systems using our proposed model. In this application, robot learns its environment to identify keypoints in the

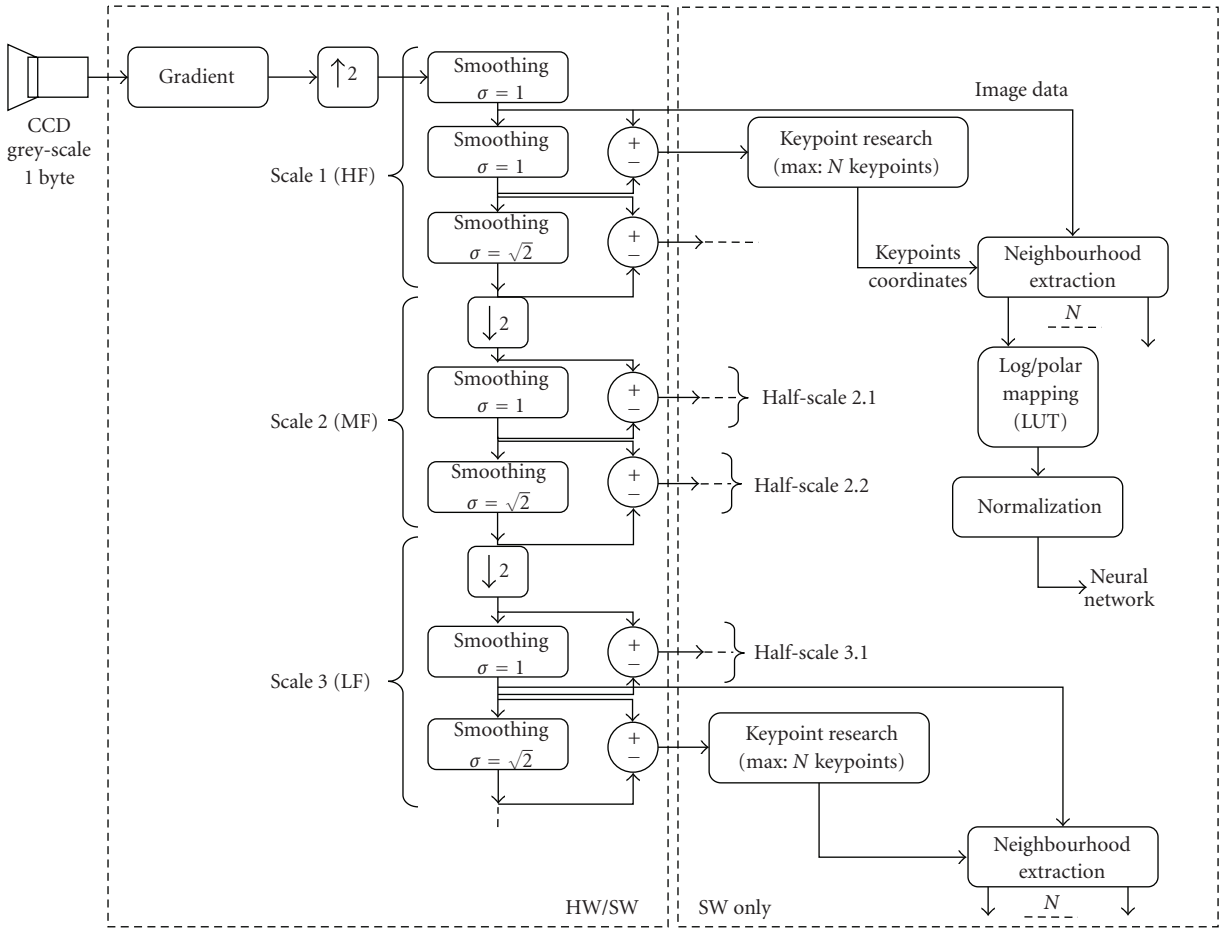


Figure 9: Global architecture of the robotic application.

landscape [39]. The keypoints correspond to the image filtered by difference of Gaussians (DoGs). This application is dynamic in the sense that the number of keypoints depends on the visual scene and is not known a priori. In addition, there are three application modes corresponding to the behavior of the robot, and each mode executes different processing under different rates (Figure 9).

(a) Fast mode (Scale 3): the robot moves around for a coarse description of the landscape but at a high frame rate. Thus, only the lower scales are processed. It is the reason why in this first mode the maximum number of extracted keypoints is fixed to $N = 10$, and the number of frames processed per second (fps) to 20.

(b) Intermediate mode (Scales 2 and 3): the robot moves slowly, for example, when the passage is blocked (obstacle avoidance, door passage) and needs more precision on its environment. In this mode, the middle scale and the lower scale are processed. In this mode, N is fixed to 30 and the system works at a rate of 5 fps.

(c) High-detail mode (Scales 1, 2, and 3): the robot is stopped in a recognition phase (object tracking, new place exploration). All scales are fully processed and full information on the visual environment is provided. The number of processes depends on the number of keypoints in the video frames. For this mode, $N = 120$, and the rate of the system is fixed to 1 fps.

As a consequence, application tasks could be divided in three groups:

(i) intensive data-flow computation tasks that execute in a constant time,

(ii) tasks whose execution number is correlated with the number of interest points,

(iii) tasks with unpredictable execution time (depending on the images features).

Figure 10 shows our proposed model for the robotic vision application. We can notice the presence of permanent branch (squared nodes) which represents permanent tasks that will be executed in all cases or modes (e.g., $T1$, $T14$, $T19$) and hazardous tasks that may occur during execution. Table 1

Table 1: Identification of the robotic application tasks.

Tasks		Permanent tasks	Hazardous tasks
Gradient	$T1$	$T1$, $T2$, $T13$, $T14$, $T15$, $T16$, $T17$, $T18$, $T19$, $T20$, $T21$	$T1$, $T3$, $T4$, $T5$, $T6$, $T7$, $T8$, $T9$, $T10$, $T11$, $T12$, $T22$, $T23$, $T24$, $T25$, $T26$, $T27$, $T28$, $T29$, $T30$
Subsampling	$T2$		
Oversampling	$T4$		
Gauss1	$T3$, $T5$, $T13$, $T14$, $T22$, $T24$		
Gauss2	$T15$, $T6$, $T25$		
DoG	$T7$, $T8$, $T16$, $T17$, $T23$, $T26$		
Search	$T9$, $T11$, $T18$, $T20$, $T27$, $T29$		
Extract	$T10$, $T12$, $T19$, $T21$, $T28$, $T30$		

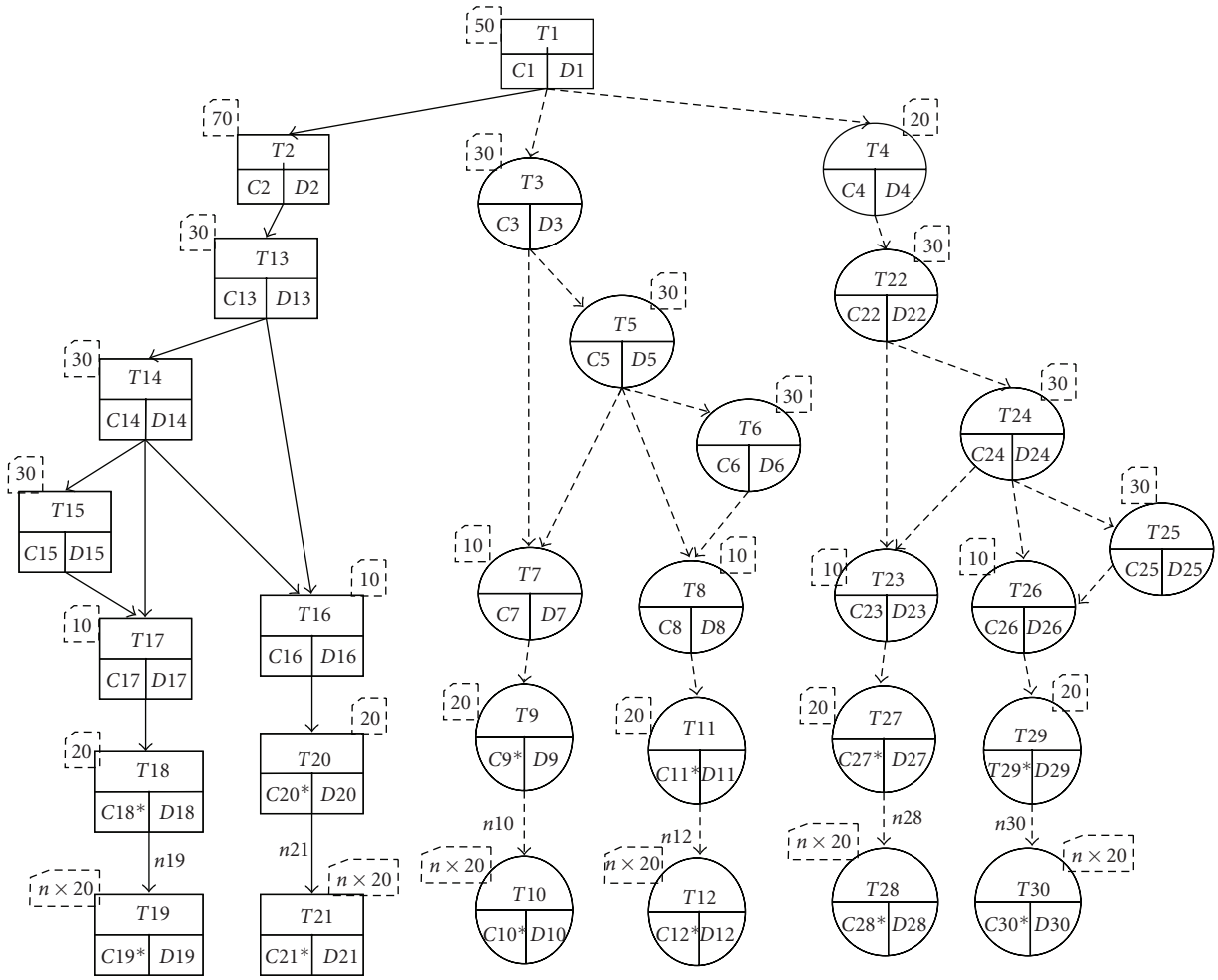


Figure 10: Dynamic model for the robotic vision application.

indicates tasks functions and classifies them to permanent and hazardous. Tasks with unpredictable execution time are indicated by the asterisk (e.g., $T9$, $T11$, $T20$). Another dynamic feature of tasks whose execution number depends on the number of keypoints is indicated by the use of resource factor n (e.g., $T9$, $T10$, $T30$).

For the prediction techniques of uncertain tasks features, we have studied the task of keypoints search. The provided measured execution times on representative samples are presented in Figure 11. As we can see, the values distribution is random especially in the first scale. The elapsed time of keypoints search in an image depends on the number of keypoints that this image contains. The execution time is correlated with the number of keypoints found; this number is also random as shown in Figure 12. Figure 12 shows the number of keypoints found in the corresponding image over

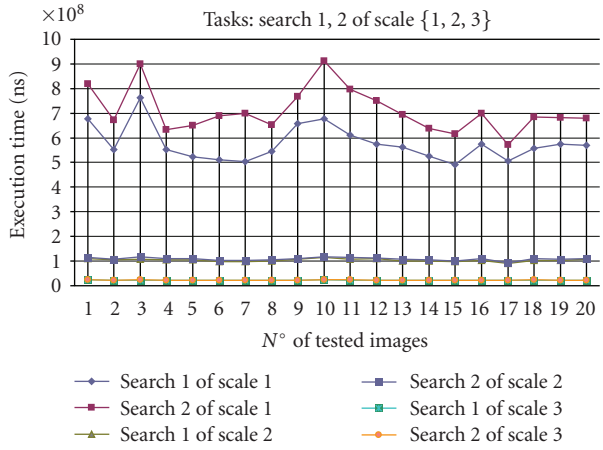


FIGURE 11: Execution time measurement of search task.

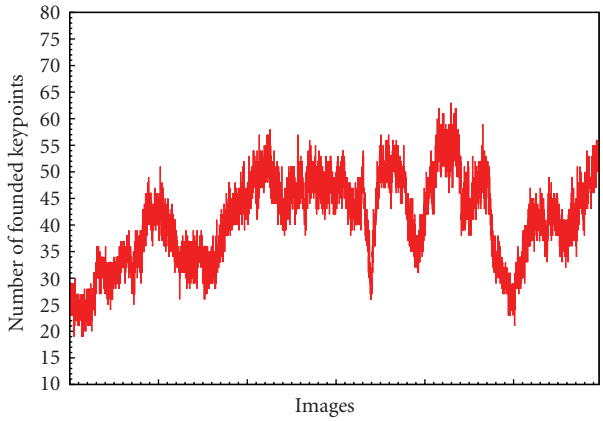


FIGURE 12: Number of detected keypoints in image sequence.

a sequence of more than 5000 images. It corresponds to the search task $T18$ of scale 1, and it has almost the same look as other search tasks $\{T20, T9, T11, T27, T29\}$ of different scales.

6.2. Prediction Results. We have compared several techniques to predict the execution time of the search keypoints task. Comparison is based on error estimation defined as follows:

$$E = \sum_i \frac{\text{Real execution time}\,(t_i) - \text{Estimated execution time}\,(t_i)}{\text{Real execution time}\,(t_i)} \times 100, \tag{2}$$

where t_i is a runtime instant and $i = \{0, \ldots, n\}$.

We will focus on the prediction function corresponding to the search 1 task of scale 1 as it presents very random distributions (Figure 11). We have tested four methods based on statistical measurements obtained by on-line monitoring (Figure 13):

(1) first method (a) uses the mean value of the three last real execution times;

(2) Second method (b) uses the mean of the three last values multiplied by different weights. Weights are determined experimentally on the basis of tested data to minimize estimation error. As in nonparametric regression techniques, the higher weights are assigned to the values closed to the actual parameter and lower weights to those more distant;

(3) the third method (c) takes the maximum of the last three real values;

(4) the fourth method (d) takes the last value increased by 5%.

We notice that all methods lead, in the twenty tested images, to an overestimation of the execution time. As we can see, for the first method after some periods the predicted values become almost constant. In the case of second method, there is a delayed correspondence between the real execution graph and the predicted one. The third methods realize more pessimistic prediction and do overestimations. And finally, the fourth method is closer than the second one with more pessimistic prediction. As seen from Figure 14, the higher error rate is for the first (a) and the third (c) methods. For the fourth method (d), and even with a 1% increase of the last measure value, the error is still greater than that of the second method (b). Hence, the least obtained error percentage is the one related to the method using weighted average of last values (Figure 13(b)). The mean error between estimated and simulated results is about 0.4% for low- and medium-frequency scales and about 2.45% for high-frequency scale. Those results show that the technique based on a weighted average of the execution time values works with an accuracy of almost 98%. Even for keypoints number estimation (Figure 12), the mean estimation error of the same technique is 0.406% for the whole 5000 images. However, a null error rate does not guarantee a good quality of service (QoS). As we can see from the shape of the graphs of Figure 13, some methods (like first one (a)) present a great overestimation leading to overuse of resources, while others have relative great underestimation leading to a decreasing QoS. With keypoints number estimation (Figure 12), we have calculated estimation error of over 1000 successive images for the second method (b) and third one (c). The first technique, based on weighted average prediction, gives less overestimation than the second one (so less useless resources allocation), but it has an almost null mean error leading to a considerable underestimated values which has a great impact on the quality of service than second one. Indeed, the second technique presents more advantage with a relative acceptable mean overestimation of 6.5%, and less than 21% of the whole predicted values were underestimated. The purpose is that prediction be almost near real values and guarantees better compromise between requested QoS and efficient resources management. For this visual system application, and particularly for the two first modes of its process, the latter estimation can assist the prediction engine for better scheduling analysis of real-time tasks and so better

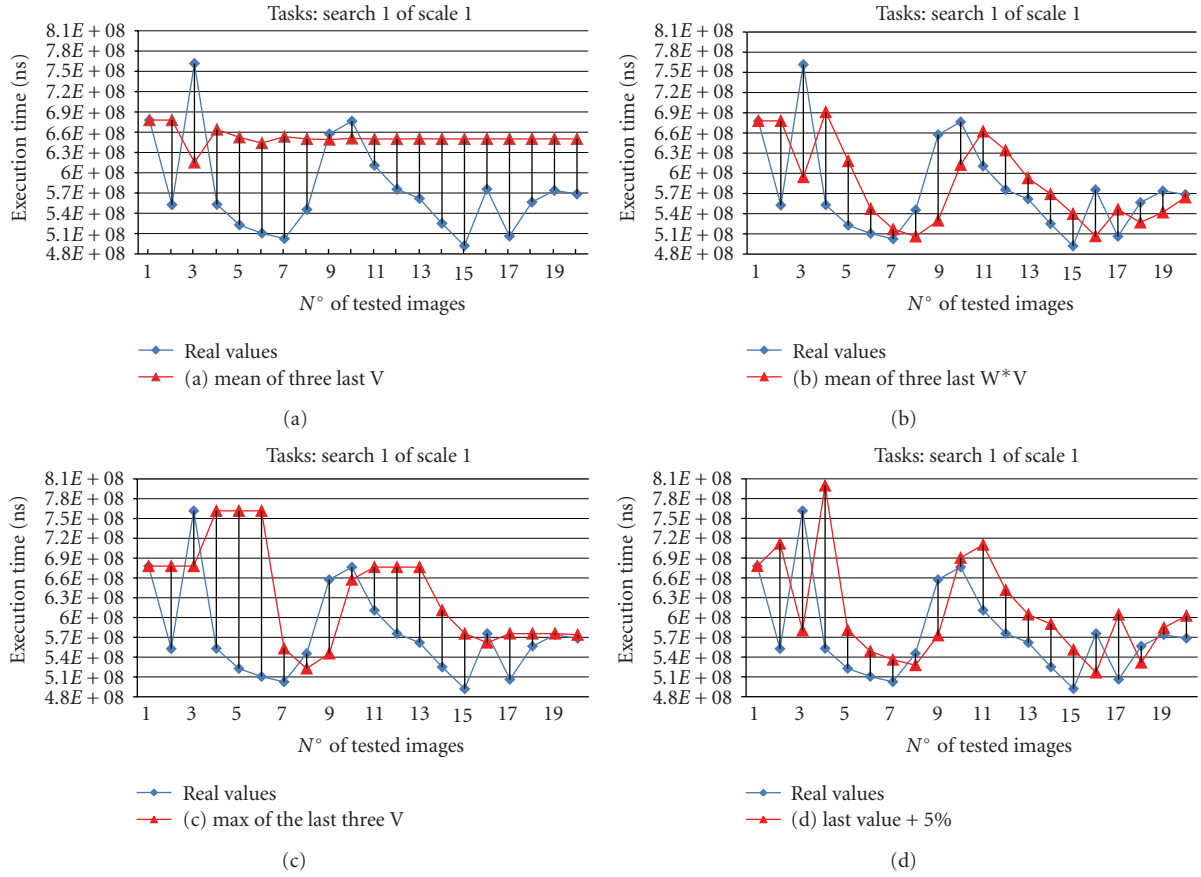


Figure 13: (a) Comparison between real measured values of the execution time of task search 1 of scale 1 and predicted values using method (a). (b) Comparison between real measured values of the execution time of task search 1 of scale 1 and predicted values using method (b). (c) Comparison between real measured values of the execution time of task search 1 of scale 1 and predicted values using method (c). (d) Comparison between real measured values of the execution time of task search 1 of scale 1 and predicted values using method (d).

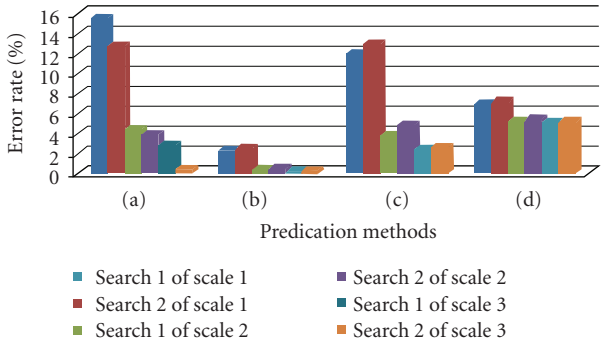


Figure 14: Prediction error rate of different methods for the execution time presented in Figure 11.

exploitation of the DRA like OLLAF. More investigations will be performed for the high-detail mode where all the treatments are realized on high frequencies.

7. Conclusion

In this work, we have proposed a new model to represent application with dynamic features. This model overcomes standard model like dataflow graph or Petri network, where it is not possible to represent dynamic characteristics of an application in a static fashion. We have demonstrated that this model is suitable to the aim of performing predictive scheduling on reconfigurable hardware, in particular on OLLAF FGDRA. The proposed model authorizes a scheduler to take into account uncertain characteristics of hardware tasks, the available resources in the target device, and the quality of service of the application. This model enables to present three considered types of dynamicity, which are uncertain tasks execution time, hazardous tasks that may occur, and variable resources needs. We have presented the scheduling method with the use of a prediction method enabling better adaptation of the architecture to the environment variations. As a validation purpose, we have used a case of an image-processing application of a visual system embedded in a mobile robot. Such application shows dynamically variable characteristics that are uncertain. We have demonstrated that with our modeling we can realize an efficient predictive scheduling on a robot vision application with a mean error of 6.5%.

Future works will consist in integrating our scheduling approach among the services of an RTOS taking into account the new possibilities offered by OLLAF. We also plan to

employ preemptive scheduling policies by using the mechanisms of migration and reallocation of tasks configuration and context data proposed by OLLAF.

References

[1] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[2] J. Noguera and R. M. Badia, "Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 2, pp. 385–406, 2004.

[3] A. Mtibaa, B. Ouni, and M. Abid, "An efficient list scheduling algorithm for time placement problem," *Computers and Electrical Engineering*, vol. 33, no. 4, pp. 285–298, 2007.

[4] S. Garcia and B. Granado, "OLLAF: a fine grained dynamically reconfigurable architecture for os support," *Eurasip Journal on Embedded Systems*, vol. 2009, Article ID 574716, 2009.

[5] P. Brucker and S. Knust, *Complex Scheduling (GOR-Publications)*, Springer, New York, NY, USA, 2006.

[6] V. T'kindt, J.-C. Billaut, and H. Scott, *Multicriteria Scheduling: Theory, Models and Algorithms*, Springer, New York, NY, USA, 2006.

[7] N. González, C. R. Vela, and I. González-Rodrguez, "Comparative study of meta-heuristics for solving flow shop scheduling problem under fuzziness," in *Proceedings of the 2nd International Work-Conference on the Interplay between Natural and Artificial Computation—part I: Bio-Inspired Modeling of Cognitive Tasks*, pp. 548–557, June 2007.

[8] A. J. Davenport and J. C. Beck, "A survey of techniques for scheduling with uncertainty," http://www.eil.utoronto.ca/profiles/chris/chris.papers.html.

[9] P. A. Fishwick, *Handbook of Dynamic System Modeling*, Cpaman & Hall/Crc Computer and Information Science, Chapman & Hall/CRC, 2007.

[10] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, 1999.

[11] L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, A. A. Jerraya, and J. Mermet, "Models of computation for embedded system design," in *System-Level Synthesis*, Kluwer Academic Publishers, 1999.

[12] A. Jantsch, *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*, Morgan Kaufmann, 2003.

[13] L. A. Cortes, *Verification and scheduling techniques for real-time embedded systems*, Ph.D. thesis, Department of Computer and Information Science, Linköping University, 2005.

[14] L. Alej, R. Cortés, P. Eles, and Z. Peng, "A survey on hardware/software codesign representation models," SAVE Project Report, Department of Computer and Information Science, Linköping University, 1999.

[15] H. Kerzner, *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*, John Wiley & Sons, 2003.

[16] C. Rust and F. J. Rammig, "A petri net based approach for the design of dynamically modifiable embedded systems," in *Design Methods and Applications for Distributed Embedded Systems (DIPES)*, pp. 257–266, Kluwer Academic Publishers, 2004.

[17] M. Tavana, "Dynamic process modelling using Petri nets with applications to nuclear power plant emergency management," *International Journal of Simulation and Process Modelling*, vol. 4, no. 2, pp. 130–138, 2008.

[18] F. Rammig and C. Rust, "Modeling of dynamically modifiable embedded real-time systems," in *Proceedings of the IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, vol. 2004, pp. 28–34, 2003.

[19] E. Badouel and J. Oliver, "Reconfigurable nets, a class of high level Petri nets supporting dynamic changes within workflow systems," in *Proceedings of the Workshop on Workflow Management: Net-based Concepts, Models, Techniques, and Tools (WFM '98)*, pp. 129–145, 1998.

[20] H. Aytug, M. A. Lawley, K. McKay, S. Mohan, and R. Uzsoy, "Executing production schedules in the face of uncertainties: a review and some future directions," *European Journal of Operational Research*, vol. 161, no. 1, pp. 86–110, 2005.

[21] W. Herroelen and R. Leus, "Project scheduling under uncertainty: survey and research potentials," *European Journal of Operational Research*, vol. 165, no. 2, pp. 289–306, 2005.

[22] G. E. Vieira, J. W. Herrmann, and E. Lin, "Rescheduling manufacturing systems: a framework of strategies, policies, and methods," *Journal of Scheduling*, vol. 6, no. 1, pp. 39–62, 2003.

[23] D. Ouelhadj and S. Petrovic, "A survey of dynamic scheduling in manufacturing systems," *Journal of Scheduling*, vol. 12, no. 4, pp. 417–431, 2009.

[24] P. B. Luh, F. Liu, and B. Moser, "Scheduling of design projects with uncertain number of iterations," *European Journal of Operational Research*, vol. 113, no. 3, pp. 575–592, 1999.

[25] O. Lambrechts, E. Demeulemeester, and W. Herroelen, "Proactive and reactive strategies for resource-constrained project scheduling with uncertain resource availabilities," *Journal of Scheduling*, vol. 11, no. 2, pp. 121–136, 2008.

[26] S. Liu, K. L. Yung, and W. H. Ip, "Genetic local search for resource-constrained project scheduling under uncertainty," *International Journal of Information and Management Sciences*, vol. 18, no. 4, pp. 347–363, 2007.

[27] M. A. Turnquist and L. K. Nozick, "Allocating time and resources in project management under uncertaint," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS '03)*, 2003.

[28] J. C. Beck and N. Wilson, "Proactive algorithms for job shop scheduling with probabilistic durations," *Journal of Artificial Intelligence Research*, vol. 28, pp. 183–232, 2007.

[29] F. Ghaffari, B. Miramond, and F. Verdier, "Run-time HW/SW scheduling of data flow applications on reconfigurable architectures," *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 976296, 13 pages, 2009.

[30] R. Wilhelm, J. Engblom, A. Ermedahl et al., "The worst-case execution-time problem-overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[31] "Determining bounds on execution times," in *Handbook on Embedded Systems 2005*, R. Wilhelm and R. Zurawski, Eds., CRC Press, 2006.

[32] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 437–455, 2003.

[33] S. Krishnaswamy, S. W. Loke, and A. Zaslavsky, "Estimating computation times of data-intensive applications," *IEEE Distributed Systems Online*, vol. 5, no. 4, 2004.

[34] M. A. Iverson, F. Ozguner, and G. J. Follen, "Run-time statistical estimation of task execution times for heterogeneous distributed computing," in *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, pp. 263–270, August 1996.

[35] O. Sinnen, *Task Scheduling for Parallel Systems*, Wiley Series on Parallel and Distributed Computing, Wiley-Interscience, 2007.

[36] T. Murata, "Petri nets: properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[37] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall PTR, 1981.

[38] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, *Scheduling in Real-Time Systems*, John Wiley & Sons, 2002.

[39] F. Verdier, B. Miramond, M. Maillard, E. Huck, and T. Lefebvre, "Using high-level RTOS models for HW/SW embedded architecture exploration: case study on mobile robotic vision," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, Article ID 349465, 2008.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 836460, 12 pages
doi:10.1155/2011/836460

Research Article

Prime Field ECDSA Signature Processing for Reconfigurable Embedded Systems

Benjamin Glas, Oliver Sander, Vitali Stuckert, Klaus D. Müller-Glaser, and Jürgen Becker

Institute for Information Processing Technology (ITIV), 76131 Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Correspondence should be addressed to Benjamin Glas, glas@kit.edu

Received 27 August 2010; Accepted 10 February 2011

Academic Editor: Gilles Sassatelli

Growing ubiquity and safety relevance of embedded systems strengthen the need to protect their functionality against malicious attacks. Communication and system authentication by digital signature schemes is a major issue in securing such systems. This contribution presents a complete ECDSA signature processing system over prime fields for bit lengths of up to 256 on reconfigurable hardware. By using dedicated hardware implementation, the performance can be improved by up to two orders of magnitude compared to microcontroller implementations. The flexible system is tailored to serve as an autonomous subsystem providing authentication transparent for any application. Integration into a vehicle-to-vehicle communication system is shown as an application example.

1. Introduction

With emerging ubiquity of embedded electronic systems and a growing part of distributed systems and functions even in safety relevant areas, the security of embedded systems and their communication gains importance quickly. One major concern of security is authenticity of communication peers and information exchange. Especially if many different remote participants have to communicate or not all participants are known in advance, asymmetric signature schemes are beneficial for authentication purposes. In contrast to symmetric schemes like the Keyed-Hash Message Authentication Code HMAC [1], asymmetric signature schemes like RSA [2], DSA [3], and the ECDSA scheme [3] considered in this contribution get along without key exchange or predistributed keys, relying usually on a certification authority as trusted third party instead.

This benefit comes at the cost of a much greater computational complexity of these schemes compared to authentication techniques based on symmetric ciphers or solely on hashing. This imposes major problems especially for embedded systems, where resources are scarce.

This contribution presents a hardware-implemented system for complete prime field ECDSA signature processing on FPGAs. It can be integrated as an autonomous subsystem for signature processing in embedded devices. As an application example the integration in a vehicle-to-vehicle communication unit is presented.

The remainder of this paper is organized as follows. In Section 2 some related work is given, Section 3 presents basics of the implemented signature scheme ECDSA, and Section 4 outlines the assumed situation and requirements for the system. The structure and implementation of the signature system itself is presented in Section 5, and Section 6 shows an application example and integration in a wireless communication system. Section 7 details performance and resource usage that are further discussed in Section 8. The paper is concluded in Section 9.

2. Related Work

Since elliptic curves were proposed as basis for public key cryptography in 1985 by Koblitz [4] and Miller [5] independently, many implementations of the prime field Elliptic Curve Digital Signature Algorithm (ECDSA) and Elliptic Curve Cryptography (ECC) in general have been published. Software implementations on general purpose processors

need a lot of computation power. The eBACS ECRYPT benchmark [6] gives values for 256-bit ECDSA of, for example, 1.88 ms for generation and 2.2 ms for verification on an Intel Core 2 Duo at 1.4 GHz and 2.9 ms respectively, 3.4 ms on an Intel Atom 330 at 1.6 GHz. Values for a crypto system based on an ARM7 32-bit microcontroller are given in [7] for a key bit length of 233 bit. Using a comb table precomputation ($w = 4$) 742 ms are needed for a generation and 1240 ms for a verification of an ECDSA signature. An implementation for a RIM Blackberry [8] using an ARM 9EJ-S core realizes 150 ms for a signature generation and 168 ms for a signature verification [9].

To achieve usable throughputs and latencies on embedded systems, various specialized hardware solutions have been proposed, for example, many approaches for implementation of $\mathbb{F}_p$ arithmetic and the ECC primitives point add and point double on reconfigurable hardware. A survey of hardware implementations can be found in [10]. McIvor et al. [11] propose a special ECC processor for $\mathbb{F}_p$ on a Virtex II Pro FPGA, calculating a 256-bit scalar multiplication in 3.86 ms using a clock frequency of 39.5 MHz. Orlando and Paar [12] achieve for a bit length of 192 a scalar multiplication in 3 ms on a Virtex-E FPGA. Güneysu and Paar present in [13] a very fast approach based on special DSP FPGA slices, achieving processing times of 620 μs for a 256-bit scalar multiplication on a Virtex-4 FPGA. The implementation presented here is based on an $\mathbb{F}_p$ ALU presented by Ghosh et al. in [14]. Implementation approaches on CMOS standard cells can be found, for example, in [15, 16], achieving scalar multiplications in 256-bit length in 2.68 ms and 4.3 ms, respectively.

Nevertheless, open implementations of full signature processing units performing complete ECDSA are scarce. Järvinen and Skyttä [17] present a Nios II-based ECDSA system on an Altera Cyclone II FPGA for a key length of 163-bit performing signature generation in 0.94 ms and verification in 1.61 ms.

This contribution presents an FPGA-based autonomous ECDSA system for longer key lengths of 256 bit containing all necessary subsystems for application in embedded systems on reconfigurable hardware.

3. ECDSA Fundamentals

The Elliptic Curve Digital Signature Algorithm (ECDSA) is based on a group structure defined on an elliptic curve E over a finite field $\mathbb{F}_q$. Mostly two types of underlying finite fields are technically used: binary fields $\mathbb{F}_{2^n}$ of characteristic two and prime fields $\mathbb{F}_p$ with large primes p and corresponding characteristic. This paper focuses on prime fields $\mathbb{F}_p$ with characteristic char $(\mathbb{F}_p) \gg 3$. In this case the group E and the respective operation is defined as follows.

Definition 1 (group operation on E). Let E be an elliptic curve over a finite field $\mathbb{F}_p$ of characteristic char $(\mathbb{F}_p) \gg 3$ given by the Weierstrass equation

$$E : y^2 = x^3 + ax + b, \tag{1}$$

Input: Domain parameter $D = (q, a, b, G, n, h)$, secret key d, message m
Output: Signature (r, s)
(1) Chose random $k \in [1, n-1], k \in \mathbb{N}$
(2) Compute $kG = (x_1, y_1)$
(3) Compute $r = x_1 \bmod n$. If $r = 0$ goto step 1.
(4) Compute $e = H(m)$
(5) Compute $s = k^{-1}(e + dr) \bmod n$. If $s = 0$ goto step 1.
(6) **return** (r, s).

Algorithm 1: ECDSA signature generation.

with $a, b \in \mathbb{F}_p$, $4a^3 + 27b^2 \neq 0$, and $P = (x_1, y_1)$, $Q = (x_2, y_2)$ points on E. A group on $E \cup \{\mathcal{O}\}$, $\mathcal{O}$ being the special point at infinity, and the group law

$$+ : E \times E \longrightarrow E, \qquad (P, Q) \longmapsto P + Q =: R = (x_3, y_3) \tag{2}$$

on E is defined by the following

(i) $P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E$.

(ii) For $P = (x_1, y_1) \in E$, the point $P = (x_1, -y_1)$ is also $\in E$ and $P + (-P) = \mathcal{O}$.

(iii) For $P \neq \pm Q$ and $P \neq -P$, the operation for $R = P + Q = (x_3, y_3)$ is given by

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2,$$
$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1. \tag{3}$$

(iv) For $P = Q$ and $P \neq -P$, there is $R = 2P = (x_3, y_3)$ defined by

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1,$$
$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1. \tag{4}$$

The set $E \cup \{\mathcal{O}\}$ with the defined group law $+$ is an abelian group with neutral element $\mathcal{O}$. The inverse to a point $P = (x, y)$ is given by $-P = (x, -y)$.

For the use of ECDSA a set of common domain parameters is needed to be known to all participants. These are the modulus p identifying the underlying field, parameters a, b defining the elliptic curve E used, a base point $G \in E$, the order n of G, and the cofactor $h = \text{order}\ (E)/n$. In addition a cryptographic hash function H is needed. The signature generation and verification for a key pair (Q, d), $Q \in E$ being a point on the curve and d a scalar factor with $Q = dG$, can then be performed using the secret key d or the public key Q, respectively. The procedures needed are shown in Algorithms 1 and 2.

Input: Domain parameter $D = (q, a, b, G, n, h)$,
public key Q, message m, signature (r, s).
Output: Acceptance or Rejection of the signature
(1) **if** $\neg(r, s \in [1, n-1] \cap \mathbb{N})$ **then**
(2) **return** "reject"
(3) **end if**
(4) Compute $e = H(m)$
(5) Compute $w = s^{-1} \bmod n$.
(6) Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
(7) Compute $X = (x_X, y_X) = u_1 G + u_2 Q$.
(8) **if** $X = \infty$ **then**
(9) **return** "reject"
(10) **end if**
(11) Compute $v = x_X \bmod n$.
(12) **if** $v = r$ **then**
(13) **return** "accept"
(14) **else**
(15) **return** "reject"
(16) **end if**

Algorithm 2: ECDSA signature verification.

For identification of the most demanding operations, a tracing of the algorithms based on the hardware implementation presented in Section 5 and the possible parallelization was done. Of the total of 395,521 clock cycles needed for signature generation with the modulus $p256$ used (see Section 4), a percentage of 99.8% or 394,752 cycles were spent computing the scalar multiplication kG. For signature verification the amount of cycles spent for the double scalar multiplication $X = u_1 G + u_2 Q$ is even 99.9%. So in the further consideration we focus on these central operations.

4. Setup and Situation

Objective of a digital signature is to guarantee authenticity and integrity of a signed message to the receiver and prove the identity of the sender, including nonrepudiation. The usual method based on an asymmetric primitive like ECDSA contains three protocol steps. First the sender generates a key pair consisting of a secret signature key SK (d in the ECDSA case) and a public verification key VK (Q for ECDSA) and publishes VK to all possible verifiers. To sign a message m of arbitrary length, the sender generates a digest $H(m)$ of the message using a publicly known cryptographic hash function H. This digest is of a fixed length and can be seen as a fingerprint of the message in the sense that finding a different message $m' \neq m$ with $H(m) = H(m')$ is infeasible. This digest is then signed, meaning *encrypted* using the signing key SK of the sender, and sent along with the original plain text message m. The receiver or verifier is then able to verify the signature by *decrypting* the received hash value using the sender's public verification key PK and comparing the decrypted value to the output of H applied to the received plain message. If the two values match, the signature is positively verified (see Algorithm 3).

The security and correctness of the signature method is based on the assumption that a signed value (encrypted with the secret key) can only be verified (decrypted) with knowledge of the corresponding public key and vice versa and that the secret key cannot be computed from the public key. Secondly the mapping of public keys to identities has to be guaranteed in some way. This is usually done using certification authorities as trusted third parties that verify the identity and issue a certificate for the public key.

We assume an embedded system communicating with several peers which are not entirely known in advance. Therefore, the exchanged signed messages are sent with a certificate attached, that is, issued, to a commonly trusted certification authority. As an example scenario the vehicle-to-vehicle (V2V) communication is considered in Section 6.

This contribution focuses on prime field ECDSA as it is proposed for vehicle-to-vehicle communications which is our general focus application (see also Application Example). Implemented are especially two elliptic curves recommended by the U.S. National Institute of Standards and Technology (NIST) in [18] and Certicom Research in [19], namely, the curves $p224$ (secp224r1) and $p256$ (secp256r1) with bit lengths 224 and 256, respectively, and the corresponding domain parameters also given in the standard.

The proposed system works as a security subsystem exclusively performing signature processing and passing and receiving messages m to and from the external system.

5. Signature Processing System

Processing of ECDSA consists of several layers of computation. On the top level the signature generation and verification algorithms as well as the certificate validation are performed. This signature scheme-dependent layer is based on the group operations point add (PA) and point double (PD) in the underlying elliptic curve. These are in turn based on the underlying finite prime field ($\mathbb{F}_p$) arithmetic, that is, modular arithmetic modulo a prime p. For the main operation of signature verification, the double scalar multiplication $kG + rQ$, the respective number of underlying operations needed on each layer to perform a single operation on the respective upper layer is given in Figure 1. In an even higher layer, there is also the communication protocol to consider at least partially as needed for the signature system.

The architecture and presentation of the system reflects this layering. The two upper layers are implemented as finite state machines (FSM) and make use of a basic $\mathbb{F}_p$ arithmetic logical unit (ALU) and some additional auxiliary modules. Figure 2 outlines the structure of the system. The different building blocks are detailed in the following paragraphs.

5.1. $\mathbb{F}_p$ Modular ALU. The central processing is done by a specialized $\mathbb{F}_p$-ALU for primes of maximum 256-bit length. It is based on the ALU proposed by Ghosh et al. in [14]. Figure 3 depicts the implemented structure. The ALU contains one $\mathbb{F}_p$ adder, subtractor, multiplier, and divider/inverter each. All registers and datapaths between

Input: Sender: Hash function H, secret key SK of sender, message m.
Output: Signed message $(m, \mathrm{Sig}(m))$

(1) Compute hash value $H(m)$ of m.
(2) Compute $\mathrm{Sig}(m) = Enc_{SK}(H(m))$ by encrypting the hash digest $H(m)$ using the sender's secret key.
(3) Send $(m, \mathrm{Sig}(m))$ to receiver.

Input: Receiver: Hash function H, public key PK of sender, received packet (m', x).
Output: Proof that Message m' originates from sender.

(1) Compute $y = Dec_{PK}(x)$ by decrypting the received signature x using the public key of sender.
(2) Compute hash value $H(m')$ of received message m'.
(3) **if** $y == H(m')$ **then**
accept signature
(4) **else**
reject signature
(5) **end if**

Algorithm 3: General digital signature procedure.

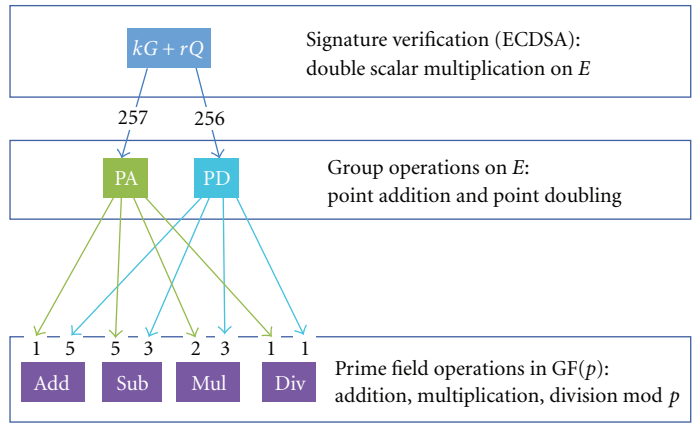


Figure 1: Execution layers of double scalar multiplication on E. On each layer the numbers of operations are given that are needed for a single operation on the respective upper layer.

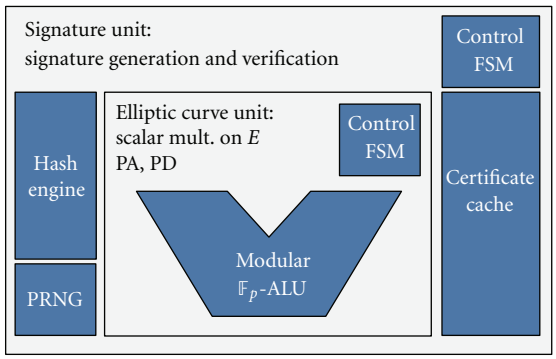


Figure 2: Overview of the signature system.

the modules are 256 bit wide so that complete operands up to 256-bit width (as in the *p256* case) can be stored and transmitted within a single clock cycle. Four inputs, two outputs, and four combined operand/result register as well as a flexible interconnect allow a start of two operations each at the same time as long as they do not use the same basic arithmetic units. The units perform operations independently, so that using different starting points parallel execution in all four subunits is possible. This allows parallelisation especially in the scalar multiplication (see Section 5.2.1).

The $\mathbb{F}_p$-adder and -subtractor perform each operation in a single clock cycle as a general addition/subtraction with subsequent reduction. The $\mathbb{F}_p$ multiplying module computes the modular multiplication iteratively as shift-and-add with reduction mod p in every step. It therefore needs $|p|$ clock cycles for one modular multiplication, $|p|$ being the bit length of the modulus and thereby also the maximum bit length of the operands.

Modular inversion and division is the most complex task of the ALU. It is based on a binary division algorithm on $\mathbb{F}_p$; see [14] for details. The runtime depends on the input values,

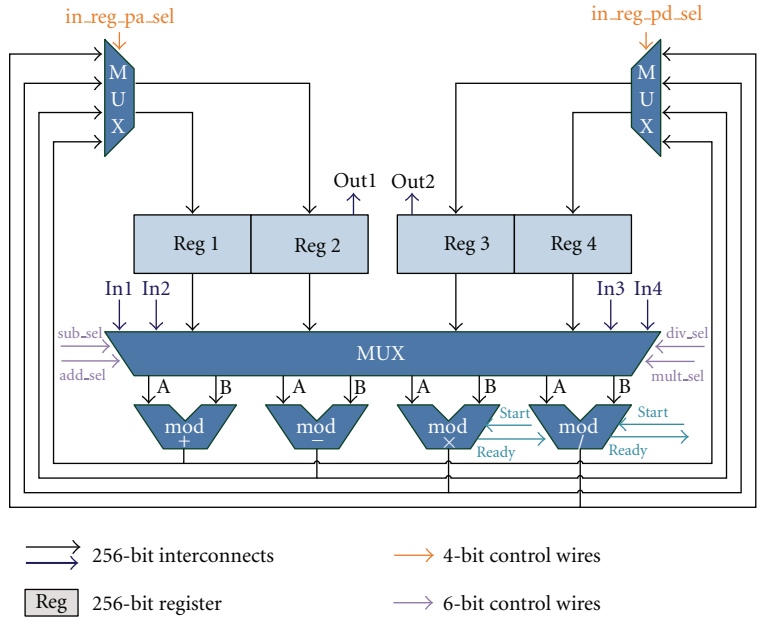


Figure 3: Schematic overview of the $\mathbb{F}_p$ ALU.

Table 1: Hardware execution of point addition.

Step	$\mathbb{F}_p$ unit	No. of cycles
(1) $t_1 = y_2 - y_1$	Sub	1
(2) $t_2 = x_2 - x_1$	Sub	1
(3) $t_2 = t_1/t_2 (= \lambda)$; $t_3 = x_1 + x_2$	div; add	max. $2\lvert p\rvert$
(4) $t_1 = t_2 \cdot t_2$	Mult	$\lvert p\rvert$
(5) $t_1 = t_1 - t_3 (= x_3)$	Sub	1
(6) $t_1 = x_1 - t_1$	Sub	1
(7) $t_1 = t_2 \cdot t_1$	Mult	$\lvert p\rvert$
(8) $t_1 = t_1 - y_1 (= y_3)$	Sub	1
		max. $4\lvert p\rvert + 5$

Table 2: Hardware execution of point doubling.

Step	$\mathbb{F}_p$ unit	No. of cycles
(1) $t_1 = x_1 \cdot x_1$	Mult	$\lvert p\rvert$
(2) $t_2 = t_1 + t_1$	Add	1
(3) $t_1 = t_1 + t_2$	Add	1
(4) $t_1 = t_1 + a$	Add	1
(5) $t_2 = y_1 + y_1$	Add	1
(6) $t_2 = t_1/t_2 (= \lambda)$; $t_3 = x_1 + x_1$	div; add	max. $2\lvert p\rvert$
(7) $t_1 = t_2 \cdot t_2$	Mult	$\lvert p\rvert$
(8) $t_1 = t_1 - t_3 (= x_3)$	Sub	1
(9) $t_1 = x_1 - t_1$	Sub	1
(10) $t_1 = t_2 \cdot t_1$	Mult	$\lvert p\rvert$
(11) $t_1 = t_1 - y_1 (= y_3)$	Sub	1
		max. $5\lvert p\rvert + 7$

maximum runtime being $2\lvert p\rvert$ clock cycles, in the *p256* case therefore up to 512 cycles. Statistical analysis showed an average runtime of $1.5 \cdot \lvert p\rvert$ clock cycles.

ALU control is performed over multiplexer and module control wires and is implemented as a finite state machine presented in the following paragraph. The complete ALU allocates 14256 LUT/FF pairs in a Xilinx Virtex-5 FPGA and allows a maximum clock frequency of 41.2 MHz (after synthesis).

In addition to the 256-bit arithmetic based on the modulus *p256* the ECDSA unit also implements the arithmetic for modulus *p224*. This is done using the same hardware and is also implemented in the overlaying FSM. Theoretically all moduli up to 256-bit width are supported by the ALU. Nevertheless, in the following, all given data refers to the 256-bit key case. Details on resource consumption and performance values are given in Section 7.

5.2. Elliptic Curve Processing. On the elliptic curve E addition of points is defined as group operation. Doubling of a point is specially implemented as it requires a different computation because general point addition is not defined with operands being equal (see Section 3). A comprehensive introduction to elliptic curve arithmetic including algorithms can be found in [20]. To map the algorithms to the implemented specific ALU, the single operation steps have to be scheduled to the respective units. The operation schedules for point addition and point doubling for execution on the ALU are given in Tables 1 and 2.

In the tables, $\lvert p\rvert$ stands for the bit length of the modulus p. In the case of *p256*, this means $\lvert p\rvert = 256$. The execution schedules map the operations to the executing units using

Input: Point $P \in E$; Integer $k = \sum_{i=0}^{l-1} k_i 2^i$ with $k_i \in \{0, 1\}$ and $k_{l-1} = 1$.
Output: Point $Q = kP \in E$.
(1) $P_1 = P$
(2) $P_2 = 2P$
(3) **for** $i = l - 2$ downto 0 **do**
(4) **if** $k = 0$ **then**
(5) $P_1^{\text{new}} = 2P_1^{\text{old}}$
(6) $P_2^{\text{new}} = P_1^{\text{old}} + P_2^{\text{old}}$
(7) **else**
(8) $P_1^{\text{new}} = P_1^{\text{old}} + P_2^{\text{old}}$
(9) $P_2^{\text{new}} = 2P_2^{\text{old}}$
(10) **end if**
(11) **end for**
(12) **return** $Q = P_1$

Algorithm 4: Scalar multiplication in E.

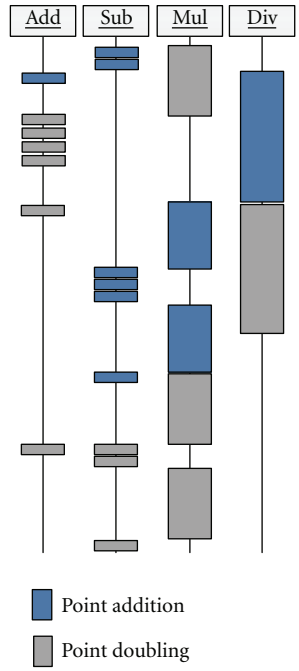


Figure 4: Parallel Scheduling of PA and PD.

three auxiliary register t_1, t_2, t_3 for storing intermediate results. As can be seen in the tables, the third register t_3 is used only once in each point operation and reduces the cycle count in each case by one. If this additional clock cycle is accepted, one 256-bit register can be saved.

5.2.1. Scalar Multiplication on E. Scalar multiplication in step 2 is the central operation of the signature generation of Algorithm 1. Computation is done iteratively using the so-called Montgomery ladder [21, 22] showed in Algorithm 4.

The operations in the branches inside the for-loop, meaning steps 5 and 6 in the if-branch, respectively, steps 8 and 9 in the else-branch, can be executed in parallel. Since it is a point addition and a point doubling each, a real parallel execution on the ALU is possible using a tailored scheduling. Figure 4 depicts the implemented schedule. Although the computation of PA and PD is now done in parallel, a total of five registers for intermediate results is sufficient because the respective t_3 register of PA and PD is not needed at the same time and can therefore be shared.

The execution time using this schedule is $6|p| + 7$ clock cycles for a single pair of point addition and point doubling. Compared to the time of $(4|p| + 5) + (5|p| + 7) = 9|p| + 12$ clock cycles needed for a sequential processing of PA and PD, a performance gain of 33% can be achieved. Execution time for the complete scalar multiplication is therefore at maximum $((|p| - 1) \cdot (6|p| + 7) + (5|p| + 7)) = 6|p|^2 + 6|p|$ clock cycles for the combination of point add and point double.

Input: Point $P, Q \in E$; Integers $k = \sum_{i=0}^{l-1} k_i 2^i$ and $r = \sum_{i=0}^{l-1} r_i 2^i$ with $k_i, r_i \in \{0, 1\}$ and $k_{l-1} \vee r_{l-1} = 1$.
Output: Point $X = kP + rQ \in E$.
(1) Precomputation: $P + Q$
(2) $X = O$ (point at infinity)
(3) **for** $i = l - 2$ downto 0 **do**
(4) $X = 2X$
(5) $X = X + (k_i P + r_i G)$
(6) **end for**
(7) **return** X

Algorithm 5: Simultaneous multiple point multiplication.

5.2.2. Double Scalar Multiplication. For verification of ECDSA signatures two independent scalar multiplications have to be executed (see Algorithm 2, step 7). Instead of computing independently in sequence, it is faster to compute them together using an approach proposed originally by Shamir (see [23]) also known as "Shamir's trick" shown in Algorithm 5.

In contrast to Algorithm 4, the central operations in steps 4 and 5 of Algorithm 5 cannot be parallelized as they depend directly on each other. The maximum time consumption of the algorithm is therefore

$$(4|p| + 5) + |p| \cdot ((5|p| + 7) + (4|p| + 5)) = 9|p|^2 + 16|p| + 5 \quad (5)$$

clock cycles for a double scalar multiplication. This is nevertheless less than the

$$2 \cdot \left(\left(6|p|^2 + 6|p|\right) + (4|p| + 5)\right) = 12|p|^2 + 16|p| + 5 \quad (6)$$

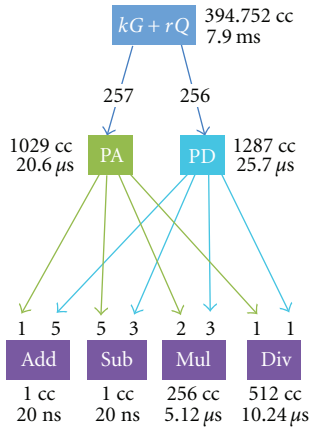


Figure 5: Complexity of execution layers of double scalar multiplication on E. On each level, the number of clock cycles needed for the respective operation is given. The times given refer to a clock frequency of 50 MHz.

cycles that two independent scalar multiplications would consume. Assuming uniform distribution step 5 is omitted in 25% of the cases leaving an estimated runtime of

$$\begin{aligned}&((4|p|+5)+|p|\cdot((5|p|+7)+0.75\cdot(4|p|+5)))\\&\quad=8|p|^2+14.75|p|+5\end{aligned}\tag{7}$$

clock cycles. The composition of the double scalar multiplication on the different levels of computation is shown in Figure 5.

5.3. Signature and Certificate Control System. On top of the elliptic curve (EC) operations and the control FSM performing them, the actual signature algorithms and the certificate verification are implemented. This is done in a separate FSM, controlling the EC arithmetic FSM, some registers, and the auxiliary hashing and random number generation. Figure 6 shows the sequence of operations of the signature verification. See Algorithms 1 and 2 for the implemented procedures.

This FSM is the upmost layer of the signature module and provides a register interface for operands like messages, signatures, certificates, and keys. For integration in an embedded system, it has to be wrapped to support the message format and create the inputs to select the function needed. An example for an integration is given in Section 6.

5.4. SHA2 Hashing Module. The SHA2 hashing unit provides functions SHA-224 and SHA-256 according to the Secure Hash Algorithm (SHA) standard [24]. It is based on a freely available verilog SHA-256 IP-core (available as *SHA IP Core* at http://opencores.com/) adapted with a wrapper performing precomputation of the input data and providing a simple register interface accepting data in 32-bit chunks. In addition the core has been enhanced to support SHA-224.

The unit processes input data in blocks of 512 bit needing 68 clock cycles each at a maximum clock frequency of 120 MHz (after synthesis) and a resource usage of 2277 LUT/FF pairs. After finishing the operation, the result is available in a 256-bit output register.

5.5. Pseudorandom Number Generation. For ECDSA signature generation, a random value k is needed. To provide this k the system incorporates a Pseudorandom Number Generator (PRNG) consisting of two linear feedback shift registers (LFSR), one with 256 bit length, feedback polynomial $x^{255}+x^{251}+x^{246}+1$, and a cycle length of $2^{256}-1$ and a second LFSR with 224 bit length, feedback polynomial $x^{222}+x^{217}+x^{212}+1$, and a cycle length of $2^{224}-1$, both taken from [25].

The LFSR occupies 480 LUT/FF pairs and allows a maximum clocking of 870 MHz although operated in the system in the general system clock of 50 MHz. It is operated continuously to reduce predictability of the produced numbers. The current register content is read out on demand.

For further improvement of the security level, a True Random Number Generator (TRNG) could be integrated. An example implementation of an FPGA-based TRNG can be found in [26].

5.6. Certificate Cache. Usually digital signatures or their respective public keys needed for verification are endorsed by a certificate issued by a trusted third party, a so-called certification authority (CA), to prove its authenticity. Verification of the certificate requires a signature verification itself and is therefore equally complex as the main signature verification of the message. If communicating several messages with the same communication peer using the same signature key, the certificate can be stored hence saving the effort for repetitive verification.

The system incorporates a certificate cache for up to 81 certificates stored in two BRAM blocks. It can be searched in parallel with the signature verification (see Figure 6). Replacement of certificates is performed using a least recently used (LRU) policy.

6. Application Example

The system offers complete ECDSA signature and certificate handling and can be used in a variety of embedded systems seeking authentication and security of communication. As an application example, we show the integration into a vehicle-to-X (V2X) communication system. V2X communication is an emerging topic aiming at information exchange between vehicles on the road and between vehicles and infrastructure like roadside units [27]. This can be used to enhance safety on roads, optimize traffic flow, and help to avoid traffic congestions [28]. Usually two types of broadcasted messages are used, a network beacon sent regularly with a frequency of 2–10 Hz containing status information of the sender and additional event-triggered messages notifying about special events and situations. Latter messages can also be forwarded over several hops to reach receivers outside the direct wireless communication range.

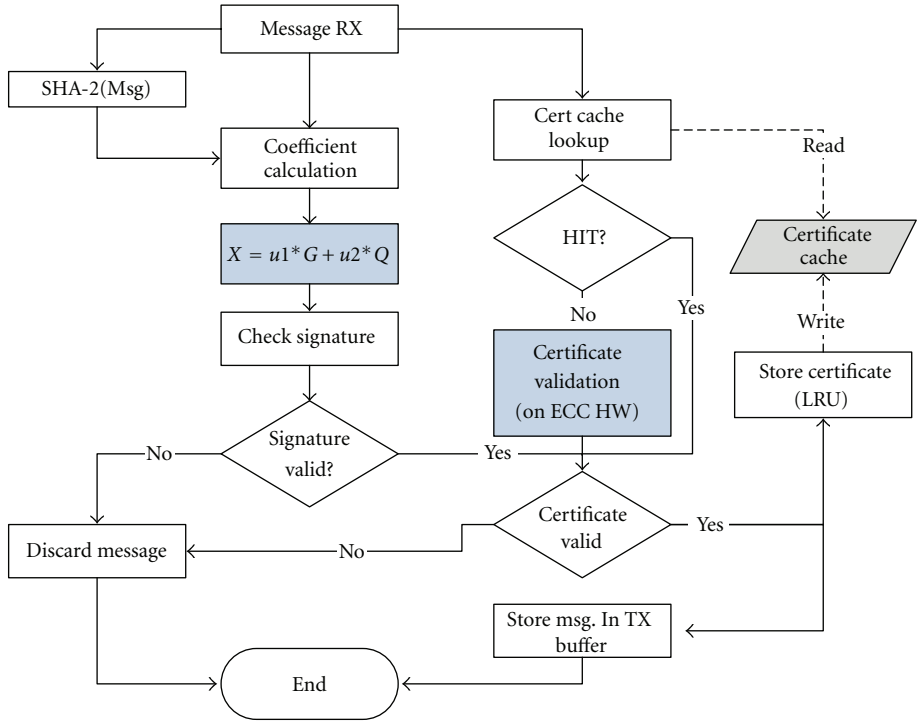


Figure 6: Procedure for signature and certificate verification on the implemented ALU. The blue states mark the main steps.

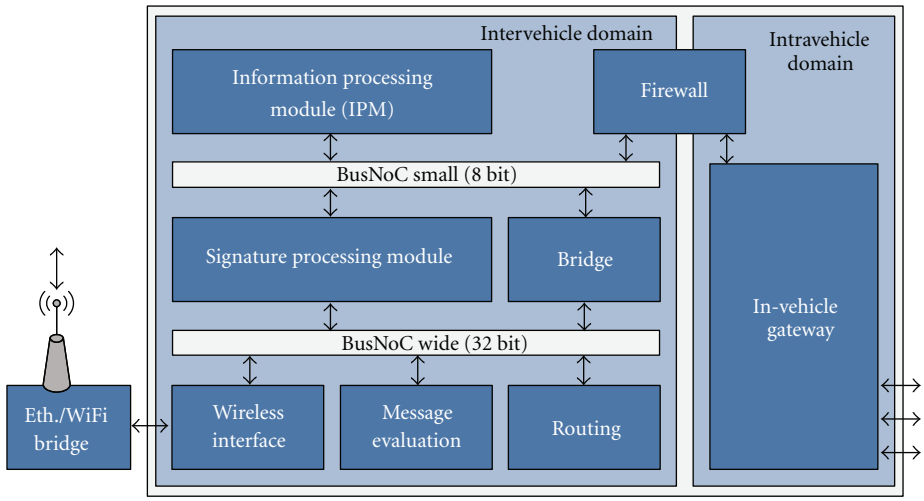


Figure 7: Schematic overview of the V2X-OBU.

To be able to base decisions and applications on information received from other vehicles, trustworthiness of this information is mandatory. To ensure the validity and authenticity of information, signature schemes are used to protect the messages broadcasted by the participating vehicles against malicious attacks [29, 30]. As V2X communication is at present in the process of standardization, no fixed settings are available yet, but the use of ECDSA is proposed in the IEEE 1609 Wireless Access in Vehicular Environments (WAVE) standard draft [31] as well as the proposals of European consortia [32], put together by the COMeSafety project [33].

In the chosen realization V2X communication is performed by a modular FPGA-based On Board-Unit (OBU) presented in [34]; see Figure 7.

It consists of different functional modules connected by a packet-based on-chip communication system [35]. The signature verification system is integrated as a submodule and performs signature handling for incoming and outgoing messages automatically, being therefore transparent to the other modules except for the unavoidable processing latency. It is connected to two different on-chip communication systems, one transmitting unsecured messages over an 8-bit wide communication structure (BusNoC small), and the

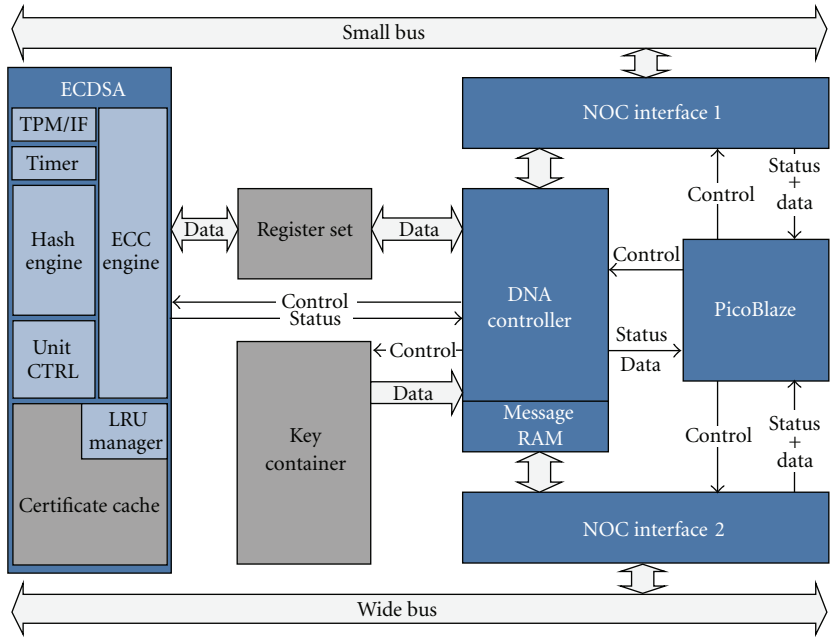


Figure 8: Wrapping of the signature system for V2X integration.

Table 3: Resource usage on an XC5VLX110T with 69.120 LUTs.

	LUT-FF Pairs (synthesis)	Rel. res. usage on FPGA	Max. frequency (MHz)
Signature unit	32.299	46.7%	50
ECDSA unit	24.637	36%	50.1
Hashing unit	2.277	3%	120.8
PRNG	482	0.7%	872.6
$\mathbb{F}_p$-ALU	14.256	20%	41.2
$\mathbb{F}_p$-ADD	858	1.2%	83
$\mathbb{F}_p$-SUB	857	1.2%	92.8
$\mathbb{F}_p$-MUL	2.320	3.4%	42.3
$\mathbb{F}_p$-DIV	5.670	8.2%	73.4

Table 4: Performance of signature verification at 50 MHz.

Verification		secp224r1	secp256r1
Compute time	Worst case	7.23	9.42
(ms/Sig)	Simulated	7.17	9.09
Throughput	Worst case	138	106
(Sig/s)	Simulated	140	110
Latency	Worst case	361.151	471.111
(Cycles/Sig)	Simulated	358.478	454.208

Table 5: Performance of signature generation at 50 MHz.

Generation		secp224r1	secp256r1
Compute time	Worst case	5.56	7.26
(ms/Sig)	Simulated	5.45	7.15
Throughput	Worst case	180	138
(Sig/s)	Simulated	184	140
Latency	Worst case	278.097	362.881
(Cycles/Sig)	Simulated	272.345	357.315

other (BusNoC wide) transmitting only secured messages containing signatures and certificates. These messages are larger because of the additional data, and the latter communication structure is therefore 32 bit wide. A short description of the security system and its system integration is given in [36]. Figure 8 depicts the wrapped signature system with the interfacing to both communication structures.

This interfacing consists of a Direct Network Access (DNA) controller and two interfaces to the Network-on-Chip (NoC) communication structures. An 8-bit PicoBlaze processor controls and configures the components. The DNA controller manages the intramodular procedure and generates the input and control data to the encapsulated ECDSA module. The register set serves as data interface and buffer for intermediate results.

The signature system accepts incoming messages, verifies signatures and certificates, and passes only verified messages on to the Information Processing Module (IPM) for further processing. In case of an invalid signature the outer system (IPM and Routing) is informed. For outgoing messages, signatures are generated, and the corresponding certificate is attached to the message which is then passed on to the wireless interface.

6.1. Key Container. In the V2X environment privacy of participants is of major importance. As messages containing

Table 6: Performance comparison for signature verification and generation for ECDSA on GF(p). Values marked with an asterisk (*) are only for the core operations scalar multiplication and multiple scalar multiplication, respectively, without all pre- and postprocessing and hashing.

	Bit length $\lvert p \rvert$	Hardware resources	Clk (MHz)	Generation (kG)		Verification ($Kg + rQ$)	
				Time	#/s	Time	#/s
Microcontroller implementations							
Drutarovsky and Varchola [7]	233	ARM7	25	742 ms	1.35	1240 ms	0.8
RIM [9]	256	ARM9EJ-S	N.a.	168 ms	5.95	150 ms	6.7
PC processor implementations							
eBACS [6]	256	Motorola PowerPC G4 7410	533	11.7 ms	85.2	14.1 ms	70.7
Petit [37]	256	Intel Pentium D	3400	3.33 ms	300	6.63 ms	151
eBACS [6]	256	Intel Atom 330	1600	2.9 ms	345	3.4 ms	294
eBACS [6]	256	Intel Core 2 Duo U9400	1400	1.88 ms	532	2.2 ms	455
Brown et al. [38]	256	Intel Pentium II	400	*1.67 ms	*599	*6.4 ms	*156
FPGA implementations							
McIvor et al. [11]	256	Xilinx Virtex II Pro, 15755 CLB, 256 MUL	39.5	*3.86 ms	*259	N.a.	N.a.
Orlando and Paar [12]	192	Xilinx Virtex-E, 11416 LUT, 35 BRAM	40	*3 ms	*333	N.a.	N.a.
This paper	256	Xilinx Virtex 5, 14256 LUT/FF pairs	20	7.15 ms	140	9.09 ms	110
ASIC implementations							
Sakiyama et al. [16]	256	243K gates (0.25 μm)	159	*4.3 ms	*233	N.a.	N.a.
Satoh and Takano [15]	256	120K gates (0.13 μm)	138	*2.68 ms	*373	N.a.	N.a.

vehicle type and further information like current position, speed, and heading are continuously broadcasted from twice to up to ten times a second, these messages could easily be used by an eavesdropper to trace participants. To counter such attempts anonymity in the form of pseudonyms is used that are changed on a regular basis. A number of pseudonyms for change are stored directly in the signature module's key container (see Figure 8). It also contains the public keys of trusted certification authorities needed for verification of certificates. The change itself is triggered by a dedicated message sent to the signature processing system by the central information processing module of the C2X system. For all other modules this privacy function is fully transparent as well.

6.2. Caching of Certificates. As V2X communication is not deployed in the fleet so far and also realistic field tests with larger numbers of vehicles are only just beginning (e.g., simTD [39] in Germany), large-scale predictions of message numbers and network behaviour have to be based on simulations and estimations. For an estimation of the expected cache hit rate results from the literature are used. Seada [40] show based on real-world measurements on American freeways that the average communication time between two vehicles is approximately 65 seconds. Based on that and assuming a beaconing frequency of 10 Hz and a sufficient cache size in only one out of 650 messages, the certificate has to be validated. In addition pseudonym change has to be regarded. Papadimitratos et al. [41] propose exchange of pseudonyms every 60 seconds. Assuming stochastical independence of both values, a cache hit rate of 99.68% is possible. Since the communication is regular while the peer vehicle is in range, an LRU strategy is suitable. The required cache size depends strongly on the number of vehicles in range and should therefore be adapted to the expected situations.

7. Resources and Performance

The presented system has been realized using a Xilinx XC5VLX110T Virtex-5 FPGA [42] on a Digilent XUP ML509 evaluation board [43]. The following values refer to an implementation of the complete signature generation and verification unit with interfacing for the application example given previously. Table 3 shows an overview of the resource usage.

After integration of all submodules, the ECDSA unit allows a maximum clock frequency of 50 MHz that has been successfully tested. Table 4 shows signature verification performance values of the ECDSA unit at 50 MHz. Values for signature generation are given in Table 5.

In both tables the *worst case* values given are calculations based on the statistically estimated runtime of the algorithms for scalar multiplication. As these runtimes depend on the operand values, the measured average computation times are different.

Direct comparison of the system's performance is difficult, because implementations of complete ECDSA signature and verification units with certificate handling are scarce. So we can only compare the performance of the GF(p) processing unit, where values are available. Table 6 gives an overview in comparison to some implementations presented already in Section 2.

8. Discussion

The presented system implements the complete ECDSA signature processing in a modular way. As shown in the application example, it can be integrated as an autonomous subsystem to authenticate message traffic and provide verified information to the overlaying system. In comparison to known full implementations (see Section 2), the system's performance of up to 110 verifications per second is by one to two orders of magnitude better than software implementations on microcontrollers, providing sufficient performance for most applications. For high-performance applications like the V2X application example given in detail in Section 6, a still higher throughput of up to 1600 [41, 44], respectively, over 2500 [45] signatures per second is needed though. This can be achieved by a number of optimization steps; see Section 9.

The complete signature module from Section 6 is nevertheless prepared for further improvements. As can be seen in Figure 8, the ECDSA system is encapsulated as a submodule wrapped by the control and communication system that fits to the external system structure. The ECDSA system can therefore easily be replaced by a more performant system without having to adapt the overall system structure.

9. Conclusion and Further Work

We presented a hardware-implemented subsystem for ECDSA signature processing for integration into embedded systems based on reconfigurable hardware. It can be integrated as a stand-alone subsystem performing transparent authentication functionality for communication systems. Applicability of the system has been shown using vehicle-to-X communication as a practical example.

The performance values presented in Section 7 are sufficient for applications like entry control systems or electronic payment, where the number of communication peers is small. For V2X communication even larger throughput is necessary. Further work therefore includes speeding up the computation. Promising approaches that are subject to ongoing work here are the use of windowing techniques on algorithmic level, the tailored use of optimized representations like projective coordinates on mathematical level, and the speedup of the field operations on implementation level, for example, by the use of hardware multipliers. Also the use of low-cost FPGAs and reduction of the footprint is required for the use in embedded systems.

References

[1] FIPS, "Pub 197: Advanced Encryption Standard (AES)," Federal information processing standards publication, U.S. Department of Commerce, Information Technology Laboratory (ITL), National Institute of Standards and Technology (NIST), Gaithersburg, Md, USA, 2001.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[3] FIPS, *Pub 186-3: Digital signature standard (dss)*, Federal information processing standards publication, U.S. Department of Commerce, Information Technology Laboratory, National Institute of Standards and Technology (NIST), Gaithersburg, Md, USA, 2009.

[4] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[5] V. S. Miller, "Use of elliptic curves in cryptography," in *Proceedings of the Advances in Cryptology (CRYPTO '85)*, pp. 417–426, 1986.

[6] eBACS, "ECRYPT Benchmarking of Cryptographic Systems," 2010, http://bench.cr.yp.to/ebats.html.

[7] M. Drutarovsky and M. Varchola, "Cryptographic system on a chip based on actel ARM7 soft-core with embedded true random number generator," in *Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS '08)*, pp. 164–169, IEEE Computer Society, Washington, DC, USA, 2008.

[8] RIM Blackberry 7230, "Datasheet," 2010, http://pdadb.net/index.php?m=specs&id=1467&view=1&c=rim.

[9] D. Hankerson, "Implementing elliptic curve cryptography (a narrow survey)," in *Proceedings of the Workshop in Implementation of Cryptographic Methods (WIMC '05)*, 2005.

[10] G. Meurice de Dormale and J. J. Quisquater, "High-speed hardware implementations of Elliptic Curve Cryptography: a survey," *Journal of Systems Architecture*, vol. 53, no. 2-3, pp. 72–84, 2007.

[11] C. J. McIvor, M. McLoone, and J. V. McCanny, "Hardware elliptic curve cryptographic processor over GF(p)," *IEEE Transactions on Circuits and Systems I*, vol. 53, no. 9, pp. 1946–1957, 2006.

[12] G. Orlando and C. Paar, "A scalable gf(p) elliptic curve processor architecture for programmable hardware," in *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES '01)*, vol. 2162 of *Lecture Notes in Computer Science*, pp. 348–363, 2001.

[13] T. Güneysu and C. Paar, "Ultra high performance ecc over nist primes on commercial fpgas," in *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES '08)*, Lecture Notes in Computer Science, pp. 62–78, Washington, DC, USA, 2008.

[14] S. Ghosh, M. Alam, I. S. Gupta, and D. R. Chowdhury, "A robust GF(p) parallel arithmetic unit for public key cryptography," in *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD '07)*, pp. 109–115, Washington, DC, USA, 2007.

[15] A. Satoh and K. Takano, "A scalable dual-field elliptic curve cryptographic processor," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 449–460, 2003.

[16] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede, "Multicore curve-based cryptoprocessor with reconfigurable modular arithmetic logic units over GF (2^n)," *IEEE Transactions on Computers*, vol. 56, no. 9, pp. 1269–1282, 2007.

[17] K. Järvinen and J. Skyttä, "Cryptoprocessor for Elliptic Curve Digital Signature Algorithm (ECDSA)," Tech. Rep., Helsinki University of Technology, Signal Processing Laboratory, 2007.

[18] NIST, "Recommended elliptic curves for federal government use," Tech. Rep., National Institute of Standards and Technology, U.S. Department of Commerce, 1999.

[19] Certicom Research, "Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters," 2000.

[20] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, New York, NY, USA, 2004.

[21] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985.

[22] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 177, pp. 243–264, 1987.

[23] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, 1985.

[24] FIPS, *Pub 180-2: Secure hash standard (shs)*, Federal information processing standards publication, U.S. Department of Commerce, National Institute of Standards and Technology (NIST), Gaithersburg, USA, 2002.

[25] R. Ward and T. Molteno, "Table of Linear Feedback Shift Registers," 2007, http://www.otagophysics.ac.nz/px/research/electronics/papers/technical-reports/lfsr_table.pdf.

[26] D. Schellekens, B. Preneel, and I. Verbauwhede, "FPGA vendor agnostic true random number generator," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 139–144, August 2006.

[27] "CAR 2 CAR Communication Consortium, *Manifesto—Overview of the C2C-CC System v1.1*," 2007.

[28] Commission of the European Communities, "European transport policy for 2010: time to decide," white paper com 370 final, 2001, http://ec.europa.eu/transport/strategies/doc/2001_white_paper/lb_com_2001_0370_en.pdf.

[29] P. Papadimitratos, L. Buttyan, T. Holczer et al., "Secure vehicular communication systems: design and architecture," *IEEE Communications Magazine*, vol. 46, no. 11, pp. 100–109, 2008.

[30] F. Kargl, P. Papadimitratos, L. Buttyan et al., "Secure vehicular communication systems: design and architecture," *IEEE Communications Magazine*, vol. 46, no. 11, pp. 110–118, 2008.

[31] IEEE Vehicular Technology Society, ITS Committee, *IEEE Trial-Use Standard for Wireless Access in Vehicular Environments (WAVE)—Security Services for Applications and Management Messages*, 2006.

[32] COMeSafety Project, "European ITS Communication Architecture—Overall Framework," 2008, http://www.comesafety.org/.

[33] COMeSafety Project—Communication for eSafety, "Project website," 2010, http://www.comesafety.org/.

[34] O. Sander, B. Glas, C. Roth, J. Becker, and K. D. Müller-Glaser, "Design of a vehicle-to-vehicle communication system on reconfigurable hardware," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '09)*, pp. 14–21, IEEE, 2009.

[35] O. Sander, B. Glas, C. Roth, J. Becker, and K. D. Müller-Glaser, "Priority-based packet communication on a bus-shaped structure for FPGA-systems," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 178–183, April 2009.

[36] B. Glas, O. Sander, V. Stuckert, K. D. Müller-Glaser, and J. Becker, "Car-to-car communication security on reconfigurable hardware," in *Proceedings of the IEEE 69th Vehicular Technology Conference (VTC '09)*, Barcelona, Spain, 2009.

[37] J. Petit, "Analysis of ecdsa authentication processing in vanets," in *Proceedings of the 3rd International Conference on New Technologies, Mobility and Security (NTMS '09)*, pp. 388–392, IEEE Press, Piscataway, NJ, USA, 2009.

[38] M. Brown, D. Hankerson, and A. Menezes, "Software implementation of the nist elliptic curves over prime fields," in *Proceedings of the Topics in Cryptology (CT-RSA '01)*, vol. 2020 of *Lecture Notes in Computer Science*, pp. 250–265, Springer, Berlin, Germany, 2001.

[39] simTD, "Sichere Intelligente Mobilität: Testfeld Deutschland. Project webpage," 2008.

[40] K. Seada, "Insights from a freeway car-to-car real-world experiment," in *Proceedings of the 3rd ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WiNTECH '08)*, pp. 49–55, ACM, New York, NY, USA, 2008.

[41] P. Papadimitratos, G. Calandriello, J. P. Hubaux, and A. Lioy, "Impact of vehicular communications security on transportation safety," in *Proceedings of the IEEE INFOCOM Workshops*, April 2008.

[42] Xilinx Inc., *UG190: Virtex-5 FPGA User Guide. v5.2*, 2009.

[43] Xilinx Inc., *UG347: ML505/ML506/ML507 Evaluation Platform—User Guide, 2009. v3.1.1*, 2009.

[44] Q. Xu, T. Mak, J. Ko, and R. Sengupta, "Vehicle-to-vehicle safety messaging in DSRC," in *Proceedings of the 1st ACM International Workshop on Vehicular Ad Hoc Networks (VANET'04)*, pp. 19–28, October 2004.

[45] M. Torrent Moreno, *Inter-vehicle communications: achieving safety in a distributed wireless environment*, Dissertation, Shaker, 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 121404, 18 pages
doi:10.1155/2011/121404

Research Article

Exploration of Heterogeneous FPGA Architectures

Umer Farooq,[1] Husain Parvez,[1] Habib Mehrez,[1] and Zied Marrakchi[2]

[1] *LIP6, UPMC, 75005 Paris, France*
[2] *FLEXRAS Technologies, 93521 Saint-Denis Cedex, France*

Correspondence should be addressed to Umer Farooq, umer.farooq@lip6.fr

Received 27 August 2010; Revised 11 January 2011; Accepted 10 February 2011

Academic Editor: Michael Hübner

Mesh-based heterogeneous FPGAs are commonly used in industry and academia due to their area, speed, and power benefits over their homogeneous counterparts. These FPGAs contain a mixture of logic blocks and hard blocks where hard blocks are arranged in fixed columns as they offer an easy and compact layout. However, the placement of hard-blocks in fixed columns can potentially lead to underutilization of logic and routing resources and this problem is further aggravated with increase in the types of hard-blocks. This work explores and compares different floor-planning techniques of mesh-based FPGA to determine their effect on the area, performance, and power of the architecture. A tree-based architecture is also presented; unlike mesh-based architecture, the floor-planning of heterogeneous tree-based architecture does not affect its routing requirements due to its hierarchical structure. Both mesh and tree-based architectures are evaluated for three sets of benchmark circuits. Experimental results show that a more flexible floor-planning in mesh-based FPGA gives better results as compared to the column-based floor-planning. Also it is shown that compared to different floor-plannings of mesh-based FPGA, tree-based architecture gives better area, performance, and power results.

1. Introduction

During recent past, embedded hard blocks (HBs) in FPGAs (i.e., heterogenous FPGAs) have become increasingly popular due to their ability to implement complex applications more efficiently as compared to homogeneous FPGAs. The work in [1] shows that the use of embedded memory in FPGA improves its density and performance. Beauchamp et al. [2] have incorporated floating point multiply-add units in the FPGA and have reported significant area and speed improvements over homogeneous FPGAs. Ho et al. [3] have proposed a virtual embedded block (VEB) methodology that predicts the effects of embedded blocks in commercial FPGA devices, and they have shown that the use of embedded blocks causes an improvement in area and speed efficiencies. Also Govindu et al. [4] and Underwood and Hemmert [5] suggest the use of embedded blocks in FPGAs for better performance regarding complex scientific applications. The work in [6] shows that the use of HBs in FPGAs reduces the gap between ASIC and FPGA in terms of area, speed and power consumption. Some of the commercial FPGA vendors like Xilinx [7] and Altera [8] are also using HBs (e.g., multipliers, memories, and DSP blocks).

Almost all the work cited above considers mesh-based (island-style) FPGAs as the reference architecture where HBs are placed in fixed columns; these columns of HBs are interspersed evenly among columns of configurable logic blocks (CLBs). The main advantage of island-style, column-based heterogeneous FPGA lies in its simple and compact layout generation. When tile-based layout for an FPGA is required, the floor-planning of similar type blocks in a column simplifies the layout generation. The complete width of the entire column, having same type of blocks, can be adjusted appropriately to generate a very compact layout. However, the column-based floor-planning of FPGA architectures limits each column to support only one type of HB. Due to this limitation, the architecture is bound to have at least one separate column for each type of HB even if the application or a group of applications that is being mapped on it uses only one block of that particular type. This can eventually result in the loss of precious logic and routing resources. This loss can become even more severe with the

increase in the number of types of blocks that are required to be supported by the architecture.

In order to reduce the amount of useless resources and increase the area density of FPGA architecture, in this work, we explore different floor-planning techniques of mesh-based FPGA. Although some domain-specific architectures [9–11], targeting multimedia and DSP domains and having a wide range of routing architectures, are proposed to address the problem of useless logic and routing resources, these architectures cannot be related to FPGA architectures as domain-specific architectures use a different routing structure compared to FPGAs. Also, unlike previous research [1–6] that mainly compares heterogeneous mesh-based FPGA architectures with their homogeneous counterparts, this work presents a detailed comparison between heterogeneous mesh- and tree-based architectures. Contrary to mesh-based architecture, a tree-based architecture is a hierarchical architecture where logic and routing resources are arranged in a multilevel clustered structure. A comparison between homogeneous mesh- and tree-based FPGA architectures was presented in [12] and in this work we extend that comparison to their heterogeneous counterparts.

Mainly six floor-planning techniques are explored for mesh-based architecture, four of which are column-based and two are non column-based. In order to evaluate these floor-planning techniques, this work also compares a tree-based heterogenous FPGA architecture [13] with different floor-planning techniques of mesh-based heterogeneous FPGA architecture. Contrary to mesh-based heterogenous FPGA, routability of a tree-based FPGA is independent of its floor-planing and the number of types of HBs required to be supported by the architecture. So, tree-based heterogenous FPGA can be advantageous as compared to mesh-based FPGA. Two different techniques are explored for tree-based architecture. First technique respects the symmetry of hierarchy, which is one of the characteristics of tree-based architectures. However, in order to ensure the optimal use of available resources, the second technique does not respect the symmetry of hierarchy. The details of the architectures under consideration and their respective techniques are presented in the sections that follow.

The remainder of the paper is organized as follows: Section 2 gives a brief overview of mesh- and tree-based architectures. Section 3 presents exploration environments and the floor-planning techniques that are explored using these environments. Section 4 describes the experimental flow. Section 5 presents experimental results, and Section 6 finally concludes the paper.

2. Reference FPGA Architectures

2.1. Mesh-Based Heterogenous FPGA Architecture. A mesh-based heterogeneous FPGA is represented as a grid of equally sized slots which is termed as slot-grid. Blocks of different sizes can be mapped on the slot-grid. A block can be either a soft-block like a configurable logic block (CLB) or a hard-block like multiplier, adder, RAM, and so forth, Each block (CLB or a HB) occupies one or more slots depending upon its size. The architecture used in this work is a VPR-style

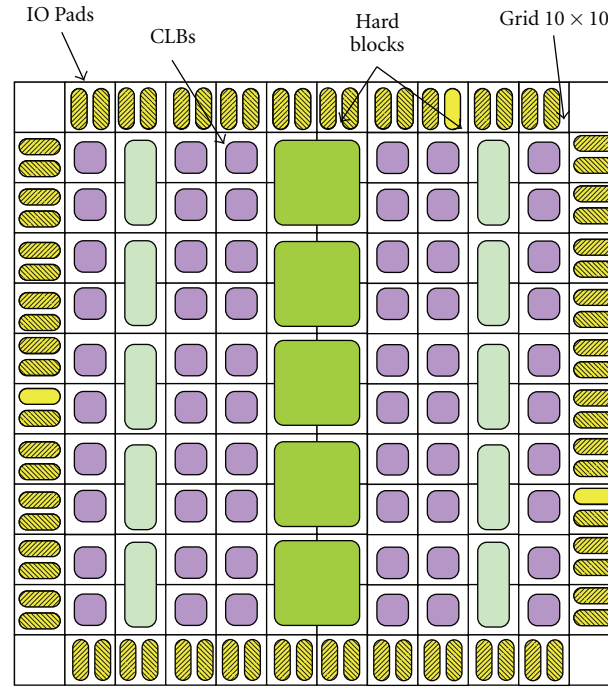


Figure 1: Mesh-based heterogeneous FPGA architecture.

(Versatile Place & Route) [14] architecture that contains CLBs, I/Os, and HBs that are arranged on a two-dimensional grid. In order to incorporate HBs in a mesh-based FPGA, the size of HBs is quantized with the size of the smallest block of the architecture, that is, CLB. The width and height of an HB are therefore a multiple of the width and height of the smallest block in the architecture. An example of such FPGA where CLBs and HBs are mapped on a grid of size 10×10 is shown in Figure 1. In mesh-based FPGA, input and output pads are arranged at the periphery of the slot-grid as shown in Figure 1. The position of different blocks in the architecture depends on the used floor-planning technique. A block (referred as CLB or HB) is surrounded by a uniform length, single driver, unidirectional routing network [15]. The input and output pins of a block connect with the neighboring routing channel. In the case where HBs span multiple tiles, horizontal and vertical routing channels are allowed to cross them [2].

An FPGA tile showing the detailed connection of a CLB with its neighboring routing network is shown in Figure 2. In this figure, 4 inputs of the CLB are connected to 4 adjacent routing channels. The output pin of the CLB is connected to the routing channel on its top and right through the diagonal connections of the switch box. The switch box uses unidirectional, disjoint topology to connect different routing tracks together. The connectivity of a routing track incident on a switch block with routing tracks of other routing channels that are incident on the same switch block, termed as switch block flexibility (Fs), is set to be 3. The connectivity of the routing channel with the input and output pins of a block, abbreviated as Fcin and Fcout, is set to be 1.

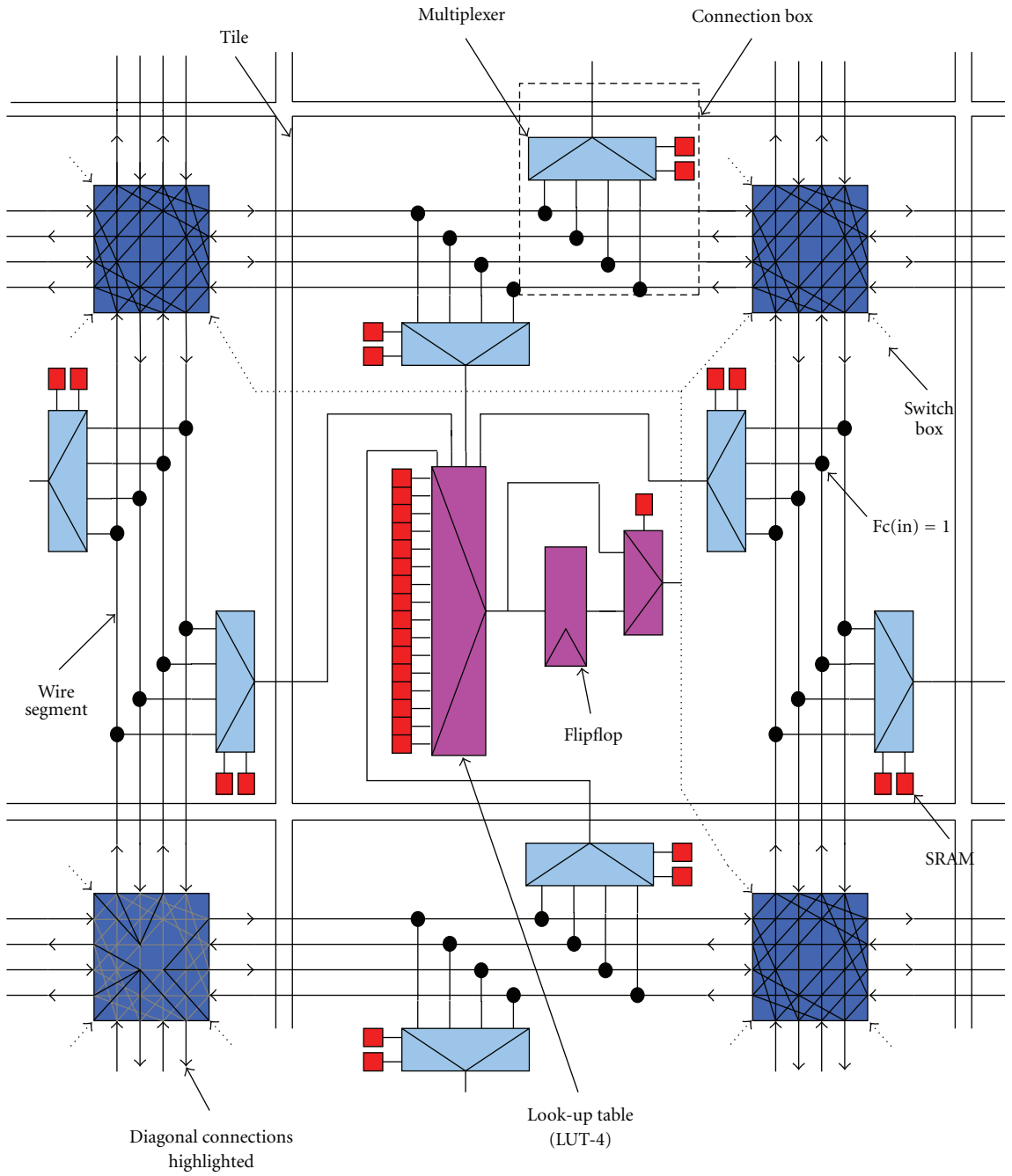


Figure 2: Detailed interconnect of a CLB with its neighboring channels.

The channel width is varied according to the netlist requirement but remains a multiple of 2 [15].

2.2. Tree-Based Heterogeneous FPGA Architecture. A tree-based architecture is a hierarchical architecture having unidirectional interconnect. Unlike mesh-based architecture where logic and routing resources are arranged in island-style, in a tree-based architecture, logic and routing resources are arranged in hierarchical manner. Tree-based architecture exploits the locality of connections that is inherent in most of the application designs. In this architecture, CLBs, I/Os, and HBs are partitioned into a multilevel clustered structure where each cluster contains subclusters and switch blocks allow to connect external signals to subclusters. Figure 3 shows generalized example of a heterogeneous tree-based architecture used in this work. In a heterogenous tree-based architecture, CLBs and I/Os are placed at the bottom of hierarchy whereas HBs can be placed at any level of hierarchy to meet the best design fit. For example, in Figure 3 HBs are placed at level 2 of hierarchy.

Tree-based architecture contains two unidirectional, single length, interconnect networks: a downward network and

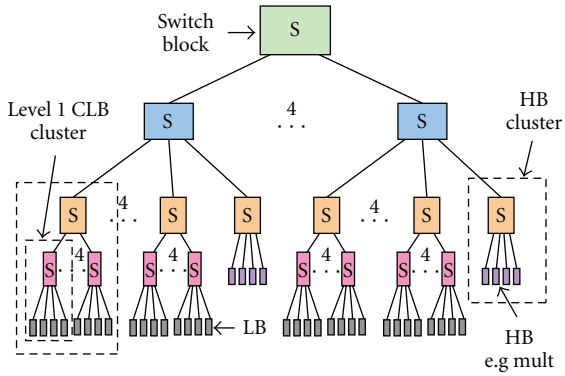


Figure 3: Tree-based heterogeneous FPGA architecture.

an upward network. downward network is based on butterfly fat tree topology and allows to connect signals coming from other clusters to its subclusters through a switch block. The upward network is based on hierarchy and it allows to connect sub-cluster outputs to other subclusters in the same cluster and to clusters in other levels of hierarchy. A level 1 cluster example of two interconnect networks, demonstrating the detailed connection of a CLB with its neighboring CLBs, is shown in Figure 4. It can be seen from the figure that switch blocks are further divided into downward and upward miniswitch boxes (DMSBs & UMSBs) where DMSBs are responsible for downward interconnect and UMSBs are responsible for upward interconnect. DMSBs and UMSBs are combined together to route different signals of the netlists that are mapped on the architecture. These DMSBs and UMSBs are unidirectional full cross bar switches that connect signals coming into the cluster to its subclusters and signals going out of a cluster to the other clusters of hierarchy. The number of DMSBs in a switch block of a cluster at level l are equal to the number of inputs of a cluster at level $l-1$ and the number of UMSBs in a cluster at level l are equal to the number of outputs of a cluster at level $l-1$. The number of signals entering into and leaving from the cluster can be varied depending upon the netlist requirement. However, they are kept uniform over all the clusters of a particular level. The signal bandwidth of clusters is controlled using Rent's rule [16] which is easily adapted to tree-based architecture. This rule states that

$$\mathrm{IO} = \left(\underbrace{k \cdot n^{l}}_{L \cdot B(p)} + \underbrace{\sum_{x=1}^{z} a_x \cdot b_x \cdot n^{(l-l_x)}}_{H \cdot B(p)} \right)^{p}, \quad (1)$$

where

$$H \cdot B(p) = \begin{Bmatrix} 0 & \text{if } (l - l_x < 0) \\ a_x \cdot b_x \cdot n^{(l-l_x)} & \text{if } (l - l_x \geq 0) \end{Bmatrix}. \quad (2)$$

In (1), l is a tree level, n is the arity size, k is the number of in/out pins of a LUT, a_x is the number of in/out pins of a HB, l_x is the level where HB is located, b_x is the number of HBs at the level where it is located, and IO is the number of in/out pins of a cluster at level l. Since there can be more than one type of HBs, their contribution is accumulated and then added to the $L \cdot B(p)$ part of (1) to calculate p. The value of p is a factor that determines the cluster bandwidth at each level of the tree-based architecture and it is averaged across all the levels to determine the p for the architecture.

2.3. Characteristics of Mesh-Based and Tree-Based Architectures. Both tree-based and mesh-based architectures have particular characteristics that are mainly dependant on the basic interconnect structure and the arrangement of different blocks in the architecture. For example, the major advantage of a tree-based heterogeneous FPGA is its predictable routing, timing behavior, and its independence of the types and position of blocks supported by the architecture. In a tree-based architecture, the number of paths required to reach a destination are limited and hence the number of switches crossed by a signal to reach a destination from a source do not vary greatly. It can be seen from Figure 3 that any CLB can reach any HB by traversing between four to six switches. Unlike tree-based FPGAs, routability of mesh-based FPGA is greatly dependent upon the position of different blocks on the architecture. In mesh-based FPGAs, routability is not predictable and the number of paths available to reach a destination is almost unlimited. Hence the number of switches crossed to reach a destination varies with respect to the position of blocks in the architecture. For example, any CLB in the leftmost column of Figure 1 crosses at least eight switches to reach an HB in the second last column of the architecture. However, this number of switches is reduced to only one if that CLB is placed beside the HB of the second last column of architecture. So, floor-planning plays a very important role in island-style heterogeneous FPGAs and this role becomes more important with the increase in types of HBs that are required to be supported by the architecture.

3. Exploration Environments

In this section, the exploration environments of two FPGA architectures are presented along with different floor-planning techniques that are explored using these exploration environments.

3.1. Exploration Environment of Mesh-Based Architecture. This work uses the mesh-based architecture exploration environment presented earlier in [17] which is further improved by implementing Range Limiter [18] and column-move operation for heterogeneous architectures. In this environment, an FPGA architecture is initially defined using an architecture description file. Architecture description file includes a certain number of parameters that are used for the exploration of the architecture. Some of these parameters include the size of slot-grid, the types of blocks supported by the architecture, the type of routing network (either unidirectional or bidirectional), initial channel width, parameters regarding the optimization of architecture, and the parameters regarding the position of different blocks on the

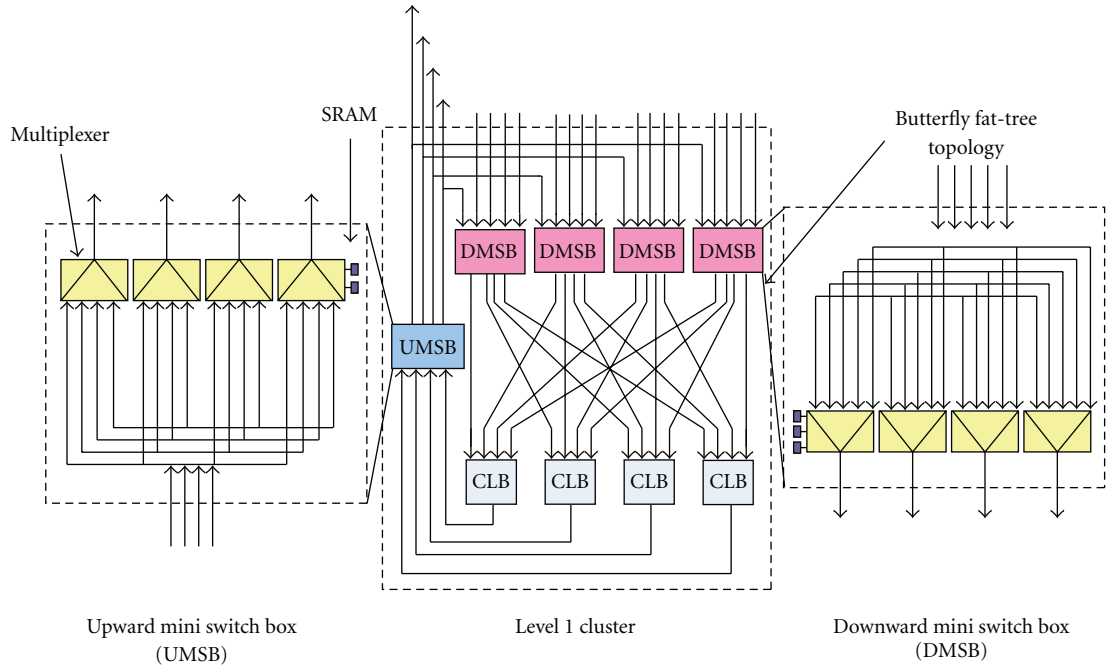


Figure 4: Detailed interconnect of level 1 cluster of tree-based FPGA.

architecture. Blocks of different sizes are defined, and later mapped on a grid of equally sized slots, called a slot-grid. Each block occupies one or more slots. The type of the block and its input and output pins are used to find the size of a block. Each pin of the block is given a name, a class number, a direction, and a slot position on the block to which this pin is connected. Pins with the same class number are considered equivalent; thus a NET targeting a receiver pin of a block can be routed to any of the pins of the block belonging to the same class. Once the architecture of FPGA is defined, the benchmark circuit is placed on the architecture using a placer that performs a number of operations to explore different floor-planning techniques of the architecture. An overview of different operations performed by placer is given below.

3.1.1. PLACER Operations. For mesh-based architecture, a simulated annealing-based [19, 20] PLACER is used to place connected instances near to each other so that placement cost of the architecture is minimized and minimum routing resources are required to connect the instances that communicate with each other. In order to minimize the routing resource and optimize the placement solution, PLACER performs a number of operations that are summarized below.

(i) Moving an instance from one block to another.

(ii) Moving a block from one slot to another.

(iii) Rotating a block around its own axis.

(iv) Moving a complete column of blocks from one slot position to another.

After each operation, the placement cost is recomputed for all the disturbed nets. Depending on the cost value and the annealing temperature, the operation is accepted or rejected. Multiple netlists can be placed together to get a single architecture floor-planning for all the netlists. For multiple netlist placement, each block allows mapping of multiple instances on itself, but multiple instances of the same netlist cannot be mapped on a single block.

PLACER performs its move and rotate operations on a "source" and a "destination". The "source" is randomly selected to be either an instance from input netlist or a block from the architecture. If the "source" is an instance to be moved, any random matching block is selected as its "destination". If the "source" is a block, then a slot position is selected as its "destination". If a "source" block is to be rotated, the same source slot position becomes the "destination". If the "source" block is to be moved, then any random slot is selected as its "destination". The rectangular window starting from this destination slot and having same size and shape as that of source is called destination window whereas the window occupied by the source is called source window. Normally, source window contains one block whereas destination window can contain multiple blocks. An example of source and destination windows is shown in Figures 5(a) and 5(b), respectively. Once the source and destination windows are selected, the move operation is performed if

(i) destination window does not contain any block that exceeds the boundary of destination window. An example violating this condition is shown in Figure 5(c),

(ii) the destination window does not exceed the boundaries of slot-grid (Figure 5(d)),

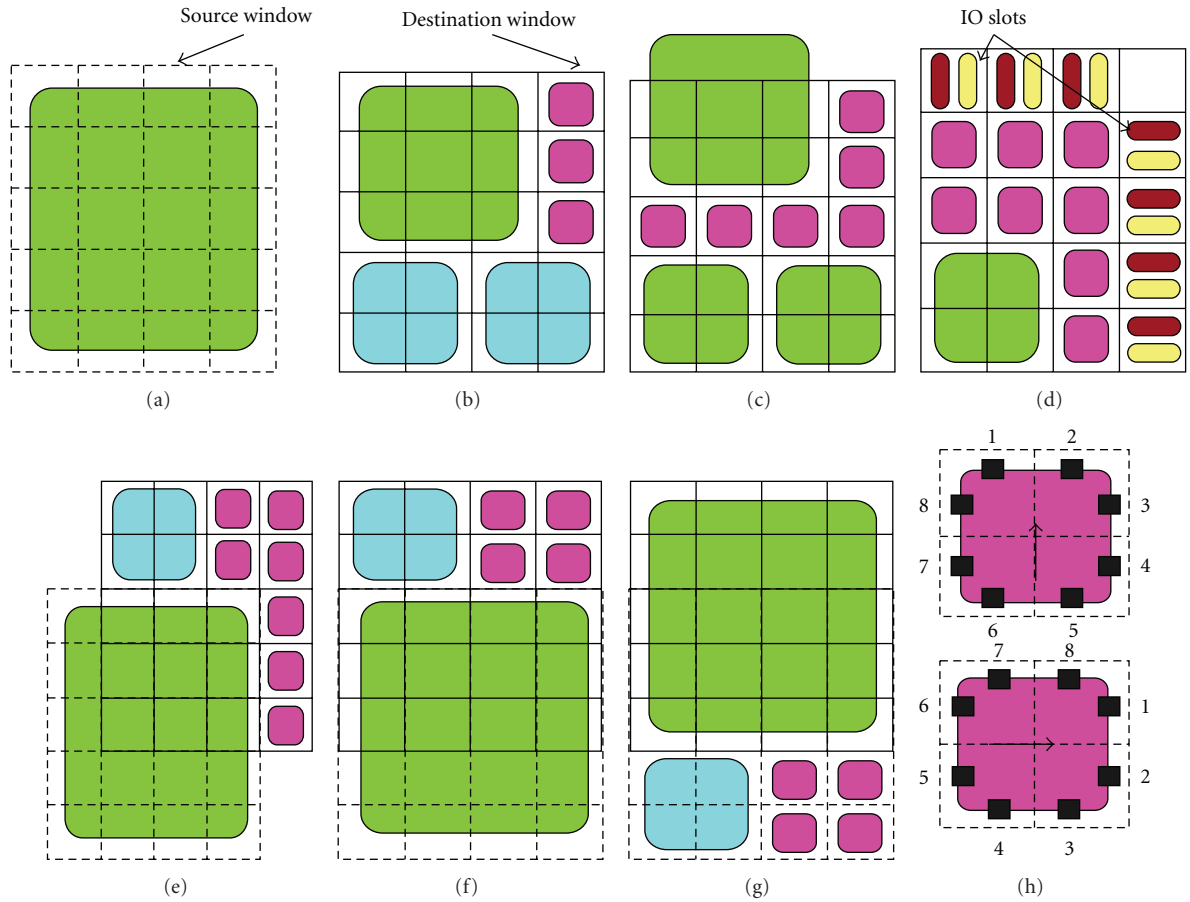


Figure 5: Placer operations.

(iii) destination window does not overlap source window diagonally (Figure 5(e)). However, if, the destination window overlaps source window vertically or horizontally, then horizontal or vertical translation operation is performed. Figure 5(f) shows an example where destination window overlaps source window vertically, and Figure 5(g) shows that the move operation is performed using vertical translation.

However, if the above three conditions are not met, the procedure continues until a valid destination window is found. After the selection of source and destination, any one of the following operations is performed by the placer.

(i) Instance Move: In this case, a move operation is applied on the source instance and the destination block. If the destination block is empty, the source instance is simply moved to the destination block. If the destination block is occupied by an instance, then an instance swap operation is performed.

(ii) Block Jump: If the source window does not overlap with the destination window, then a JUMP operation is performed. All the blocks in the destination window are moved to the source window, and the source block is moved in the destination window. The instances mapped on a block also move along with the block.

(iii) Block Translate: If the source and the destination windows overlap, then a translation operation is performed. Only horizontal and vertical translations are currently performed; diagonal translation is not performed in this work.

(iv) Block Rotate: The rotation of blocks is important when the class number assigned to the input pins of a block are different; bounding box varies depending upon the pin positions and their directions. A block can have an orientation of 0°, 90°, 180°, or 270°. The orientation of a block is used by the bounding box (the minimum rectangle containing the source instance and all its destination instances) evaluation function to correctly calculate the exact position and direction of each of its pins. When an instance of a netlist is moved from one block to another block having different orientations, the orientation of both the old block and the new block are used to compute the difference in the bounding box. Figure 5(h) depicts a 90° clock-wise rotation. Multiples of 90° rotation are allowed for all the blocks having a square shape, whereas at the moment only multiples of

180° rotation are allowed for rectangular (nonsquare) blocks. A 90° rotation for nonsquare blocks involves both rotation and move operations, which is left for future work.

(v) Column Move: The column move operation moves a complete column of blocks from one slot position to another. If the source block is restricted to remain in a column, a column move operation is performed.

3.1.2. Floor-Planning Techniques. By using different PLACER operations, six floor-planning techniques are explored. The details of these floor planning techniques are as follows.

(i) Apart: In this technique, hard blocks are placed in fixed columns, apart from the CLBs. This technique is shown in Figure 6(a) and is termed as Apart (A). Such kind of technique can be beneficial for datapath circuits as described by [21]. It can be seen from the figure that if all HBs of a type are placed and still there is space available in the column, then in order to avoid wastage of resources, CLBs are placed in the remaining place of the column.

(ii) Column-Partial: Figure 6(b) shows the Column-Partial (CP) technique where columns of HBs are evenly distributed among columns of CLBs.

(iii) Column-Full: Figure 6(c) shows Column-Full (CF) technique where columns of HBs are evenly distributed among CLBs. Contrary to the first and second techniques, the whole column contains only one type of blocks. This technique is normally used in commercial architectures.

(iv) Column-Move: In this technique, HBs are placed in columns but unlike the first three techniques, columns are not fixed, rather they are allowed to move using the column-move operation of PLACER. This technique is shown in Figure 6(d) and it is termed as Column-Move (CM).

(v) Block-Move: In this technique, HBs are not restricted in columns; and they are allowed to move through block move operation. This technique is termed as Block-Move (BM) and it is shown in Figure 6(e).

(vi) Block-Move-Rotate: The blocks in this technique are allowed to move and rotate through block move and rotate operations. This floor-planning technique is shown in Figure 6(f) and it is termed as Block-Move-Rotate (BMR).

3.2. Exploration Environment of Tree-Based Architecture. A tree-based architecture is defined using an architecture description file. The architecture description file contains different architectural parameters along with the definition of different blocks used by the architecture. Some of these architecture parameters include the number of levels in the architecture, the types of blocks supported by the architecture, initial signal bandwidths of different clusters situated at different levels of hierarchy, the arity of different clusters of the architecture and so forth. In a tree-based architecture, the definition of a block (CLB or HB) includes the type, area, the level where it is located, and the class numbers for each of its input and output pins. Similar to mesh-based architecture, in this architecture, pins with the same class number are considered equivalent.

Once the architecture is defined, a Fidducia-Mattheyses-(FM) [22] based PARTITIONER partitions the netlist using a top-down recursive partitioning approach. The main objective of PARTITIONER is to reduce communication between different partitions (clusters), and FM algorithm achieves this objective using a hill-climbing, nongreedy, iterative improvement approach. During each iteration, a block with the highest gain is moved from one partition to another and then it is locked and it is not allowed to move during the remaining time of iteration. After the block is moved, the gain of all of its associated blocks is recomputed and this process continues until all the blocks are locked. At the end of an iteration, total cost is compared to that of previous iteration and the algorithm is terminated when it fails to improve during an iteration. After the netlist is partitioned, it is placed and routed on the architecture.

3.2.1. Exploration Techniques. In order to explore the architecture, we have used two exploration techniques.

(i) Symmetric: A generalized example of first technique is shown in Figure 7. This technique is referred to as symmetric (SYM). In this technique, HBs can be placed at any level of hierarchy in order to have best design fit. However, in this technique the symmetry of hierarchy is respected which can eventually result in wastage of HBs and their associated routing resources. For example, in Figure 7, it can be seen that this architecture supports 4 clusters of HBs of a certain type where each cluster contains 4 HBs. This is because of the fact that this is an arity 4 architecture. However, the respect for the symmetry of hierarchy may lead to underutilization of HBs and their associated routing resources in the case where a netlist requires less HBs than supported by the architecture.

(ii) Asymmetric: Contrary to the first technique, where the architecture contains only one structure, the second technique contains two substructures: one substructure contains only CLBs while the other contains only HBs. An example of the second technique is shown in Figure 8. The main motivation behind this technique is the easy management of logic and routing resources. Also the substructure containing only HBs does not have to respect the arity of the substructure containing only CLBs, hence leading to more optimized logic and routing resources. This technique is referred to as asymmetric (ASYM).

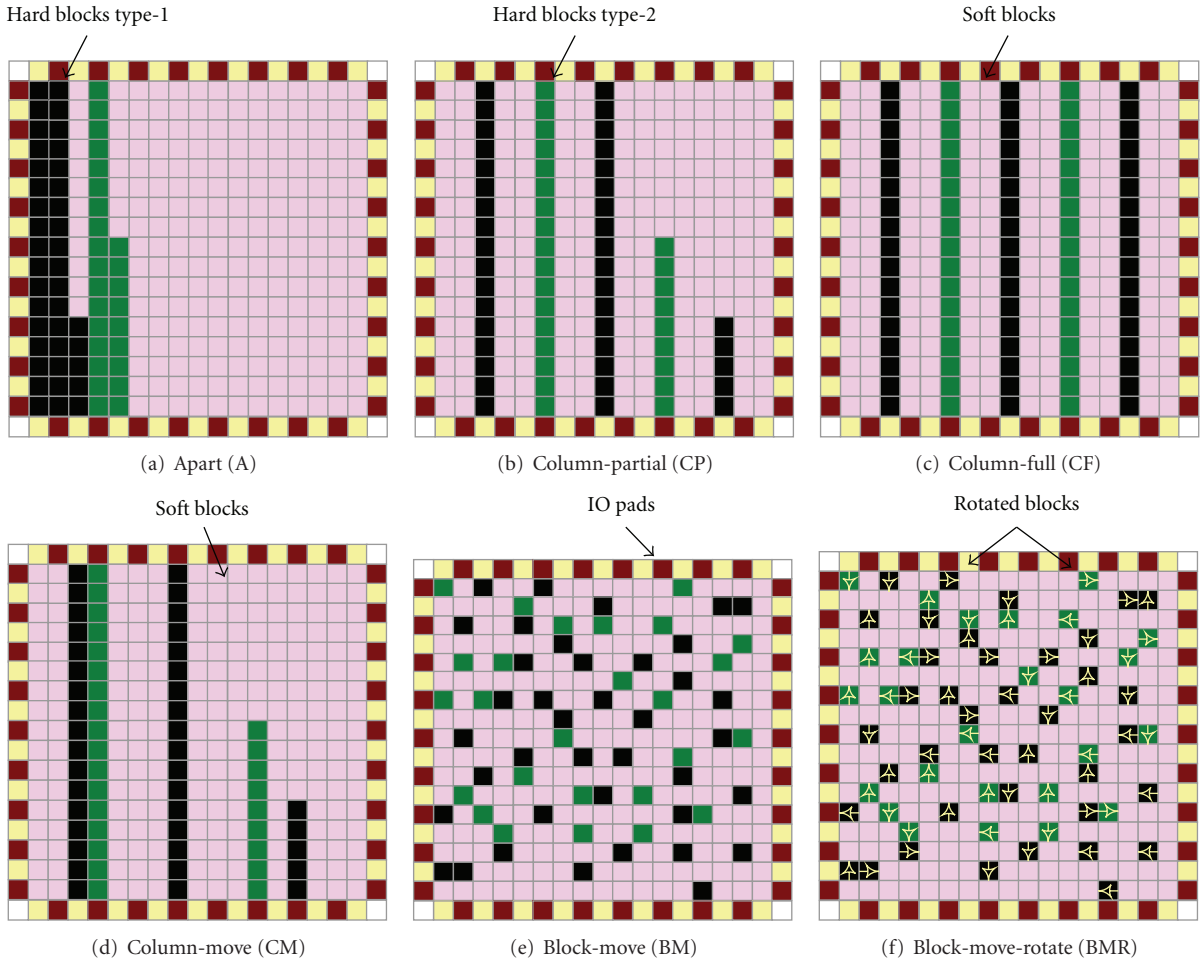


Figure 6: Floor-planning techniques.

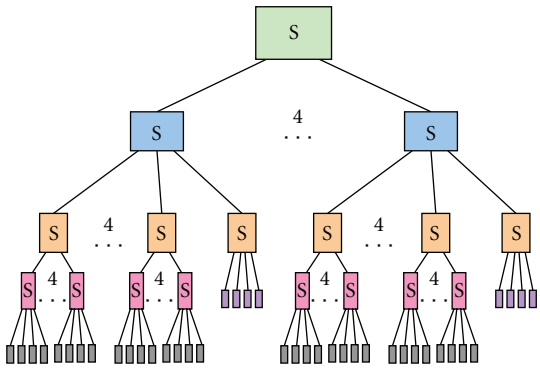


Figure 7: Symmetric tree-based heterogeneous FPGA architecture.

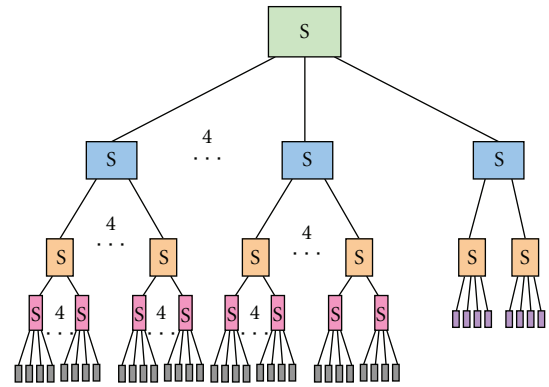


Figure 8: Asymmetric tree-based heterogeneous FPGA architecture. (a) substructure containing only CLBs. (b) substructure containing only HBs.

4. Experimental Flow

Evaluation of different floor-planning/exploration techniques of the two architectures is performed using a specifically designed experimental flow. Details of this experimental flow are explained in this section.

4.1. Benchmark Selection. Generally, in academia and industry, the quality of an FPGA architecture is measured by mapping a certain set of benchmarks on it. Thus the selection of benchmarks plays a very important role in the exploration of heterogeneous FPGAs. This work puts special emphasis on

Table 1: DSP benchmarks set I.

Circuit name	Inputs	Outputs	CLBs (LUT4)	Mult (8×8)	Slansky ($16 + 16$)	Sff (8)	Sub ($8 - 8$)	Smux ($32 : 16$)
ADAC	18	16	47	—	—	2	—	1
DCU	35	16	34	1	1	4	2	2
FIR	9	16	32	4	3	4	—	—
FFT	48	64	94	4	3	—	6	—

Table 2: Open core benchmarks set II.

Circuit name	No of inputs	No of outputs	No of LUTs	No of multipliers (16×16)	No of adders ($20 + 20$)
cf_fir_3_8_8	42	18	159	4	3
cf_fir_7_16_16	146	35	638	8	14
cfft 16×8	20	40	1511	—	26
cordic_p2r	18	32	803	—	43
cordi_r2p	34	40	1328	—	52
fm	9	12	1308	1	19
fm_receiver	10	12	910	1	20
lms	18	16	940	10	11
reed_solomon	138	128	537	16	16

Table 3: Open core benchmarks set III.

Circuit name	No of inputs	No of outputs	No of LUTs	No of multipliers (18×18)
cf_fir_3_8_8	42	22	214	4
diffeq_f_system C	66	99	1532	4
diffeq_paj_convert	12	101	738	5
fir_scu	10	27	1366	17
iir1	33	30	632	5
iir	28	15	392	5
rs_decoder_1	13	20	1553	13
rs_decoder_2	21	20	2960	9

the selection of benchmark circuits, as different circuits can give different results for different architecture floor-planning techniques. This work categorizes the benchmark circuits by the trend of communication between different blocks of the benchmark. So, three sets of benchmarks are assembled having distinct trend of interblock communication. These benchmarks are shown in Tables 1, 2, and 3 and they are obtained from [23–25], respectively. The communication between different blocks of a benchmark can be mainly divided into the following four categories.

(i) CLB-CLB: CLBs communicate with CLBs.

(ii) CLB-HB: CLBs communicate with HBs and vice versa.

(iii) HB-HB: HBs communicate with other HBs.

(iv) IO-CLB/HB: I/O blocks communicate with CLBs and HBs.

In SET I benchmarks, the major percentage of total communication is between HBs (i.e., HB-HB) and only a small part of total communication is covered by the communication CLB-CLB or CLB-HB. Similarly, in SET II the major percentage of total communication is HB-CLB and in SET III, major percentage of total communication is covered by CLB-CLB. Normally the percentage of IO-CLB/HB is a very small part of the total communication for all the three sets of benchmarks.

4.2. Software Flow. The software flow used to place and route different benchmarks (netlists) on the two heterogeneous FPGAs is shown in Figure 9. The input to the software flow is a VST file (structured vhdl). This file is converted into BLIF format [26] using a modified version of VST2BLIF tool. The BLIF file is then passed through PARSER-1 which removes HBs from the file and adds temporary inputs and outputs to the file to preserve the dependance between HBs and the rest of the netlist. The output of PARSER-1 is then passed through SIS [27] that synthesizes the BLIF file into LUT format which is later passed through T-VPACK [28] which packs and converts it into. NET format. Finally, the netlist is passed through PARSER-2 that adds previously removed HBs and also removes temporary inputs and outputs. The final netlist in. NET format contains CLBs, HBs, and I/O instances that are connected to each other via NETS. After obtaining the netlist in. NET format, floor-plannings of Section 3 are explored separately for both tree-based and mesh-based architectures using their respective flow.

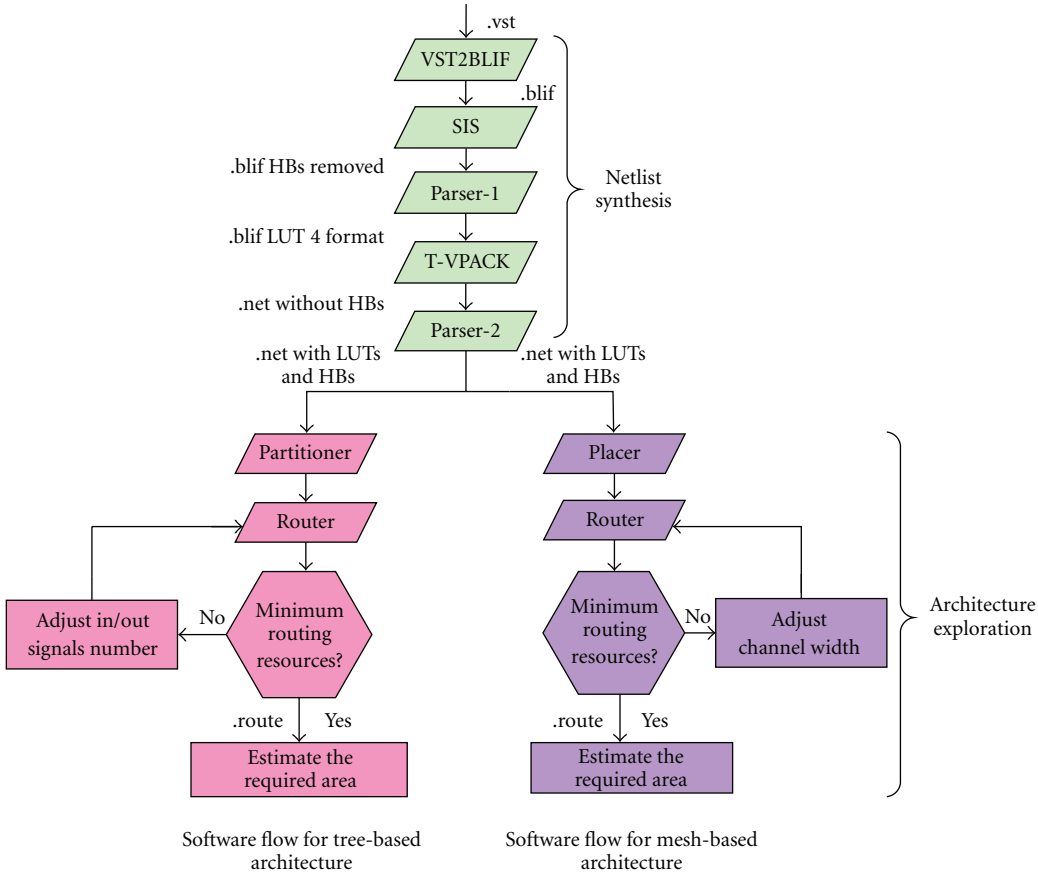


Figure 9: Software flow.

4.2.1. Software Flow for Tree-Based Architecture. For tree-based architecture, the netlist obtained in. NET format is first partitioned using a software module called PARTITIONER. This module partitions CLBs, HBs, and I/Os into different clusters in such a way that the intercluster communication is minimized. By minimizing intercluster communication we obtain a depopulated global interconnect network which leads to smaller intercluster signal bandwidths and eventually gives good tradeoff both in terms of delay and area. PARTITIONER is based on hMetis [29] platform. hMetis combines FM algorithm with its multiphase refinement approach to optimize the partitioning of the netlist. These phases include coarsening, initial partitioning, uncoarsening, and refinement phase [30]. Once partitioning is done, a placement file is generated that contains positions of different blocks on the architecture. This placement file along with netlist file is then passed to another software module called ROUTER which is responsible for the routing of netlist. In order to route all the NETS of netlist, routing resources of the interconnect structure are first assigned to the respective blocks of the netlist that are placed on the architecture. These routing resources are modeled as directed graph abstraction $G\ (V, E)$ where the set of vertices V represents the in/out pins of different blocks and the routing wires in the interconnect structure and an edge E between two vertices represents a potential connection between the two vertices. ROUTER is based on PathFinder [31] routing algorithm that uses an iterative, negotiation-based approach to successfully route all nets in a netlist. In order to optimize the FPGA architecture, a binary search algorithm is used. This algorithm determines the minimum number of signals required to route a netlist on FPGA. Once the optimization is over, area of the architecture is estimated using an area model which is based on symbolic standard cell library SXLIB [32].

4.2.2. Software Flow for Mesh-Based Architecture. For mesh-based architecture, the netlist file is passed to a software module called PLACER that uses simulated annealing algorithm [18, 19] to place CLBs, HBs, and I/Os on their respective blocks in FPGA. The main objective of the PLACER is to place connected instances close to each other so that minimum routing resources are required to route their connection. For this purpose, PLACER optimizes the placement cost of the architecture which is equal to the sum of half-perimeters of the bounding boxes of all NETS. The bounding box (BBX) of a NET is a minimum rectangle that contains the driver instance and all receiving instances of a NET. PLACER moves an instance randomly from one block position to another; the BBX cost is updated. Depending on cost value and annealing temperature, the operation is accepted or rejected. After placement, a software module named ROUTER routes the netlist on the architecture.

Table 4: Area of different blocks of three sets.

Block name	Inputs	Outputs	Block size (λ^2)
clb	4	1	58500
mult (8×8)	16	16	1075250
slansky_16	32	16	306750
sff_8	8	8	36000
sub_8	17	8	154500
smux_16	33	16	36000
mult (16×16)	32	32	1974000
adder ($20 + 20$)	41	21	207000
mult (18×18)	36	36	2498300
sram	—	—	1500
buffer	1	1	1000
flip-flop	1	1	4500
mux 2 : 1	2	1	1750

Similar to the ROUTER of tree-based FPGA, mesh-based FPGA uses a pathfinder algorithm [31] to route the netlist using FPGA routing resources. In order to optimize the FPGA resources, a binary search algorithm similar to the one used for tree-based FPGA is used to determine the smallest channel width required to route a netlist.

4.2.3. Architecture Evaluation. Once the netlist routing is completed, the area, performance, and power estimation of architecture is performed (separately for mesh- and tree-based architectures). The area of FPGA architecture is estimated by combining areas of CLBs, HBs, multiplexors of interconnect, and all associated programming bits. Our area model is based on standard cell library SXLIB [32] and the area of different cells that are used for the calculation of area is shown in Table 4. Since we do not have accurate wire length estimation, performance evaluation of the architecture is performed by counting the number of switches that are crossed by critical path and static power estimation is performed by combining the number of configuration memories and buffers.

4.3. Experimental Methodology. In order to have a detailed analysis of different techniques, we have employed two different methodologies for the experimentation: individual experimentation and generalized experimentation.

4.3.1. Individual Experimentation. In the first methodology, experiments are performed individually for each netlist (both for mesh- and tree-based architectures). The architecture definition, floor-planning, placement, routing and optimization is performed individually for each netlist. Although such an approach is not applicable to real FPGAs, as their architecture, floor-planning, and routing resources are already defined, this methodology is useful in order to have detailed analysis of a particular floor-planning technique and usually it is employed to evaluate different parameters of the architecture under consideration. If a generalized architecture is defined for a group of netlists, the netlists with the highest logic and routing requirements decide logic and routing resources of the architecture and the behavior of remaining netlists of the group is overshadowed by larger netlists of the group. So, to get more profound results, the architecture and floor-planning is optimized individually for each netlist; later average of all netlists gives more thorough results.

4.3.2. Generalized Experimentation. However, in order to further validate the results, we have also performed experimentation based on the generalized architecture. In this methodology, for mesh-based architecture, a generalized architecture is defined for each SET of netlists and the floor-planning is then optimized for this architecture. Generalized floor-planning is achieved by allowing the mapping of multiple nelists on the same architecture where each block of the architecture allows mapping of multiple instances on it, but multiple instances of the same netlist are not allowed. Similarly, for tree-based architecture, multiple netlists are partitioned using generalized architecture description. Once generalized floor-planning optimization/partitioning is over, individual netlists are placed and routed separately on both architectures using the above described flow except that optimization of the architecture is not performed for individual netlists.

5. Experimental Results

5.1. Experimental Results Using Individual Methodology. Since placement cost and channel width of mesh-based architecture is directly related to its area, we first present the effect of different floor-planning techniques of mesh-based architecture on these two values. Placement cost and channel width results obtained for three sets of benchmarks are shown in Figures 10 and 11, respectively. In these figures, the results for benchmarks 1 to 4, 5 to 13, and 14 to 21 correspond to SET I, SET II, and SET III, respectively. The avg1, avg2, and avg3 in the Figures 10 and 11 correspond to the geometric average of these results for SET I, SET II, and SET III, respectively. The avg corresponds to the average of all netlists.

As explained earlier, placement cost is the sum of the bounding box cost of all the nets of the netlist being implemented on the architecture and this cost gives us a measure of the quality of the placement solution that is provided by a certain floor-planning technique. In Figure 10, for each of the 21 benchmark circuits, the placement cost given by five floor-planning techniques (i.e., Apart (A), Column-Partial (CP), Column-Full (CF), Column-Move (CM), and Block-Move (BM)) is normalized against the placement cost of Block-Move-Rotate (BMR). As it can be seen from the figure that in general, Apart (A) gives the worst and BMR gives the best placement cost results whereas the results of remaining techniques are in between these two techniques. In Apart, the average placement cost is higher than the other floor-planning techniques because in this technique columns of hard blocks are fixed and they are separated from CLBs. Although this kind of floor-planning technique

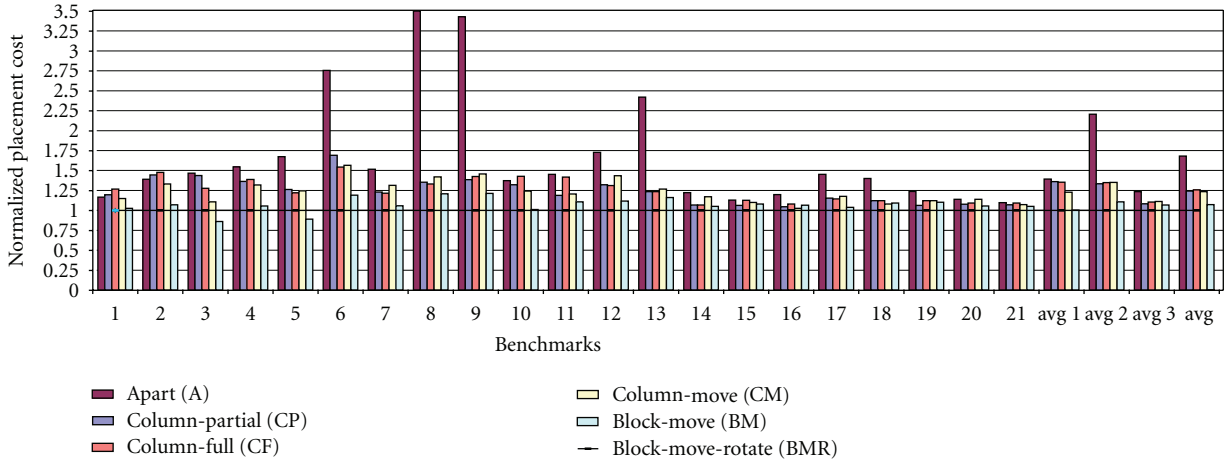


Figure 10: Placement cost comparison between different techniques of mesh-based architecture.

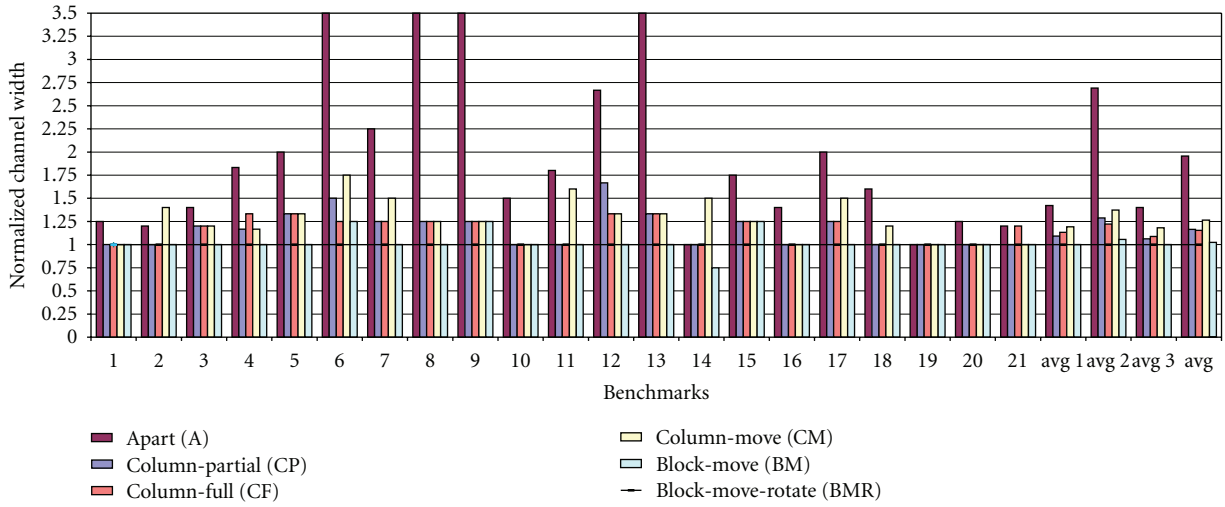


Figure 11: Channel width comparison between different techniques of mesh-based architecture.

can give good results for datapath circuits, it gives poor placement solution for control path circuits as the columns of HBs are fixed and they are not mixed with CLBs. This situation further aggravates if there are more than one type of HBs that are required to be supported by the architecture. Although the columns of HBs are fixed in CF and CP, they give better placement cost results when compared to Apart as in those techniques, the columns of hard blocks are not placed apart rather they are interspersed evenly among CLBs, hence leading to smaller placement costs. BMR gives the best placement cost results because it is the most flexible technique among the six floor-planning techniques. Although the only difference between BM and BMR is that of hard-block rotation, it gives slightly more flexibility to BMR which might eventually lead to smaller BBX and eventually lower placement costs of the architecture.

Figure 11 gives the channel width results of the six floor-planning techniques of mesh-based architecture. In this figure, for 21 benchmarks, channel widths of 5 floor-planning techniques are normalized against the channel width of BMR. Similar to the results in Figure 10, BMR gives the best results and Apart gives the worst results. The two figures (i.e., Figures 10 and 11) look similar to each other as (i) both figures give normalized results and (ii) placement cost and channel width are closely related to each other. Generally an architecture with higher placement cost indicates a poor placement solution as in this solution instances connected to each other are placed far from each other. A poor placement solution normally leads to higher channel width of the architecture as the instances placed far from each other require more routing resources than the ones placed close to each other. Analysis of the results in Figures 10 and 11 shows that, on average, CF gives 35%, 35%, and 11% more placement cost than BMR, for SET I, SET II, and SET III benchmark circuits, respectively. Figure 11 shows that, on average, CF requires 13%, 22%, and 9% more channel width than BMR for SET I, SET II, and SET III, respectively. The increase in channel width increases the overall area of the architecture, as shown in Figure 12. In this figure, the area results of A, CF, BM floor-planning techniques of

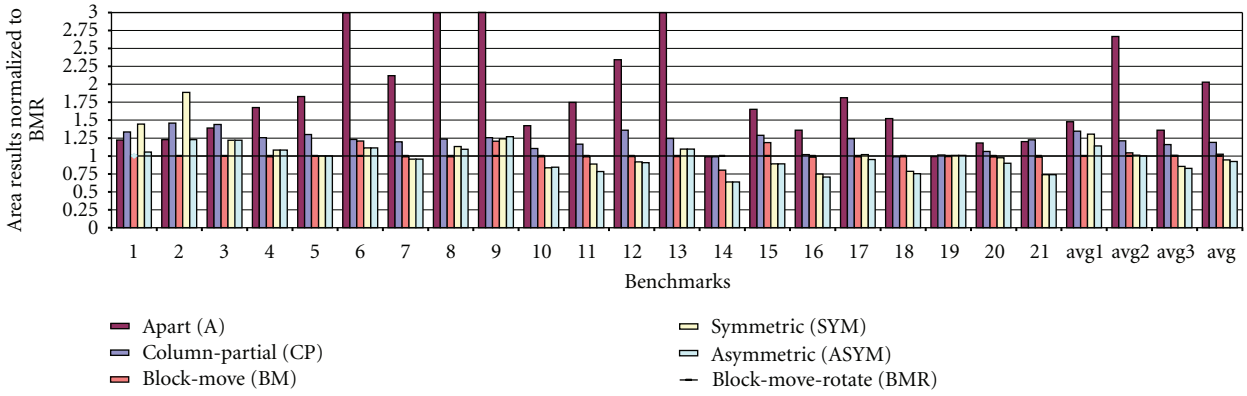


Figure 12: Area comparison between different techniques of mesh- and tree-based architectures.

mesh-based FPGA and SYM and ASYM techniques of tree-based FPGA are normalized against the area results of BMR floor-planning technique of mesh-based FPGA. For the sake of clarity, the results for CP and CM floor-planning techniques are not presented. On average, CF requires 36%, 23%, and 10% more area than BMR for SET I, SET II, and SET III, respectively. For SET I benchmark circuits, SYM requires 35% more area than BMR, and ASYM requires 10% more area than BMR. However, for SET II benchmark circuits, on average, BMR is almost equal to SYM and ASYM. For SET III benchmark circuits BMR is worse than SYM and ASYM by 14%, and 18%, respectively.

Analysis of the results shown in Figure 12 reveals that area of CF compared to BMR varies depending upon the set of benchmarks that are used. For SET I benchmark circuits, where the types of blocks for each benchmark are two or more than two and communication is dominated by HB-HB type of communication, CF produces worse results than the other two sets of benchmarks. This is because of the fact that columns of different HBs are separated by columns of CLBs and HBs need extra routing resources to communicate with other HBs. However, in BMR there is no such limitation as HBs can always be placed close to each other. For the other two set the gap between CF and BMR is relatively less. The reduced HB-HB communication in SET II, and SET III benchmark circuits is the major cause of reduction in the gap between CF and BMR. However, 23% and 10% area difference for SET II, and SET III is due to the placement algorithm. In CF, the simulated annealing placement algorithm is restricted to place hard-block instances of a netlist at predefined positions. This restriction reduces the quality of placement solution which leads to the demand for more routing resources to route the netlist; thus more area is required. The results show that BMR technique produces the least placement cost, the smallest channel width, and hence the smallest area for mesh-based heterogeneous FPGA. However, BMR floor-planning technique is dependent upon target netlists to be mapped upon FPGA. Although such an approach is not suitable for generalized FPGAs, it can be beneficial for domain-specific FPGAs. Moreover, the hardware layout of BMR requires more efforts than CF.

For tree-based FPGA, ASYM produces the best results in terms of area and it is better than the best technique of mesh-based FPGA (i.e., BMR) by an average of 5% for a total of 21 benchmarks. The major advantage of a heterogeneous tree-based FPGA is that the maximum number of switches required to route a connection between CLB-HB or HB-HB remain relatively constant. However, in case of SET I benchmarks, extensive HB-HB communication gives rise to total switch requirement of tree-based architecture; hence giving poor area results as compared to mesh-based architecture. The architecture floor-planning of tree-based FPGA does not affect the switch requirement of the architecture. However, the floor-planning of mesh-based FPGA causes a drastic impact on the switching requirement of the architecture.

In order to evaluate the performance of different techniques of two architectures, we have calculated the number of switches crossed by critical path. Since we are exploring a number of techniques for both mesh- and tree-based architectures, it will be very difficult to perform layout for each technique and determine the exact critical path delay. So, we use a simple model that gives an overview of the impact of active routing resources (switches) on the overall performance of the architecture. Similar to area results, critical path results are normalized against BMR floor-planning of mesh-based FPGA. These results are shown in Figure 13. To avoid congestion, results for only 6 out of 8 techniques are shown. It can be seen from Figure 13 that due to its higher flexibility, BMR gives higher performance results than other floor-planning techniques of mesh-based FPGA. On average, CF critical path crosses 5%, 7%, and 10% more switches than BMR technique for SET I, SET II, and SET III benchmarks, respectively. Although Apart (A) gives worst results in terms of placement cost, channel width, and area, it is quite interesting to note that critical path results of Apart (A) are comparatively better than CF. This is because of the fact that in Apart, columns of HBs are placed close to each other and apart from the CLBs. Since in SET I benchmarks the majority of the communication involves HB-HB communication, there is a strong probability that critical path involves HBs which ultimately leads to 50% of benchmarks of SET I crossing a smaller number of switches than CF. For SET II benchmarks this percentage drops to 44% as there is more

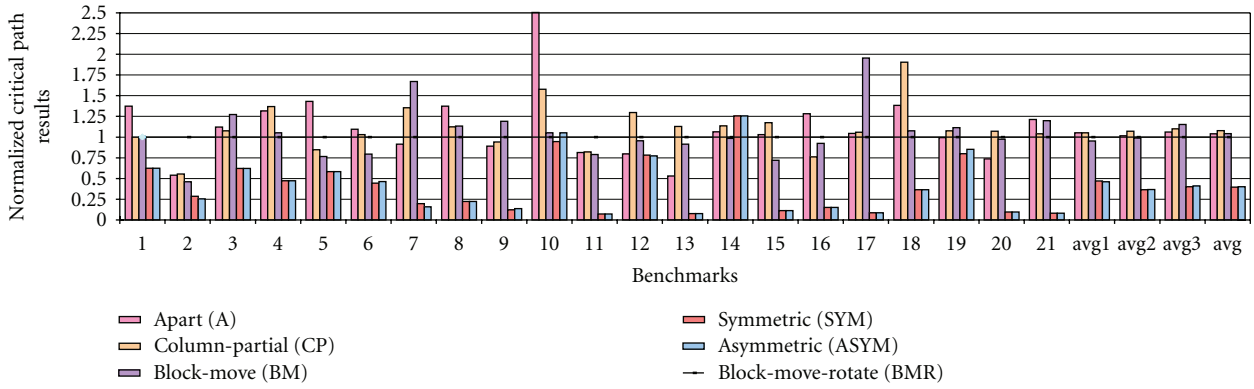


Figure 13: Critical path comparison between different techniques of mesh- and tree-based architectures.

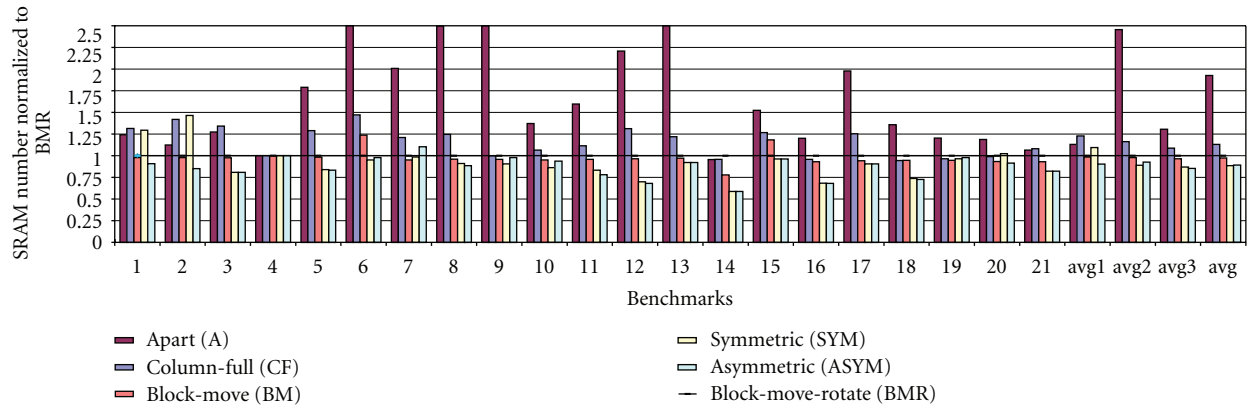


Figure 14: Configuration memory comparison between different techniques of mesh- and tree-based architectures.

communication between CLBs and HBs. However, in case of SET III benchmarks, 75% of benchmarks cross a smaller number of switches for Apart than CF as the communication pattern is dominated by CLB-CLB and in case of Apart there are no columns of HBs interspersed in between CLBs, hence leading to smaller number of switches that are crossed by critical path. Although Apart gives better results than CF, BMR manages to produce the best overall results among floor-planning techniques of mesh-based architecture due to its higher flexibility.

However, compared to the tree-based FPGA, both SYM and ASYM techniques of tree-based FPGA produce far better performance than BMR technique due to the inherent characteristic of tree-based architecture (see Section 2.3). Compared to BMR technique of mesh-based architecture, on average, SYM and ASYM techniques of tree-based architecture cross 53%, 54% less switches for SET I, 64%, 63% less switches for SET II, and 60%, 59% less switches for SET III benchmarks, respectively. It can also be observed from these results that, on average, ASYM technique crosses 1% more switches than SYM technique. In ASYM technique, HBs have a separate substructure, and if critical path involves HBs and CLBs, then it can lead to an increase in the number of switches crossed by critical path (Figures 7 and 8). However, if critical path involves no HBs or only HBs and I/Os, it can lead to a smaller number of switches than SYM technique (Figure 13 results for benchmark 2 and 7).

Power optimization of FPGAs has become very important with the advancement in process technology. Although in this work a detailed power analysis of mesh-based and tree-based FPGA architectures is not performed, it gives a brief overview of the static power consumption of the two architectures; which has become increasingly important for smaller process technologies [33]. Static power of the FPGAs is directly related to the configuration memory and the number of buffers in an FPGA architecture [34]. Therefore, a comparison of configuration memory and number of buffers for different techniques of the two architectures is shown in Figures 14 and 15, respectively.

Figure 14 shows the number of SRAMs for different techniques normalized against the BMR technique of mesh-based FPGA. Comparison of BMR with CF shows that, on average, CF consumes 23%, 16%, and 9% more SRAMs than BMR for SET I, SET II, and SET III, respectively. Comparison of BMR with tree-based architecture techniques shows that, on average, SYM consumes 9% more and ASYM consumes 10% less SRAMs for SET I. However, for SET II and SET III SYM, and ASYM consume 11%, 7%, and 13%, 15% less SRAMs than BMR, respectively. Similarly Figure 15 shows that, compared to BMR, CF consumes 9%, 22%, and 18%

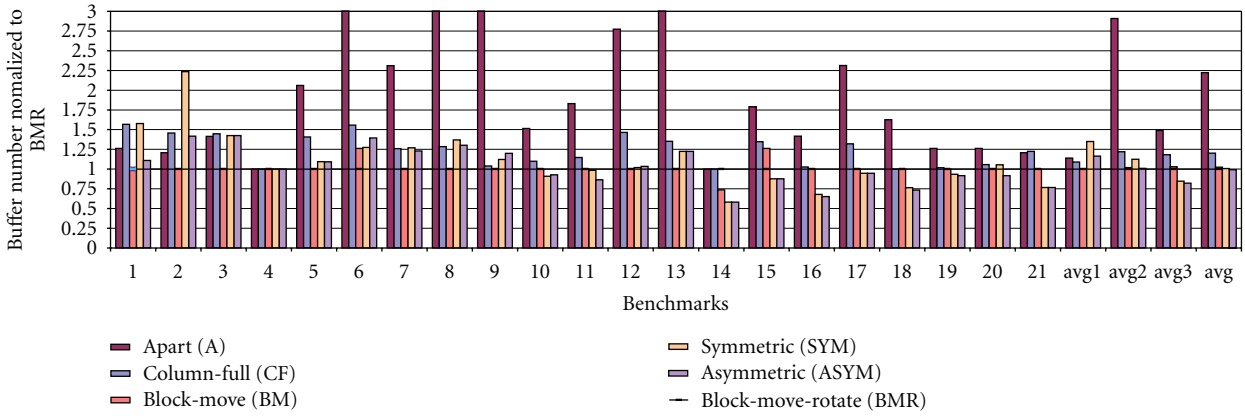


Figure 15: Buffer comparison between different techniques of mesh- and tree-based architectures.

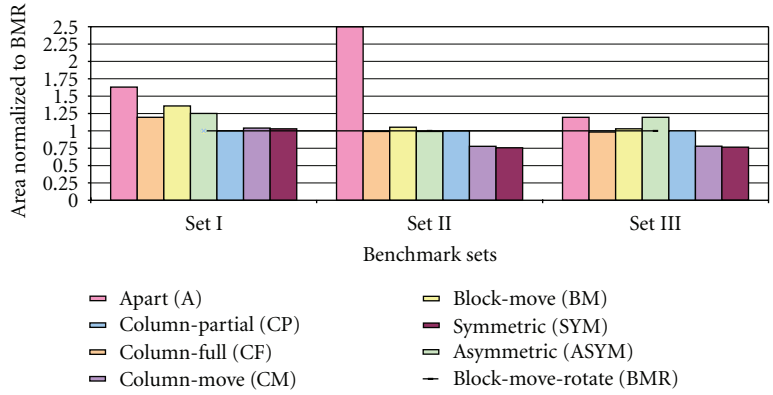


Figure 16: Generalized area results of mesh- and tree-based architectures.

more buffers for SET I, SET II, and SET III, respectively. Comparison of SYM and ASYM with BMR shows that both consume 6% more buffers for SET I, 3% less buffers for SET II and 15%, 18% less buffers for SET III. Although the comparison presented in Figures 14 and 15 does not give detailed power estimation of the two architectures, it gives an empirical estimate of the static power of the two architectures and as stated by [35], it closely correlates to the average area results of the two architectures presented in Figure 12 (avg1, avg2, and avg3 of Figure 12).

5.2. Experimental Results Using Generalized Methodology. Figure 16 shows the area results obtained using generalized experimental methodology (Section 4.3.2). In this methodology, a generalized architecture is defined for each SET of netlists that can place and route all netlists of that particular SET. It can be seen from the figure that this methodology further enhances the results obtained by the first experimental methodology. In this methodology too, compared to other floor-planning techniques of mesh-based FPGA, BMR produces equal or better results. However, the gain of BMR compared to CF is reduced from 23%, 10% to 5%, 3% for SET II and SET III benchmarks, respectively, while the gain for SET I benchmarks remains unchanged. This drop in gain is mainly due to the combined floor-planning optimization of all the netlists of a SET where the routing requirements of smaller netlists are overshadowed by larger netlists. As far as the comparison of BMR with SYM and ASYM techniques is concerned, the results of tree-based topologies are further improved. For SET I benchmarks, SYM and ASYM techniques are only 4% and 3% worse than BMR and for SET II, their gain is increased from 0 to 22% and 24% and for SET III their gain is increased from 14% and 18% to 22% and 24%, respectively. Figures 17, 18, and 19 show the generalized critical path, configuration memory and buffers results that are obtained using generalized experimental methodology. The results shown in these figures further reinforce the observations made using individual experimental methodology. Although in our case the results of individual and generalized experimental methodologies comply with each other, there can be cases where the two methodologies contradict each other. So, in order to have a profound insight about different exploration techniques, it is good to perform both kinds of experimentation.

6. Conclusion

This paper has explored two heterogeneous FPGA architectures. Different mesh-based floor-planning techniques are compared. The floor-plannings of mesh-based FPGA

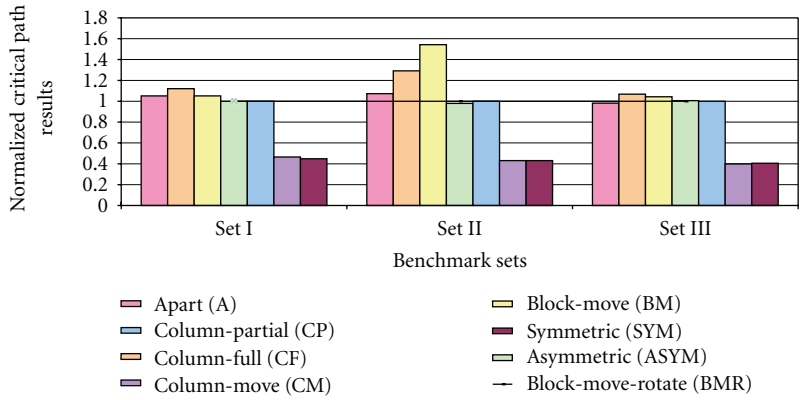


Figure 17: Generalized critical path results of mesh- and tree-based architectures.

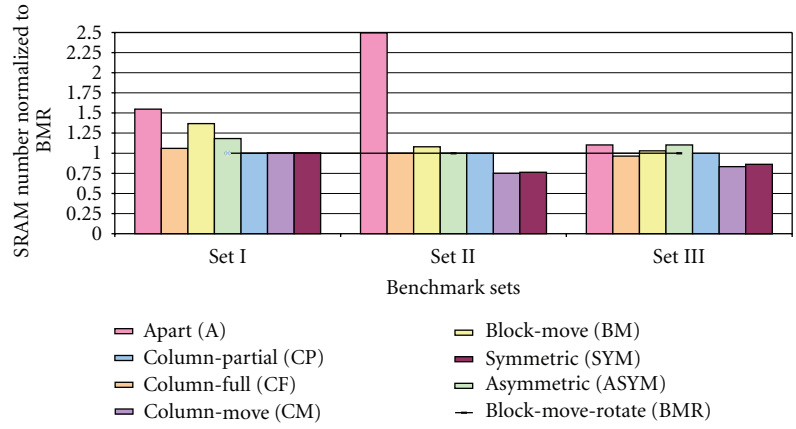


Figure 18: Generalized configuration memory results of mesh- and tree-based architectures.

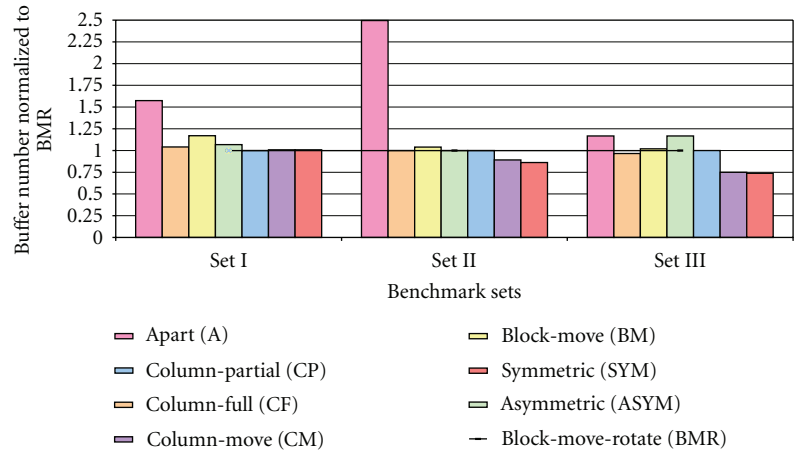


Figure 19: Generalized buffer comparison results of mesh- and tree-based architectures.

influence the routing network requirement of the architecture. Individual experimentation results show that CF floor-planning is on average 36%, 23%, and 10% more area consuming than the BMR floor-planning for three sets of netlists. The tree-based architecture is independent of its floor-planning; however, its layout is relatively less scalable than the mesh architecture. The CF floor-planning is on average 18%, 21%, and 40% larger than the ASYM technique of tree-based FPGA architecture for the same sets of netlists. Performance evaluation results show that BMR technique of mesh-based architecture crosses 4.7%, 6.5%, and 9% less switches and ASYM technique of tree-based architecture crosses 56%, 66%, and 63% less switches than CF technique of mesh-based FPGA. Static power estimation results show that, for three sets, BMR technique consumes 18%, 13% and 8% less SRAMs and 8%, 18%, and 15% less buffers than CF floor-planning of mesh-based FPGA. Similarly, ASYM consumes 26%, 20%, and 22% less SRAMs and 3%, 20%, and 30% less buffers than CF floor-planning of mesh-based FPGA. Among all the techniques compared, ASYM technique of tree-based FPGA gives the best area, performance and power estimation results for all three benchmark sets. However, hardware layout efforts are required to maintain these area benefits on tree-based FPGAs. A mesh containing smaller trees architecture can be designed to resolve scalability issues of tree architecture.

References

[1] S. Wilton, *Architectures and algorithms for field-programmable gate arrays with embedded memory*, Ph.D. dissertation, Citeseer, 1997.

[2] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, "Embedded floating-point units in FPGAs," in *Proceedings of the 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '06)*, pp. 12–20, February 2006.

[3] C. H. Ho, P. H. W. Leong, W. Luk, S. J. E. Wilton, and S. Lopez-Buedo, "Virtual embedded blocks: a methodology for evaluating embedded elements in FPGAs," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 35–44, April 2006.

[4] G. Govindu, S. Choi, V. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar, "A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs," in *Proceedings of the18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, pp. 2035–2042, April 2004.

[5] K. D. Underwood and K. S. Hemmert, "Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04)*, pp. 219–228, April 2004.

[6] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proceedings of the 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '06)*, pp. 21–30, ACM, Monterey, Calif, USA, February 2006.

[7] Xilinx, "Xilinx," http://www.xilinx.com, 2010.

[8] Altera, "Altera," http://www.altera.com, 2010.

[9] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiDReconfigurable pipelined datapath," *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, pp. 126–135, 1996.

[10] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Matt, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[11] A. Abnous and J. Rabaey, "Ultra-low-power domain-specific multimedia processors," in *Proceedings of the 9th IEEE Workshop on VLSI Signal Processing*, pp. 461–470, November 1996.

[12] Z. Marrakchi, H. Mrabet, E. Amouri, and H. Mehrez, "Efficient tree topology for FPGA interconnect network," in *Proceedings of the 18th ACM Great Lakes Symposium on VLSI (GLSVLSI '08)*, pp. 321–326, March 2008.

[13] U. Farooq, H. Parvez, Z. Marrakchi, and H. Mehrez, "A new tree-based coarse-grained FPGA architecture," in *Proceedings of the 5th International Conference on Ph.D. Research in Microelectronics and Electronics (PRIME '09)*, pp. 48–51, July 2009.

[14] J. Luu, I. Kuon, P. Jamieson et al., "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 133–142, ACM, February 2009.

[15] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in FPGA interconnect," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 41–48, December 2004.

[16] B. Landman and R. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Transactions on Computers*, vol. 20, no. 12, pp. 1469–1479, 1971.

[17] H. Parvez, Z. Marrakchi, U. Farooq, and H. Mehrez, "A new coarse-grained FPGA architecture exploration environment," in *Proceedings of the International Conference on Field-Programmable Technology (ICFPT '08)*, pp. 285–288, December 2008.

[18] V. Betz and J. Rose, "VPR: a new packing placement and routing tool for FPGA research," in *Proceedings of the 2nd International Workshop on Field-Programmable Gate Arrays (FPGA '97)*, pp. 213–222, IEEE Press, 1997.

[19] C. C. Skiścim and B. L. Golden, "Optimization by simulated annealing: a preliminary computational study for the tsp," in *Proceedings of the 15th Winter Simulation Conference (WSC '83)*, pp. 523–535, IEEE Press, Piscataway, NJ, USA, 1983.

[20] C. Sechen and A. Sangiovanni-Vincentelli, "The timberwolf placement and routing package," *IEEE Journal of Solid-State Circuits*, vol. 20, no. 2, pp. 510–522, 1985.

[21] D. Cherepacha and D. Lewis, "DP-FPGA: an FPGA architecture optimized for datapaths," *VLSI Design*, vol. 4, no. 4, pp. 329–343, 1996.

[22] C. M. Fiduccia and R. M. Mattheyeses, "A linear-time heuristic for improving network partitions," in *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pp. 175–181, 1982.

[23] http://www-asim.lip6.fr.

[24] http://www.opencores.org/.

[25] http://www.eecg.utoronto.ca/vpr/.

[26] "Berkeley logic Synthesis and Verification Group,University of California, Berkeley. Berkeley Logic Interchange Format (blif)," http://vlsi.colorado.edu/vis/blif.ps.

[27] E. M. Sentovich, K. J. Singh, and L. Lavagno, "Sis: a system for sequential circuit analysis," Tech. Rep. UCB/ERL M92/41, University of California, Berkeley, Calif, USA, 1992.

[28] A. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," in *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pp. 39–46, Monterey, Calif, USA, February 1999.

[29] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proceedings of the 36th Annual Design Automation Conference (DAC '99)*, pp. 343–348, June 1999.

[30] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: application in VLSI domain," in *Proceedings of the 34th Design Automation Conference*, pp. 526–529, ACM, June 1997.

[31] L. McMurchie and C. Ebeling, "PathFinder: a negotiation-based performance-driven router for FPGAs," in *Proceedings of the ACM 3rd International Symposium on Field-Programmable Gate Arrays*, pp. 111–117, February 1995.

[32] A. Greiner and F. Pecheux, "Alliance: a complete set of cad tools for teaching vlsi design," in *Proceedings of the 3rd Eurochip Workshop on VLSI Design Training*, 1992.

[33] Altera, "40-nm FPGA Power Management and Advantages," http://www.altera.com, 2010.

[34] M. Zied, M. Hayder, F. Umer, and M. Habib, "FPGA interconnect topologies exploration," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

[35] P. Jamieson, W. Luk, S. J. E. Wilton, and G. A. Constantinides, "An energy and power consumption analysis of FPGA routing architectures," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '09)*, pp. 324–327, December 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 295385, 15 pages
doi:10.1155/2011/295385

Research Article

A Middleware Approach to Achieving Fault Tolerance of Kahn Process Networks on Networks on Chips

Onur Derin,[1] Erkan Diken,[2] and Leandro Fiorin[1]

[1] *ALaRI, Faculty of Informatics, University of Lugano, 6900 Lugano, Switzerland*
[2] *Faculty of Electrical Engineering, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands*

Correspondence should be addressed to Onur Derin, derino@alari.ch

Received 28 August 2010; Revised 12 January 2011; Accepted 24 March 2011

Academic Editor: Michael Hübner

Kahn process networks (KPNs) is a distributed model of computation used for describing systems where streams of data are transformed by processes executing in sequence or parallel. Autonomous processes communicate through unbounded FIFO channels in absence of a global scheduler. In this work, we propose a task-aware middleware concept that allows adaptivity in KPN implemented over a Network on Chip (NoC). We also list our ideas on the development of a simulation platform as an initial step towards creating fault tolerance strategies for KPNs applications running on NoCs. In doing that, we extend our SACRE (Self-Adaptive Component Run Time Environment) framework by integrating it with an open source NoC simulator, Noxim. We evaluate the overhead that the middleware brings to the the total execution time and to the total amount of data transferred in the NoC. With this work, we also provide a methodology that can help in identifying the requirements and implementing fault tolerance and adaptivity support on real platforms.

1. Introduction

Past decade has witnessed a change in the design of powerful processors. It has been realized that running processors at higher and higher frequencies is not sustainable due to unproportional increases in power consumption. This led to the design of multicore chips, usually consisting of multiprocessor symmetric Systems on Chip (MPSoCs), with limited numbers of CPU-L1 cache nodes interconnected by simple bus connections and capable in turn of becoming nodes in larger multiprocessors. However, as the number of components in these systems increases, communication becomes a bottleneck and it hinders the predictability of the metrics of the final system. Networks on Chip (NoCs) [1] emerged as a new communication paradigm to address scalability issues of MPSoCs. Still, achieving goals such as easy parallel programming, good load balancing and ultimate performances, dependability and low-power consumption pose new challenges for such architectures. Technology scaling decreases the yield in manufacturing and thermal effects cause defects at run time. Thus fault tolerance becomes a major concern not only for economical reasons but also for the end user.

In addressing these issues, we adopted a component-based approach based on Kahn Process Networks (KPNs) for specifying the applications [2]. KPN is a stream-oriented model of computation based on the idea of organizing an application into streams and computational blocks; streams represent the flow of data, while computational blocks represent operations on a stream of data. KPN presents itself as an acceptable tradeoff point between abstraction level and efficiency versus flexibility and generality. It is capable of representing many signal and media processing applications, which occupy the largest percentage of the consumer electronics in the market.

Our eventual goal is to run a KPN application directly on an NoC platform with self-adaptivity and fault-tolerance features. It requires us to implement a KPN run-time environment that will run on the NoC platform and support adaptation and fault-tolerance mechanisms for KPN applications on such platforms. In order to achieve our goal, we propose the use of a self-adaptive run-time environment

(RTE) that is distributed among the tiles of the NoC platform. It consists of a middleware that provides standard interfaces to the application components allowing them to communicate without knowing about the particularities of the network interface and the communication network in general. Moreover, the distributed run-time environment can manage the adaptation of the application for high-level goals such as fault-tolerance, high performance, and low-power consumption by migrating the application components between the available resources and/or increasing the parallelism of the application by instantiating multiple copies of the same component on different resources [3, 4]. Such a selfadaptive RTE constitutes a fundamental part in order to enable systemwide self-adaptivity and continuity of service support [5].

In view of the goal stated above, we propose to use our SACRE framework [4] that allows creating self-adaptive KPN applications. In [3], we listed platform level adaptations and proposed a middleware-based solution to support such adaptations. In the present paper, we define the details of the self-adaptive middleware particularly for NoC platforms. In doing that, we choose to integrate SACRE with the Noxim NoC simulator [6] in order to realize functional simulations of KPN applications on NoC platforms.

An important issue regarding the NoC platform is the choice of the communication model. Depending on the NoC platform, we may have a shared memory space with the Non-Uniform Memory Access (NUMA) model or we may rely on pure message passing with the No Remote Memory Access (NORMA) model [7]. While NUMA case, the implementation of the KPN semantics is straightforward as long as the platform provides some synchronization primitives, on the NORMA case it represents the main challenge.

The remainder of this paper is organized as follows. Section 2 overviews related work. Section 3 explains details of the middleware in the NORMA case. Section 4 presents our implementation of the middleware by integrating SACRE and Noxim. Section 5 discusses an analytical model for the calculation of the communication traffic and computational time for a given application, while Section 6 presents evaluation results for a JPEG case study. Section 7 provides the requirements for the implementation of fault tolerance mechanisms and lists some application level fault tolerance patterns. Section 8 presents conclusions and future work.

2. Related Work

Kahn process networks (KPNs) is a widely studied distributed model of computation used for describing systems where streams of data are transformed by processes executing in sequence or parallel [2]. Previous research on the use of KPN in multiprocessor embedded devices have been mainly focusing on the design of frameworks which employ them as model for the application [8–10], and which aim at supporting and optimizing the mapping of KPN processes on the nodes of a reference platform [11, 12]. In [8, 9], different methods and tools are proposed for automatically generating KPN application models from programs written in Matlab or C/C++. Design space exploration tools and performance analysis are then usually employed for optimizing the mapping of the generated KPN processes on a reference platform. A design phase usually follows in which software synthesis for multiprocessor systems [10, 12], or architecture synthesis for FPGA platforms [8] is implemented.

Software synthesis relies on the high-level APIs provided by the reference platform for facilitating the programming of a multiprocessor system. The trend from single-core design to many-core design has forced to consider interprocessor communication issues for passing the data between the cores. One of the emerged message passing communication API is Multicore Association's Communication API (MCAPI) [13] that targets the intercore communication in a multicore chip. MCAPI is the light-weight (low communication latencies and memory footprint) implementation of message passing interface APIs such as Open MPI [14].

However, the communication primitives available with these message passing libraries do not support the blocking write operation as required by KPN semantics. Main features in order to implement KPN semantics are blocking read and, in the limited memory case, blocking write. Key challenge is the implementation of the blocking write feature. There are different approaches addressing this issue. In [15], a programming model is proposed based on the MPI communication primitives (MPI_send() and MPI_recv()). MPI_recv() blocks the task until the data is available while MPI_send() is blocking until the buffer is available on the sender side. Blocking write feature is implemented via operating system communication primitives that ensure the remote processor buffer has enough space before sending the message. Another approach is presented in [16], a network end-to-end control policy is proposed to implement the blocking write feature of the FIFO queues.

In this work, we propose an active middleware layer that implements the blocking write feature through *virtual connectors*, that are introduced in opposite directions to the original ones. This work introduces an approach which is based on a novel implementation of KPN semantics which can be employed on Network-on-Chip platforms. The use of this approach can be considered complementary to the aforementioned related work about generation and mapping of KPN processes, being in fact one of our goals the proposal and evaluation of a middleware that can be employed by these types of tool as a support for the programmability of the cores, when requesting adaptability of the applications.

Similarly to [16], we target as reference platform a NoC architecture. However, with respect to [16], instead of directly relying on the hardware NoC buffers for implementing the KPN FIFOs, we decided to maintain them implemented in software, in order to keep the flexibility needed in an adaptive application.

The *virtual connector* concept for implementing the blocking write feature can be considered in part similar to other general protocols in which data are pulled rather than pushed. An example can be, for instance, found in [17], in the case of the implementation of node-to-node communication in asynchronous NoCs. However, the work

in [17] mainly addresses the aspects of design and simulation of asynchronous NoC platforms, while in our case this type of protocol has been introduced, for the first time, as a modification of our middleware to support blocking writes in KPN, independently of the implementation of the hardware NoC platform below.

In this work, we also deal with the aspects of fault tolerance of KPNs. Fault tolerance has been object of a large amount of research, addressing both the hardware and software aspects of a platform [18]. When focusing on software techniques, several approaches were proposed, suggesting for instance the use of N-version software [19], self-checking software [20], Multiple-Task N-Modular Redundancy [21], or, more recently, the use of data and code duplications for detecting and correcting transient faults affecting the processor data segment and control flow duplication for correcting and detecting faults in the code segment [22, 23]. In KPNs, reconfiguration algorithms for dealing with single or multiple faulty nodes or channels were presented in [24, 25]. Algorithms are based on the redistribution of the faulty nodes actions to other nodes, as well as the input and output channels of the faulty nodes. A set of rules is presented that should be followed at the occurrence of faults in order to minimize data loss. The approach presented in this paper work can be considered complementary to the one presented in [24, 25]. Differently from [24, 25], this work presents in fact the implementation of fault tolerant techniques in KPNs that allow the detection and masking of faults in hardware elements running KPN processes and evaluates the associated costs. Reconfiguration and task remapping would be a consequence of the detection of a faulty hardware component.

Another fundamental contribution of our approach is moreover the separation of fault tolerance concerns from the functional application development. The proposed fault tolerance techniques can be applied at run time without manual intervention of the application developer.

3. Task-Aware Middleware

A KPN application consists of a set of parallel running tasks (application components) connected through nonblocking write and blocking read unbounded FIFO queues (connectors) that carry tokens between input and output ports of application components. A token is a typed message. Figure 1 shows a simple KPN application. Running a KPN application on an NoC platform would require to map the application components on the several tiles of the NoC platform.

Given the assumption that we have a heterogeneous NoC-based platform and that the application will be specified using the KPN model of computation, we need a middleware as a software layer that runs on the programmable cores and as a hardware wrapper for nonprogrammable cores. The main functionality of the middleware is to implement KPN semantics for the application tasks that are either software or hardware tasks. However, we put some more requirements due to the fault-tolerance goal. In this section the middleware requirements is discussed and a middleware solution is described for programmable and nonprogrammable cores that addresses those requirements.

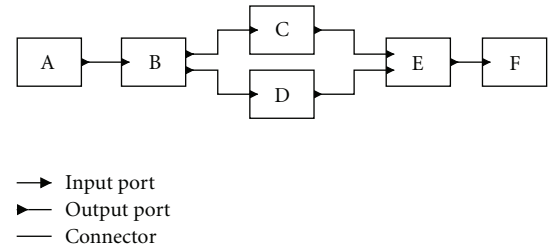


Figure 1: A simple KPN application with application components A, B, C, D, E, and F.

3.1. Requirements. When deciding on the implementation details of a middleware that will support running of a KPN application on a NoC platform, we came up with some requirements for the middleware. The most fundamental requirement for a middleware to support KPN semantics is the ability to transfer tokens among tiles assuring blocking read. Since unbounded FIFOs cannot be realized on real platforms, FIFOs have to be bounded. Parks' algorithm [26] provides bounds on the queue sizes through static analysis of the application. In the case of bounded queues, blocking write is also required to be supported.

Another requirement is that we would like to have platform-independent application components. This will make it easier to program for the platform by allowing the development of application components in isolation and running them without modifications. This can be achieved by separating the KPN library that will be used to program the application from the communication primitives of the platform. Middleware will link the KPN library to the platform specific communication issues.

In line with the above requirement, we would like that application components are not aware of the mapping of components on the platform. They should only be concerned with communicating tokens to certain connectors. Therefore the middleware should enable mapping-independent token routing. These requirements are of great importance if we want to achieve fault tolerance and adaptivity of KPN applications on NoC platforms in such a way that assures separation of concerns. This means that it is the platform that provides fault tolerance and adaptivity features to the application and not the application developer.

3.2. Middleware Implementation in the NORMA Case. In the NORMA model, tasks only have access to the local memory and there is no shared address space. Therefore tasks on different tiles have to pass the tokens among each other via message passing routines supported by the network interface (NI).

In order to address the middleware requirements previously listed, our key argument is the implementation of an active middleware layer that appears as a KPN task and gets connected to other application tasks with the connectors of

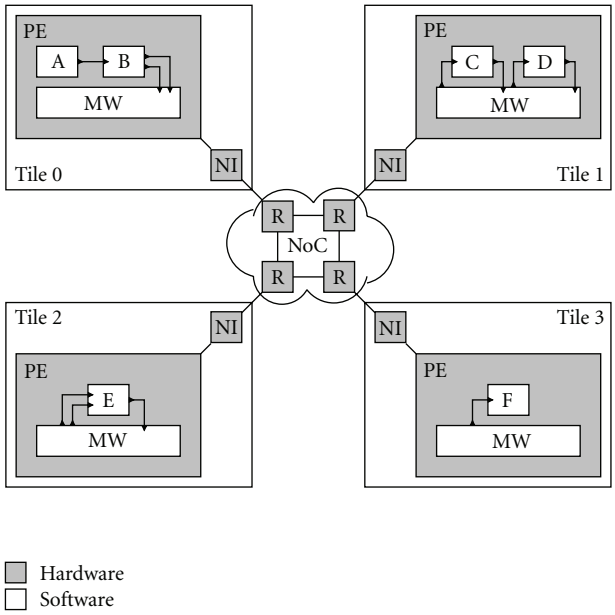


Figure 2: KPN example mapped on 2×2 mesh NoC platform.

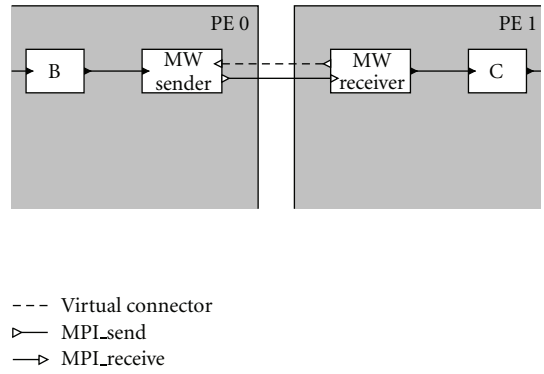


Figure 3: Middleware details for the connector between B and C.

Loop
  $t \leftarrow input\_port.read()$
  $MPI\_recv(t_{vc}, P_R, vc)$
  $MPI\_send(t, P_R, oc)$
end loop

Algorithm 1: Sender middleware task per outgoing connector (t: data token, t_{vc}: dummy token for virtual connector, P_R: process identifier of the remote middleware task, vc: virtual connector tag, oc: original connector tag).

the specific KPN library that is adopted by the application components. Opposedly, a passive middleware layer would be a library with a platform specific API for tasks to receive and send tokens.

We build our middleware on top of *MPI_recv()* and *MPI_send()* primitives. These methods allow sending/receiving data to/from a specific process regardless of which tile the process resides on. MPI_recv() blocks the process until the message is received from the specified process-tag pair. MPI_send() is nonblocking unless there is another process on the same tile that has already issued an MPI_send(). MPI_send() also blocks when the remote buffer is full.

Every tile has a middleware layer that consists of middleware sender and receiver processes. Figure 2 shows the middleware layers and a possible mapping of the example pipeline on four tiles of a 2×2 mesh NoC platform. There is a sender process for each outgoing connector. An outgoing connector is one that is connected to an input port of the application component that resides on a different tile. Similarly, there is a receiver process for each incoming connector. These processes are actually KPN tasks with a single port. This is an input port for a sender process and an output port for a receiver process. The job of a sender middleware task is to transmit the tokens from its input over the network to the receiver middleware task on the corresponding tile (i.e., the tile containing the application component to receive the token). Similarly, a receiver middleware task should receive the tokens from the network and put them in the corresponding queue. Figure 3 shows the sender and receiver middleware tasks between the ports of application components B and C.

We need to implement a blocking write blocking-read bounded channel that has its source in one processor and its sink in another one. MPI_send() as described above does not implement blocking write operation. It can be modified and be implemented in such a way that it checks whether the remote queue is full or not by using low-level support from the platform [15, 16]. In order to do this in a way that would not require changes to the platform, we make use of the *virtual connector* concept. A virtual connector is a software queue that is connected in the reverse direction to the original connector. For every connector between sender and receiver middleware tasks, we add a virtual connector that connects the receiver middleware task to the sender middleware task. Figure 3 shows the virtual connector along with the sender and receiver middleware tasks for the outgoing connector from application component B to C. The receiver task initially puts as many tokens to the virtual connector as the predetermined bound of the original connector. The sender has to read a token from the virtual connector before every write to the original connector. Similarly, the receiver has to write a token to the virtual connector after every read from the original connector. Effectively, the virtual connector enables the sender to never get blocked on a write. The read/write operations from/to original and virtual connectors can thus be done using MPI_send() and MPI_recv() as there is no more need for blocking write in presence of virtual connectors. Algorithms 1 and 2 show the pseudocodes for sender and receiver middleware tasks, respectively.

With the middleware layer, an outgoing blocking queue of bound b in the original KPN graph is converted into three blocking queues: one with bound $b1$ between the output

```
for i = 1 to connector_bound do
    MPI_send(t_vc, P_R, vc)
    i ← i + 1
end for
Loop
    MPI_recv(t, P_R, oc)
    MPI_send(t_vc, P_R, vc)
    output_port.write(t)
end loop
```

Algorithm 2: Receiver middleware task per incoming connector.

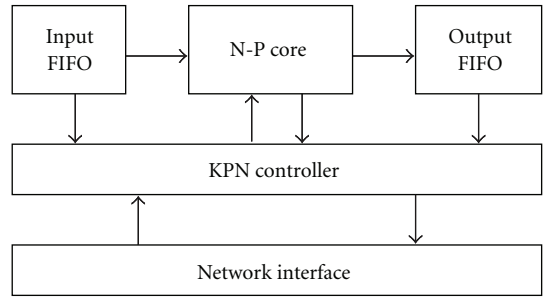


Figure 4: Block diagram of the KPN middleware for a nonprogrammable core (N-P Core).

port of the source component and the sender middleware task; one with bound $b2$ between the sender and receiver middleware tasks; one again with bound $b2$ between the receiver middleware task and the input port of the sink component. Values $b1$ and $b2$ can be chosen freely such that $b1 + b2 \geq b$ and $b1, b2 > 0$.

If the middleware layer is not implemented as an active layer, then the application tasks would need to be modified to include the virtual connectors. Moreover, use of virtual connectors enables us to not require changes to the NoC for custom signalling mechanisms.

Another benefit of having virtual connectors is in avoiding deadlocks. Since MPI_send() can be issued by different middleware tasks residing on the same tile in a mutually exclusive way, there may be deadlock situations for some of the task mapping decisions. For example, consider the case (see Figure 1 and the mapping in Figure 2) where an application task (C) is blocked on a call to MPI_send() until the queue on the receiver end is not full. It may be that the application task on the receiver end (E) is also blocked waiting for a token from an application task (D) on the tile where C resides. Since tasks on the same tile has to wait until the MPI_send() call of the other task returns, D cannot write the token to be received by E. Therefore we have a deadlock situation where C is blocked on E, E is blocked on D, and D is blocked on C. With virtual connectors, it is guaranteed that an MPI_send() call will not ever be blocked.

The problem of deadlocking can be solved also without using virtual connectors. However, that would require implementing expensive distributed conditional signalling mechanisms on the NoC or inefficient polling mechanisms.

3.3. KPN Middleware Wrapper for Nonprogrammable Cores. For the case of nonprogrammable cores, the middleware can be implemented as a hardware KPN wrapper. It will allow treating those cores as KPN tasks. This wrapper will include input and output FIFO buffers and KPN controller logic as shown in Figure 4. These additional units will reside along with the hardware core and the network interface (NI) in the tile. The functionality of the middleware as described in the previous section will be implemented by the KPN controller logic. It will initially send as many virtual tokens as the predetermined size of the input buffer. Whenever there is a new packet coming in, the NI will pass the received packet to the KPN controller which will extract the token to be put into the input buffer and signal this event to the KPN controller. In response, the controller will send back a virtual token and it will activate the hardware core to execute using the first token in the input FIFO. After the completion of the execution, the controller will wait until it receives a virtual token before writing the result to the output buffer. The KPN controller will send the tokens in the output FIFO by wrapping them into middleware packets by adding their destination component information and passing them over to the NI. The KPN controller may possibly interact with another controller that implements the self-checking and/or self-testing policies or its functionality may be extended for those purposes. Thus the KPN controller, when feeding nominal input to the core has to work in accordance with the specific interface of the nonprogrammable core.

4. Middleware Implementation

In order to realize functional simulation of the proposed system, and testing its functionalities, we integrated our SACRE tool with an NoC simulator.

Noxim [6] is an open source and cycle-accurate simulator developed in SystemC. It can generate NoC models by setting several configuration parameters such as the network size, the routing algorithm, and the traffic type. Models generated implement wormhole control flow, in which NoC packets are divided in an arbitrary number of flits which all follow the route taken by the first one which contains the header of the packet and which specifies the destination address. The NoC models generated by Noxim allow to analyze and evaluate a set of quality indices such as delay, throughput, and energy consumption.

SACRE [4] is a component framework that allows creating self-adaptive applications based on software components and incorporates the Monitor-Controller-Adapter loop [5] with the application pipeline. The component model is based on KPN. It supports run-time adaptation of the pipeline via parametric and structural adaptations.

4.1. Integration of SACRE and Noxim. The integration of SACRE and Noxim is done in order to be able to simulate

Table 1: Port connection table for Figure 1.

Source		Destination	
component	port	component	port
A	out1	B	in1
B	out1	C	in1
B	out2	D	in1
C	out1	E	in1
D	out1	E	in2
E	out1	F	in1

Table 2: Component mapping table for Figure 2.

Component	Tile
A	0
B	0
C	1
D	1
E	2
F	3

KPN applications on NoC platforms. The proposed middleware is implemented for the NORMA case. First of all, we do not have the MPI_send() and MPI_recv() primitives in Noxim. Actually the simulator does not even implement a proper network interface, while it provides traffic generators and sinks. We implemented the transport layer such that we can send data and reconstruct the data on the other end. In absence of MPI primitives in SACRE-Noxim, we implemented the task-aware middleware over the transport layer of the NoC network interface as described below.

We conceived the middleware as a KPN task by extending it from *SACREComponent* in order to be able to connect it to the queues of the local application tasks. In the SACRE framework, a SACREComponent is the base class for all the KPN tasks and allows them to specify their input/output ports and functions [4]. To have a complete communication scheme, two types of middleware tasks are implemented, *MWSender* and *MWReceiver*.

Implementation details of the SACRE and Noxim integration is depicted in Figure 5. We will refer to this scheme in order to explain the details of the integration. It is assumed that two application components are mapped on two different tiles of the NoC. One of the application components, which is mapped to tile 0, produces tokens, while the other application component, which is accommodated on tile 1, consumes token.

On the sender side, since middleware tasks and application components are SACRE components and are connected with blocking queues, the producer application component writes tokens into the queue that resides between producer component and *MWSender* task. Blocking queues block the SACRE components in case of an attempt to read from an empty queue or an attempt to write into a full queue. *MWSender* uses MPI_send() primitive in order to send tokens to the destination application component after reading tokens from its corresponding input port. MPI_send() has the signature shown in Algorithm 3. As *MWSender* task is independent from the application components, it accepts generic data types. Furthermore, a *port connection table* is accessed so as to get the destination component name and destination port identifier which are passed as arguments to the MPI_send() call. As shown in Table 1, the port connection table represents the KPN application and shows which components and which ports are connected with each other.

When there is a token to be forwarded, MPI_send() is called. MPI_send() wraps the token into a *MWPacket* object as shown in Figure 6 by adding the destination task name and destination port name as the header information. Since, only information related to destination is passed as parameters to the MPI_send(), in addition to these information, *port connection table* is accessed in order to add source component name and source port name to the *MWPacket*.

In the following step, the *MWPacket* object is sent via the NI to the destination tile by wrapping it in a *NIPacket* object. *NIPacket* has the structure shown in Figure 6. The destination tile identifier is looked up from the *component mapping table*. This table stores which components reside on each tile as shown in Table 2. Currently NI transfers packets splitting them into several flits through wormhole routing.

On the receiver side, the network interface receives the flits and transport layer reconstructs the *NIPacket* object. Then the receiver process extracts the *MWPacket* from *NIPacket* and puts it in the *receive list*. This receive list is accessed by MPI_recv() primitive which is called by *MWReceiver* tasks. MPI_recv() has the signature shown in Algorithm 3.

MWReceiver tasks are connected to the application components which have input ports for receiving tokens. Similarly to the *MWSender*, output ports of the *MWReceiver* tasks are connected to the input ports of the application components through blocking queues. Differently from *MWSender* tasks, *MWReceiver* tasks use MPI_recv() primitive in order to read tokens from the receive lists of the corresponding processing element. *Port connection table* is accessed to get the source component name and source port which are used to check the receive list in order to identify the received packet. Then, token is passed to the application component by writing the token into the blocking queue via corresponding port.

4.2. Case Study: Simulation of JPEG Encoder and Decoder. In order to verify the correctness of the integration of SACRE and Noxim, we run a KPN-modeled application on the Noxim simulator. Components of the application were mapped on different tiles of NoC. In order to do that, as a first step, a JPEG Encoder & Decoder application is implemented by using the SACRE KPN library. This allows to verify the functional correctness of the application before running it on the NoC platform. Figure 7 depicts the application components of the JPEG Encoder & Decoder.

Second step requires mapping the application components on the tiles of NoC. Figure 8 shows one possible mapping of the JPEG application components on different

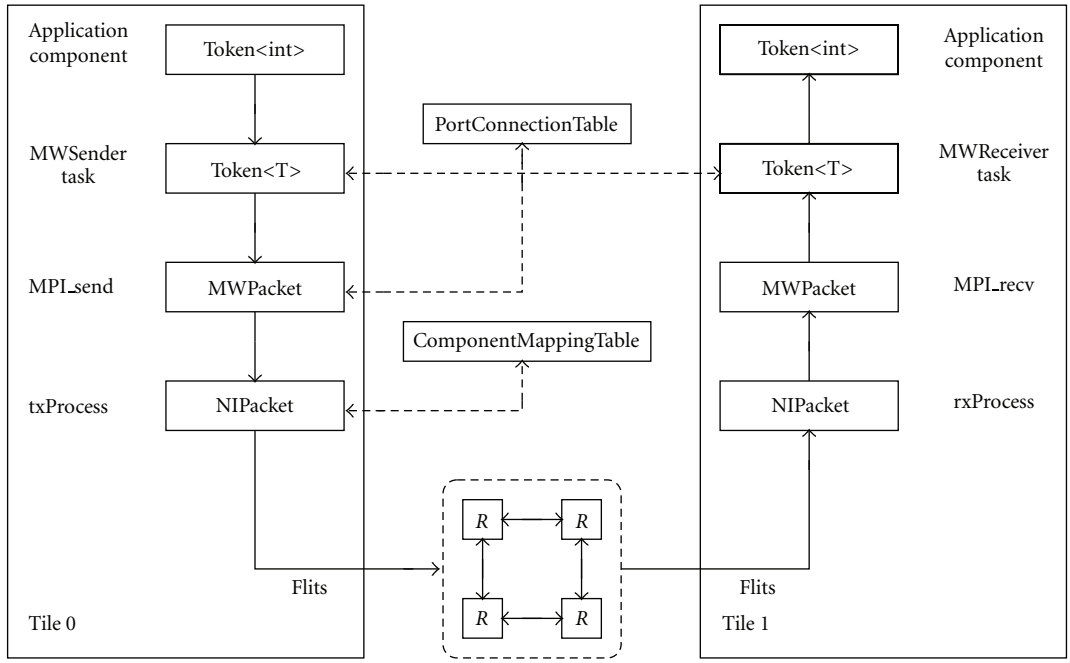


Figure 5: Implementation details of the SACRE and Noxim Integration.

$$MPI\_recv(recv\_buffer, size, src, tag)$$
$$MPI\_send(send\_buffer, size, dest, tag)$$

Algorithm 3: Signatures of MPI_recv() and MPI_send() (*recv_buffer*: memory to allocate received token, *send_buffer*: allocated memory for token to be sent, *size*: size of the data token, *src*: source component, *dest*: destination component, and *tag*: port identifier).

tiles of the 4×4 NoC. Application is mapped such that all communicating components have a Manhattan distance of one. It is also possible to map more than one component on one tile of NoC. The yellow arcs refer to the data flow between components.

Application mapping is accomplished by simply inserting component and tile information into the *component mapping table*. Additionally, it is required to insert information about the connection of the components into the *port connection table* by providing input and output port identifiers with component names. The next step requires to create the application components and to connect them with their related middleware tasks. It is important to note that bound of the blocking queues that reside between components and middleware tasks is provided during that connection phase. Similar to the bounds on blocking queues, number of the virtual tokens are decided when each *MWReceiver* task is created.

Functional correctness is tested by giving a gray scale image as an input, after encoding and decoding, an output image is produced as an output at the end of the simulation. Moreover, as stated previously, Noxim simulator provides statistics in order to be able to evaluate the interconnection network traffic.

5. Middleware Analytical Models

In this Section, we present and discuss an analytical model for calculating the communication load and the execution time associated with an application, in order to be able to evaluate the proposed middleware in terms of the overhead it introduces. In particular, as shown in Section 6, we focus our evaluation on the JPEG case study.

The following analytical model is proposed.

(i) A *KPN task graph* $g_t = (V_t, E_t)$ is composed of tasks $t \in V_t$ and data dependencies (connectors) $e \in E_t \subseteq V_t \times V_t$.

(ii) An *architecture graph* $g_a = (V_a, E_a)$ is composed of processing nodes $n \in V_a$ and bidirectional communication links $l \in E_a \subseteq V_a \times V_a$.

(iii) A *task mapping* $\beta_t : V_t \rightarrow V_a$ is an assignment of tasks $t \in V_t$ to nodes $n \in V_a$.

(iv) A *communication mapping* $\beta_c : E_t \rightarrow E_a^i$ is an assignment of data dependencies $e \in E_t$ to paths of length i in the architecture graph g_a. A *path* p of length i is given by i-tuple $p = (l_1, l_2, \ldots, l_i)$.

(v) $path : (V_a, V_a) \rightarrow E_a^i$ is a function that implements a deterministic routing algorithm and returns a path between two given nodes. *Path set* P is the set of paths between all node pairs:

$$P = \left\{ p_k : p_k = path\left(n_i, n_j\right), \forall n_i, n_j \in V_a \wedge n_i \neq n_j \right\}. \quad (1)$$

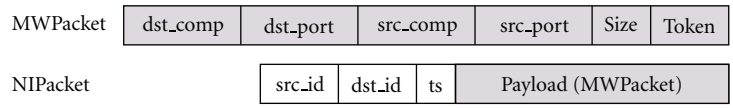


Figure 6: Structure of packets at the NI and middleware levels (src_id: source tile, dst_id: destination tile, ts: timestamp, dst_comp: destination component, dst_port: destination port, src_comp: source component, and src_port: source port).

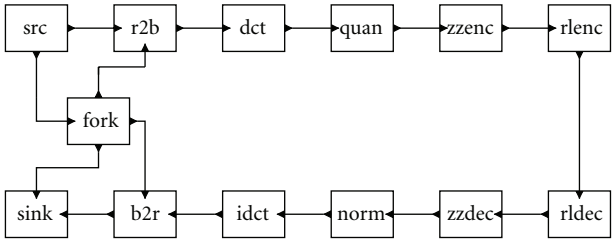


Figure 7: JPEG Encoder & Decoder implemented with SACRE KPN library.

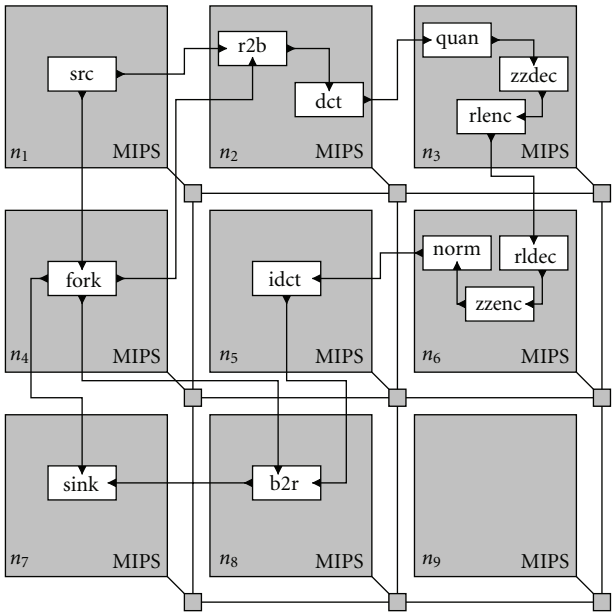


Figure 8: Mapping of JPEG application components on 3×3 NoC.

Initial and final nodes of a path can be obtained by *source* and *sink* functions

$$p_k = path\left(n_i, n_j\right) \Longrightarrow source(p_k) = n_i \wedge sink(p_k) = n_j. \tag{2}$$

(vi) The task graph can be annotated with demand values where *demand* d_i on a data dependency $e_i \in E_t$, denotes the amount of data transferred between the two tasks. Demand values are obtained considering the network protocol of the platform spanning all layers from application and middleware to the network interface. At the application level, the number of writes w_i for each connector e_i is calculated by profiling the application with a test input. Let s_i be the size (in bytes) of the data type of the data transferred on connector e_i. Then total number of bytes transferred at the application level on connector e_i is

$$d_i^{\text{app}} = w_i \times s_i. \tag{3}$$

At the middleware level, each token of the application is augmented with the MW header data (see Figure 6). Let $h_m w$ be the size of the header section of the *MWPacket*. The amount of data to be supplied to the network interface by the middleware is

$$d_i^{mw} = w_i \times (s_i + h_{mw}). \tag{4}$$

Finally, at the NI level, *NIPacket*s are created by adding the header information (see Figure 6) to the *MWPacket* and transfer the packet in flits of size f bytes. Let h_{ni} be the size of the header section of the *NIPacket*. Then the total number of bytes transferred at the NI level for connector e_i is

$$d_i^{ni} = w_i \times \left(\left(\left\lceil \frac{s_i + h_{mw}}{f} \right\rceil \times f\right) + h_{ni}\right). \tag{5}$$

Due to the virtual connector that is connected in the reverse direction, there is an additional transfer of a virtual token for every nominal token. We represent the virtual token as a single header flit in the network, thus incurring an additional traffic of $w_i \times h_{ni}$. The number of bytes transferred at the NI level for connector e_i with our middleware is

$$d_i = w_i \times \left(\left(\left\lceil \frac{s_i + h_{mw}}{f} \right\rceil \times f\right) + 2 \times h_{ni}\right). \tag{6}$$

If the task mapping is fixed and no task will be migrated at run time, tasks can be programmed by embedding the mapping information in them in which case the middleware would not be needed. In such a case, the number of bytes transferred for connector e_i without a middleware is

$$d_i = w_i \times \left(\left(\left\lceil \frac{s_i}{f} \right\rceil \times f\right) + h_{ni}\right). \tag{7}$$

(vii) Core type set C consists of core types C_i and lists the types of cores available in a given NoC platform.

5.1. Communication Cost. In order to formulate the analytical model for the calculation of total amount of data transferred in the network during the execution of the

application, we define several application-specific parameters, architecture-specific parameters, and mapping-specific variables in the form of incidence matrices.

X^{NT} is an incidence matrix of size $|V_a| \times |V_t|$ that denotes the mapping of tasks onto the nodes.

$$X_{ij}^{NT} = \begin{cases} 1 & \text{if } t_j \in V_t \text{ is bound onto node } n_i \in V_a, \\ 0, & \text{otherwise .} \end{cases} \tag{8}$$

Y^{PE} is an incidence matrix of size $|P| \times |E_t|$ that denotes which path realizes which data dependency

$$Y_{ij}^{PE} = \begin{cases} 1 & \text{if } e_j \in E_t \text{ is mapped to } p_i \in P, \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

M^{TE} is an oriented incidence matrix of size $|V_t| \times |E_t|$ that relates the tasks to the data dependencies. For a given task graph, M^{TE} is known

$$M_{ij}^{TE} = \begin{cases} 1 & \text{if } \exists t_k, e_j = (t_i, t_k) \in E_t, \\ -1 & \text{if } \exists t_k, e_j = (t_k, t_i) \in E_t, \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

M^{PL} is an incidence matrix of size $|P| \times |E_a|$ that denotes the relation between all paths resulting from a given deterministic routing algorithm and the links that make up the path. For a given routing algorithm and architecture graph, M^{PL} is known.

$$M_{ij}^{PL} = \begin{cases} 1 & \text{if } l_j \in p_i, \\ 0, & \text{otherwise.} \end{cases} \tag{11}$$

Total Amount of Data Transferred. The total amount of data transfer, d_{tot}, on the links can be calculated as the sum of all demands d_i on the links of the paths that arise according to a given mapping with the following equation:

$$d_{\text{tot}} = d^T \; Y^{EP} M^{PL} \; \mathbf{1}_{|E_a|}, \tag{12}$$

where $\mathbf{1}_m$ is a matrix of size $m \times 1$ with all elements equal to 1. It is to be noted that $Y^{EP} = (Y^{PE})^T$. Similar relation holds for all defined matrices.

This static model for communication has also been used in [27] and disregards the congestion on the links. However at low load conditions, it is argued that it is a good approximation.

Note that the communication cost takes into account the intertile communication done over the NoC between tasks and not the intratile communication when communicating tasks are mapped onto the same node.

5.2. Computational Cost. In order to formulate the model for calculating total execution time of the application, we define additional parameters in matrix form, namely, M_{cap}^{TC}, T_{cap}^{TC}, and M^{NC}.

M_{cap}^{TC} is an incidence matrix of size $|V_t| \times |C|$ that denotes which core types are capable of realizing which tasks. Programmable cores would be capable of realizing different kinds of task functionalities, whereas nonprogrammable cores would have dedicated functions

$$M_{\text{cap}\,ij}^{TC} = \begin{cases} 1 & \text{if } C_j \in C \text{ is capable of realizing task } t_i \in V_t, \\ 0, & \text{otherwise.} \end{cases} \tag{13}$$

T_{cap}^{TC} is a matrix of size $|V_t| \times |C|$ that denotes the completion times of all tasks on all core types for a test input. Given an application and architecture, this matrix can be obtained by offline profiling

$$T_{\text{cap}\,ij}^{TC} = \begin{cases} \text{completion time of } t_i \text{ on } C_j & \text{if } M_{\text{cap}\,ij}^{TC} = 1, \\ 0 & \text{if } M_{\text{cap}\,ij}^{TC} = 0. \end{cases} \tag{14}$$

M^{NC} is an incidence matrix of size $|V_a| \times |C|$ that denotes the core type of the architectural nodes. Given an architecture, M^{NC} is known

$$M_{ij}^{NC} = \begin{cases} 1 & \text{if } n_i \in V_a \text{ is of core type } C_j \in C, \\ 0 & \text{otherwise.} \end{cases} \tag{15}$$

M^{TC} is an incidence matrix of size $|V_t| \times |C|$ that denotes the actual mapping of tasks on core types. Given a task to node mapping matrix, X^{TN}, it can be calculated as

$$M^{TC} = X^{TN} M^{NC}. \tag{16}$$

T^T is a matrix of size $|V_t| \times 1$ that denotes the completion time of the task on the core type that it is mapped onto. It can be calculated as

$$T^T = \left(M^{TC} \cdot T_{\text{cap}}^{TC}\right) \mathbf{1}_{|C|}, \tag{17}$$

where the "$\cdot$" operator represents elementwise matrix multiplication.

T^N is a vector of size $|V_a| \times 1$. T_i^N denotes the sum of execution times of tasks that are mapped on the same node, n_i. It can be calculated as

$$T^N = X^{NT} T^T. \tag{18}$$

Total Execution Time of Tasks. We calculate the total computation time of tasks by finding the maximum of the sum of the execution times of tasks mapped on the same core.

$$\max\left(T^N\right) = \max\left(X^{NT}\left(\left(\left(X^{TN} M^{NC}\right) \cdot T_{\text{cap}}^{TC}\right)\mathbf{1}_{|C|}\right)\right), \tag{19}$$

where max is a function that returns the maximum value in a given vector.

This is a static model that has been also used in [28] and disregards the context switching times. This model can be considered to have a reasonable accuracy in the case of

typical streaming applications when there is enough inter- and intra-process parallelism such that the resources do not stall because of blocking and when communication is smooth.

However, it is of particular importance in our case to incorporate the context switching times because the proposed middleware tasks work as threads. In order to calculate the number of context switches on a core, we adopt a scheduler that executes a task until it is blocked on a read or a write. In the particular case of m pipelined tasks (t_1 to t_m) with single input and output connectors mapped on the same node n_i where t_j is connected to t_{j+1} through the bounded connector (t_j, t_{j+1}) with equal sized bounds b, the number of context switches in node n_i can be calculated as

$$N_{CSi} = m \times \left\lceil \frac{w}{b} \right\rceil - 1, \tag{20}$$

where w is the number of writes to a connector and assumed to be equal for all connectors.

For the particular mapping of the JPEG application in our case study, this formula will suffice because the mapping conforms to the assumptions of the above analysis (see Figure 8). We leave it as an open issue to calculate the number of context switches for general mappings and schedulers.

The total time T_{CSi}, spent in making N_{CSi} context switches can be calculated by

$$T_{CSi} = N_{CSi} CS_{\text{cap}\,i}, \tag{21}$$

where CS_{cap_i} is the single context switching time of the core type of node n_i.

With the inclusion of context switching time, the total execution time of the application is

$$T^{\text{app}} = \max\left(T^N + T_{CS}\right). \tag{22}$$

6. Evaluation of Results

In this section, we evaluate the overhead associated with the use of the middleware in the case of the JPEG case study. Being our work a first attempt to evaluate the cost of using a middleware layer for supporting the adaptivity of the application, we cannot directly compare our results with those obtained in previous work adopting the KPN model of computation. We therefore present and discuss the costs of adding this additional layer. We calculate the computation and communication overhead by means of the analytical model presented in Section 5. The computational overhead is interpreted as the increase in the total execution time of an application. In the case of our case study, we look at the increase in the processing time of a single image. The communicational overhead is interpreted as the increase in the communication cost which is defined as the total amount of data transferred in the interconnection network.

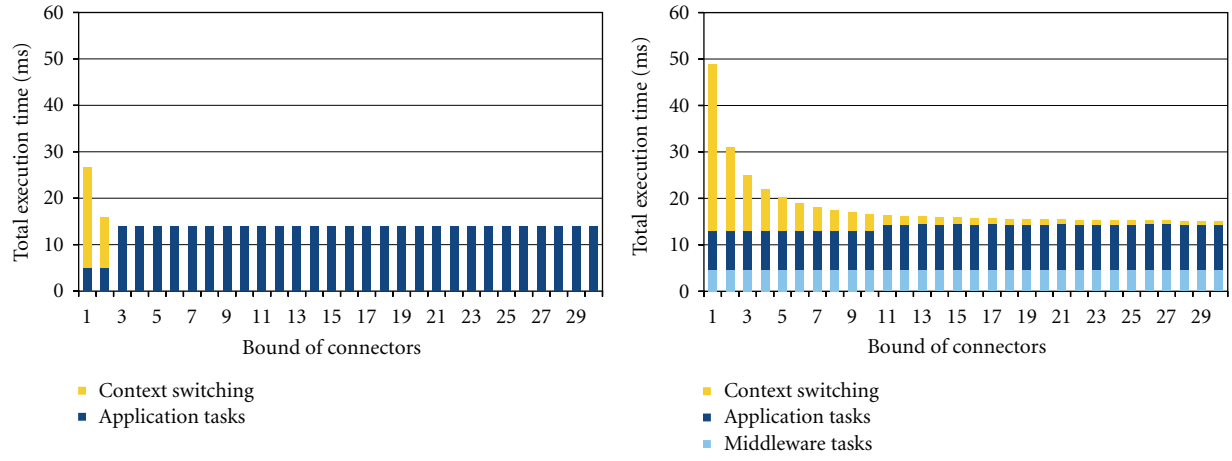
6.1. Communication Overhead. In the case of our middleware, allocating 1 byte for each of *dst_comp*, *dst_port*, *src_comp*, *src_port* headers and 4 bytes for the *size* header, we have $h_{mw} = 8$. In our model of the NoC, we consider a datawidth of the physical link of 32 bits. We assume an XY-routing algorithm, and a wormhole control flow, in which each packet is divided in an arbitrary number of flits, whose dimension is equal to the width of the physical link. The flits of a packet follow the route taken by the first one, into which routing information are inserted (header). Our network interface has therefore an *NIPacket* header of size 1 flit, thus $h_{ni} = f = 4$ bytes. In the JPEG application, all the connectors are either of data type *Token<int>* or *Token<float>* which are both 8 bytes giving $s_i = 8$ bytes for all connectors e_i.

Evaluating (6), (7), and (12) for the mapping of JPEG application shown in Figure 8, we obtain a communication overhead of 1.00, meaning double amount of data transfer in the network, when compared to the transmission of the information without the use of the middleware. As the size of the data type of tokens transferred on the connectors increases, the overhead due to the middleware decreases. In that respect, the JPEG application with only 8-byte-long tokens gives a higher communication overhead.

Figure 10(a) shows the overhead for varying token sizes (in bytes) in case these would be the token sizes in the JPEG case study. It can be seen that the communication overhead falls under 10% when the tokens are bigger than 116 bytes. It should be noted that token size is an application-dependent parameter, and depending on the application, the application programmer can modify the application to work with bigger token sizes with additional effort. Although increasing token size results in lower communication overhead, due to the pipelined propagation of wormhole switching the communication could not be considered as ideal anymore, and delay due to the possible contention should be taken into account in the choice of the right token dimension [29]. Another possibility could imply the propagation of the same token by using several network packets, therefore taking into consideration in the calculation of the communication overhead the traffic generated by the header flits of the additional packets employed.

6.2. Computational Overhead. We profiled the JPEG encoder/decoder application by using the SESC simulator [30] with a test input image of 211 × 205 pixels and by considering a MIPS processor running at 5 GHz. As done for the calculation of the communication overhead, we consider tokens of 8 bytes. The execution times of the application tasks are shown in Table 3. The execution times of the middleware tasks for each connector are shown in Table 4. In the tables, values of execution time that are equal to 0.00 are due to the rounding to the second decimal number. As can be noticed, the execution time of the middleware tasks increases with the amount of data they transfer. The number of writes to the connectors are shown in Table 5. We used the mapping and the XY-routing-based NoC platform shown in Figure 8 and a context switching time of 0.16 μs for the processor.

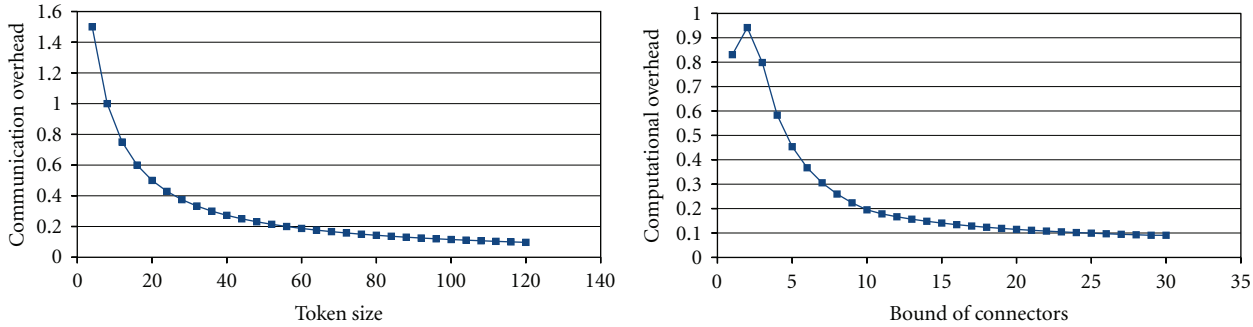
Using (22) and varying the bound of the connectors, we obtained the overhead results shown in Figure 10(b). The results show that for $b > 24$, the computational overhead can



(a) The case without middleware with varying sizes of KPN queues

(b) The case with middleware with varying sizes of KPN queues

Figure 9: The distribution of total execution time between application tasks, middleware tasks, and context switching for the JPEG encoder/decoder case study.



(a) Overhead in the total data transferred in the network due to proposed middleware with varying sizes (in bytes) of the data type of tokens

(b) Overhead in the execution time due to proposed middleware with varying sizes of KPN queues (tokens of 8 bytes)

Figure 10: The communication and computational overhead due to the proposed middleware for the JPEG encoder/decoder case study.

Table 3: Execution times (in *ms*) of tasks on the available core types (T^{CT}_{cap}).

Core type	Tasks												
	src	fork	r2b	dct	quan	zzenc	rlenc	rldec	zzdec	norm	idct	b2r	sink
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}
MIPS	13.89	0.00	1.84	6.43	1.81	2.01	1.35	1.12	1.95	1.81	9.08	1.81	8.75

Table 4: Execution times (in *ms*) of MWSender and MWReceiver tasks for each connector.

Core type	src → fork	fork → r2b	fork → b2r	fork → sink	src → r2b	dct → quan	rlenc → rldec	norm → idct	idct → b2r	b2r → sink
	$e_{(1,2)}$	$e_{(2,3)}$	$e_{(2,12)}$	$e_{(2,13)}$	$e_{(1,3)}$	$e_{(4,5)}$	$e_{(7,8)}$	$e_{(10,11)}$	$e_{(11,12)}$	$e_{(12,13)}$
MIPS	0.00	0.00	0.00	0.00	2.24	2.35	0.82	2.35	2.33	2.24

Table 5: Number of writes to the connectors (w).

$e_{(1,2)}$	$e_{(2,3)}$	$e_{(2,12)}$	$e_{(2,13)}$	$e_{(1,3)}$	$e_{(3,4)}$	$e_{(4,5)}$	$e_{(5,6)}$	$e_{(6,7)}$	$e_{(7,8)}$	$e_{(8,9)}$	$e_{(9,10)}$	$e_{(10,11)}$	$e_{(11,12)}$	$e_{(12,13)}$
3	3	3	3	43255	44928	44928	44928	44928	15682	44928	44928	44928	44928	43255

be lower than 10%. Obviously the queue size can be increased subject to the availability of memory in the tiles.

Looking deeper into the distribution of the total execution time among the application tasks, middleware tasks, and the context switching, we obtained the results shown in Figure 9. In the case without the middleware, the critical node that determines the total execution time is n_3 for $bound \leq 2$ and n_1 for $bound > 2$. Since n_1 is mapped only one task (i.e., *src*), we do not see any time spent for context switching. As the connector bound is increased, n_1 becomes the critical node rather than n_3 due to the decreasing context switching penalty on n_3.

In the case with the middleware, the critical node is n_2 for $bound \leq 10$ and n_5 for $bound > 10$. The change of critical node is again due to the decreasing context switching times when the size of the KPN queues is increasing.

7. Fault Tolerance Support

Having isolated the application tasks from the network interface, we believe it will be easier to implement fault tolerance mechanisms, mainly based on tasks migration. Below we describe the task migration support over the proposed middleware and list a number of self-checking patterns that are enabled by the task migration support. We give implementation details for the adoption of these patterns at KPN level. These patterns have been implemented and validated with SACRE framework.

The proposed patterns can be applied as run-time adaptations [4] enabling an adaptive degree of fault tolerance. They enable adaptive dependability levels for different parts of the application pipeline at the granularity of an application component. Besides conventional adaptation parameters such as voltage level and frequency, this allows to increase fault tolerance characteristic of the application in presence of excess power or resources. Typically, in self-adaptive systems, the application programmer provides a set of goals (e.g., performance, power) to be met by the application [5]. These goals are translated into parameters to be monitored by the self-adaptive platform. The adaptations are driven by a self-adaptation control mechanism that tries to meet the goals by monitoring those parameters. The proposed self-checking patterns, if applied at run time, enhance the self-adaptive platform to accept fault tolerance as a goal.

In our discussion, we mainly assume a functional fault model which considers a processing element as *working*, *nonworking*, or *partially working*. In case of a *working* processing element, the module is fully functional; in *partially working* ones, some errors have been detected, and the module is assumed to be faulty. However, the core has a relevant complexity and modularity (e.g., a processor with the floating point pipeline as well as with the fixed point one) and identification and confinement of a specific faulty subunit and degradation of the core to the surviving functions can be applied (see, e.g., [31]. However, the module can still be used, even if partial degradation of the functionality and of the performances must be taken into account; for *nonworking* processing elements, the module is considered too faulty to be operative.

The fault tolerance patterns discussed in the following part of the section can be used to detect the operativeness of the processing element, as well as providing methods for detecting and correcting errors due to faulty behaviors of the components. Once the level of operativeness is detected, and depending on the faults detected, tasks migration can be used to move information from a nonworking (or a partially working) processing core to a working one.

7.1. Task Migration. In the case when a tile fails, the tasks mapped on that tile should be moved to other tiles. Therefore we will have a controller implementing a task remapping strategy. For now, we do not focus on this but rather highlight how the proposed middleware eases the implementation of task migration mechanisms.

Moving an application component from one tile to another requires the ability to start the task on the new tile, update the component mapping tables on each tile, create/destroy MW tasks for outgoing and incoming connectors of the migrated components, and transfer the tokens already available in the connectors of the migrated components along with those components. In case of a fault, the tokens in the queues pending to be forwarded by the middleware tasks in the failed tile may be lost along with the state of the task if it had any. Similarly, there may be some number of received flits that have not been reconstructed to make up a packet yet. We may need to put in measures to checkpoint the state of the task and the middleware queues. As a rollback mechanism, we should be able to transfer both the state of the tasks and the queues on the faulty tile to the new tiles. The flits already in the NoC buffers destined to the faulty tile should be rerouted to the new tiles accordingly. This may be easier to achieve if we implement the task-awareness feature in the NoC routers. Otherwise, it should be the NI or the router of the faulty tile that should resend those flits back in the network with correct destinations. We need to further analyze the scenarios according to the extent of faults (e.g., only the processing element is faulty or whole tile is faulty). However, thanks to the middleware layer, application tasks will not need to know that there has been a task migration.

7.2. Triple Modular Redundancy Pattern at Application Level. The run-time environment can adapt the dependability at the application level. This enables adaptive dependability levels for different parts of the application pipeline at the granularity of an application component.

In the case of a single component, parallel instances of the component are created on different cores along with multiplicator components and majority voter components for each input and output ports, respectively, as shown in Figure 11. Multiplicator component creates a copy of the incoming message for each redundant instance and forwards each copy to the input ports of those instances. Majority voter component reads a token from all of its input ports and finds out the most recurrent token and sends it to its output connector. If there is a different input token then this

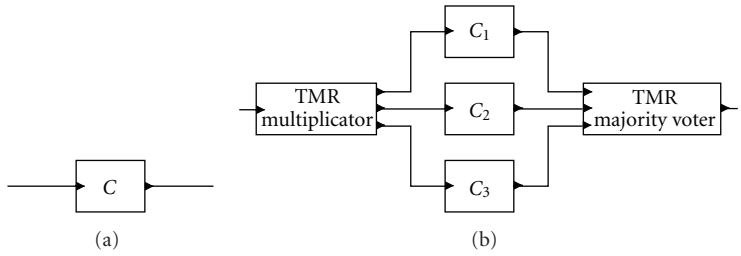


Figure 11: Triple modular redundancy adaptation pattern applied at KPN level. KPN task C in (a) is replicated by three as shown in (b).

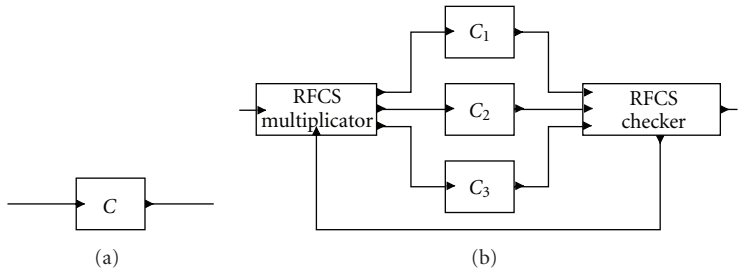


Figure 12: RFCS adaptation pattern applied at KPN level.

component can signal that the core producing that token is faulty. A timeout mechanism can also be put in place to tolerate when a core is faulty and no message is being received by a component.

7.3. Roll-Forward Checkpointing Scheme (RFCS) Pattern at Application Level. A lower power version of the TMR is the duplex with spare technique also referred to as Roll-Forward Check-pointing Scheme (RFCS) [32]. We propose to employ it for a KPN task as shown in Figure 12. In this scheme, the RFCS multiplicator component forwards the copy of the token to the first two components (C_1 and C_2). After C_1 and C_2 execute, RFCS checker component reads the tokens from its first two input ports. If the tokens are the same, it forwards either one of the tokens and sends a *no-fault* token back to the multiplicator. The multiplicator keeps forwarding the tokens to its first two output ports as long as it reads a *no-fault* token before forwarding. If there is a fault in either of C_1 and C_2, then the checker will forward one of the resulting tokens and signal back to the multiplicator that there has been a fault by sending a *fault-present* token. If the multiplicator receives such a token, it does not consume any tokens from its input port, instead, it forwards two copies of the last read token to C_2 and C_3. After having sent the *fault-present* token, in the next round, the checker component reads the tokens from its second and third input ports. Comparing the results from the previous mismatch (between C_1 and C_2) and the current round (C_2 and C_3), the checker component can detect the faulty core.

7.4. Parallelization Pattern at Application Level. An example of a structural adaptation in order to meet performance and low-power goals is the parallelization pattern explained below.

Parallelization of a component is one type of structural adaptation that can be used to increase the throughput of the system as shown in Figure 13. This is done by creating parallel instances of a component and introducing a router before and a merger after the component instances for each of the input and output ports. A router is a built-in component in our framework that works in a round-robin fashion; this component routes the incoming messages to the parallel instances sequentially. The merger components simply merge the output messages from the output ports of the instances into one connector by reading again in a round-robin fashion from their input ports. For the general class of KPN applications, semantics require that the processes comply with the monotonicity property. The round-robin policy of the router and the merger preserves the ordering relation among tokens. However the condition for applicability of such an adaptation is the absence of intermessage dependencies.

7.5. Semiconcurrent Error Detection Pattern at Application Level. We propose to employ semiconcurrent error detection (SCED) [33] as a dependability pattern for self-adaptivity. This pattern is used on top of the parallelization pattern. It can be used in the case where there are already multiple parallel instances for performance reasons and we want fault tolerance on top of it. The basic idea is to have a redundant component instance (C_r) that is used to process the same token as one of the other parallel instances in a round robin fashion. We make a statistical assumption that fault rate and distribution of nominal input data values are such that with very high probability occurrence of a fault will be detected by the input data before occurrence of a second, independent fault. (In other words, coverage of all single faults in a unit is granted by the flow of nominal data within the time

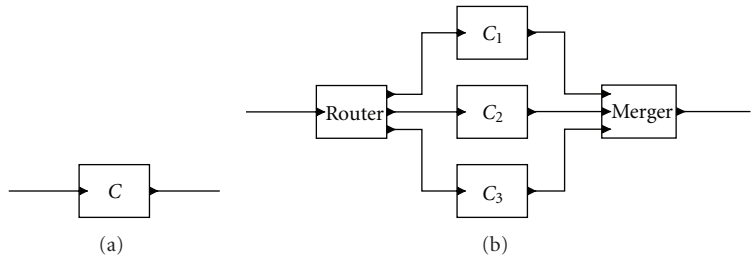


Figure 13: Adaptation pattern for parallelization. Component in (a) is parallelized by three as shown in (b).

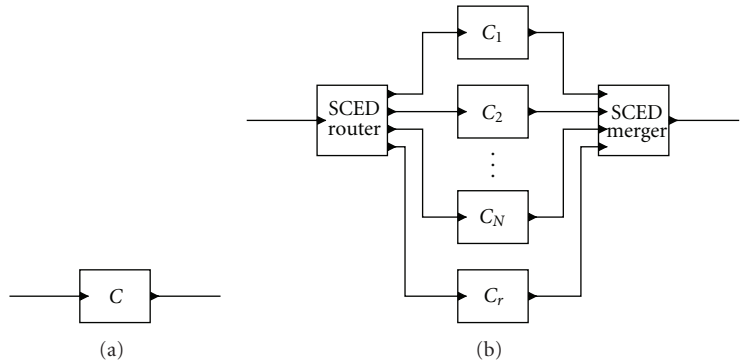


Figure 14: Semiconcurrent error detection at KPN level.

between two possible fault occurrences) Figure 14 shows the components and their interconnections. The *SCED router* component forwards a token from its input port to a different parallel instance just like the *router* component in the parallelization pattern. It also keeps track of an index variable that shows which is the instance currently being checked. Given N parallel instances, every N tokens, one of the instances is checked by sending the same token to the redundant instance. *SCED merger* component reads its input ports in order just like the *merger* component in the parallelization pattern. It also compares the results of C_r and the currently checked instance. If the results are identical, then both components are fault-free. If they are different, then it may be that either of them is faulty. If the results are again different in the next checking round, then the redundant instance is declared faulty. Otherwise it is the parallel instance that has been faulty in the previous checking round. The faulty result can be propagated to the rest of the system depending on the application characteristics. If no fault propagation is allowed, the results will be buffered internally in the SCED merger component until the faulty component is identified correctly. Then the results can be forwarded to the output of the SCED merger by choosing the results of the nonfaulty component.

8. Conclusion and Future Work

In this work, we proposed an active middleware layer to accommodate KPN applications on NoC-based platforms. Besides satisfying KPN semantics, the middleware allows application components to be platform independent with regard to the on-chip communication infrastructure. The middleware is solely based on MPI_send() and MPI_recv() communication primitives, thus it does not require any modification to the NoC platform. The middleware is an initial step towards implementing a self-adaptive run-time environment on the NoC platform. In order to realize a functional simulation of the system proposed, we integrated our SACRE tool with a NoC cycle-accurate simulator, and we evaluated the overhead, in terms of computational time and total data traffic, associated to the use of the middleware in the case of a JPEG case study. The results show that the overhead in both metrics can be lower than 10% depending on the chosen bound of the connectors and the size of tokens being transferred at the application level.

Moreover, we proposed application level fault tolerance patterns, namely, TMR, RFCS, and SCED patterns, that can be applied on a KPN application regardless of tasks being run on programmable or nonprogrammable cores as long as the middleware and the adaptation mechanisms allow us to treat them in the same way while modifying the application graph. The patterns can be applied to a given KPN application graph automatically by an adaptation controller and in cooperation with the self-adaptive run-time environment. Thus they set the application programmer free from having to care about dependability concerns.

Acknowledgments

This work was funded by the European Commission under Project MADNESS (no. FP7-ICT-2009-4-248424). The paper reflects only the authors' view; the European Commission is not liable for any use that may be made of the information contained herein. The authors would like

to thank Mariagiovanna Sami for her valuable comments on this work.

References

[1] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*, Morgan Kaufmann, Burlington, Mass, USA, 2006.

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress (Information Processing '74)*, J. L. Rosenfeld, Ed., pp. 471–475, North-Holland Puplishing Company, New York, NY, USA, 1974.

[3] O. Derin and A. Ferrante, "Simulation of a self-adaptive runtime environment with hardware and software components," in *Proceedings of the ESEC/FSE Workshop on Software Integration and Evolution at Runtime (SINTER '09)*, pp. 37–40, ACM, New York, NY, USA, August 2009.

[4] O. Derin and A. Ferrante, "Enabling self-adaptivity in componentbased streaming applications," *SIGBED Review*, vol. 6, no. 3, special issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES '09), 2009.

[5] O. Derin, A. Ferrante, and A. V. Taddeo, "Coordinated management of hardware and software self-adaptivity," *Journal of Systems Architecture*, vol. 55, no. 3, pp. 170–179, 2009.

[6] "Noxim NoC simulator," http://noxim.sourceforge.net .

[7] E. Carara, A. Mello, and F. Moraes, "Communication models in networks-on-chip," in *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP '07)*, pp. 57–60, IEEE Computer Society, Washington, DC, USA, 2007.

[8] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: the compaan/laura approach," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, p. 10340, Paris, France, February 2004.

[9] A. Nieuwland, J. Kang, O. P. Gangwal et al., "C-heap: a heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, vol. 7, pp. 233–270, 2002.

[10] S. Kwon, Y. Kim, W. C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel-programming framework for MPSoC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 1–39, 2008.

[11] I. Bacivarov, W. Haid, K. Huang, and L. Thiele, "Methods and tools for mapping process networks onto multi-processor systems-on-chip," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., pp. 1007–1040, Springer, 2010.

[12] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, "Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos," in *Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '09)*, pp. 35–44, IEEE, Grenoble, France, 2009.

[13] "Multicore associations communication api.," http://www.multicore-association.org.

[14] "A high performance message passing library," http://www.open-mpi.org/.

[15] G. M. Almeida, G. Sassatelli, P Benoit et al., "An adaptive message passing MPSoC framework," *International Journal of Reconfigurable Computing*, vol. 2009, p. 20, 2009.

[16] A. B. Nejad, K. Goossens, J. Walters, and B. Kienhuis, "Mapping kpn models of streaming applications on a network-on-chip platform," in *Proceedings of the Workshop on Signal Processing, Integrated Systems and Circuits(ProRISC '2009)*, November 2009.

[17] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin, "An asynchronous noc architecture providing low latency service and its multi-level design framework," in *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 54–63, IEEE Computer Society, Washington, DC, USA, 2005.

[18] I. Koren and C. M. Krishna, *Fault Tolerant Systems*, Morgan Kaufmann, San Francisco, Calif, USA, 2007.

[19] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1491–1501, 1985.

[20] M. R. Lyu, *Handbook of Software Reliability Engineering*, McGraw-Hill, Hightstown, NJ, USA, 1996.

[21] C. Fuhrman, S. Chutani, and H. Nussbaumer, "A fault-tolerant implementation using multiple-task triple modular redundancy," in *Proceedings of the IEEE International Workshop on Factory Communication Systems (WFCS '95)*, pp. 75–80, October 1995.

[22] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new approach to software-implemented fault tolerance," *Journal of Electronic Testing*, vol. 20, no. 4, pp. 433–437, 2004.

[23] E. L. Rhod, C. A. Lisbôa, L. Carro, M. Sonza Reorda, and M. Violante, "Hardware and software transparency in the protection of programs against SEUs and SETs," *Journal of Electronic Testing*, vol. 24, no. 1–3, pp. 45–56, 2008.

[24] J. Ceponis, E. Kazanavicius, and A. Mikuckas, "Fault tolerant process networks," *Information Technology and Control*, vol. 35, no. 35, pp. 124–130, 2006.

[25] J. Čeponis, E. Kazanavičius, and L. Čeponiene, "Handling multiple failures in process networks," *Information Technology and Control*, vol. 37, no. 1, pp. 19–25, 2008.

[26] T. M. Parks, *Bounded scheduling of process networks*, Ph.D. thesis, University of California, Berkeley, Calif, USA, December 1995.

[27] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto NoC architectures," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 2, pp. 896–901, February 2004.

[28] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Proceedings of the 7th International Conference on Application of Con-currency to System Design (ACSD '07)*, pp. 29–40, July 2007.

[29] T. T. Ye, L. Benini, and G. De Micheli, "Packetized on-chip interconnect communication analysis for mpsoc," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, vol. 1, p. 10344, IEEE Computer Society, Washington, DC, USA, 2003.

[30] "Sesc simulator," http://sesc.sourceforge.net .

[31] K. Reick, P. N. Sanda, S. Swaney et al., "Fault-tolerant design of the IBM Power6 microprocessor," *IEEE Micro*, vol. 28, no. 2, pp. 30–38, 2008.

[32] D. K. Pradhan and N. H. Vaidya, "Roll-forward checkpointing scheme: a novel fault-tolerant architecture," *IEEE Transactions on Computers*, vol. 43, no. 10, pp. 1163–1174, 1994.

[33] A. Antola, F. Ferrandi, V. Piuri, and M. Sami, "Semiconcurrent error detection in data paths," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 449–465, 2001.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 601986, 22 pages
doi:10.1155/2011/601986

Research Article

Exploring Online Synthesis for CGRAs with Specialized Operator Sets

Stefan Döbrich and Christian Hochberger

Chair for Embedded Systems, Dresden University of Technology, Nöthnitzer Straße 46, 01187 Dresden, Germany

Correspondence should be addressed to Stefan Döbrich, stefan.doebrich@tu-dresden.de

Received 10 August 2010; Revised 16 December 2010; Accepted 10 February 2011

Academic Editor: Michael Hübner

The design of energy-efficient systems has become a major challenge for engineers over the last decade. One way to save energy is to spread out computations in space rather than in time (as traditional processors do). Unfortunately, this requires to design specialized hardware for each application. Also, the nonrecurring expenses for the manufacturing of chips continuously grow. Implementing the computations on FPGAs and CGRAs solves this dilemma, as the non recurring expenses are shared between many different applications. We believe that online synthesis that takes place during the execution of an application is one way to broaden the applicability of reconfigurable architectures as no expert knowledge of synthesis and technologies is required. In this paper, we give a detailed analysis of the amount and specialization of resources in a CGRA that are required to grant a significant speedup of Java bytecode. In fact, we show that even a relatively small number of specialized reconfigurable resources is sufficient to speed up applications considerably. Particularly, we look at the number of dedicated multipliers and dividers. Also, we discuss the required number of concurrent memory access operations inside the CGRA. Again, it shows that two concurrent memory access operations are sufficient for almost all applications.

1. Introduction

Designers of almost all types of systems experience a continuously increased demand for performance and/or higher energy efficiency. Various options are intensely discussed to satisfy this demand.

The currently most often named technology is multicore processors. Using them to gain substantial performance improvements is a rather involved process, and, up till now, it is a technology which is typically mainly found in desktop and server systems. Only now dual or quad core systems are emerging in the area of embedded systems.

Popular technologies like general purpose graphics processors (GPGPU) consume vast amounts of energy and require very specialized programming environments (e.g., OpenCL or CUDA).

In the area of embedded systems, MPSoCs are a valid choice to cast the available transistors into usable computing power. However, existing application code has to be rewritten to distribute the application over the different cores and to synchronize the processing.

A more flexible way to use the available transistors is given by field programmable gate arrays (FPGAs). Here, the user can configure logic resources on a bit level (fine-grained logic) to build individual circuits that accomplish the required behaviour of the application. Unfortunately, this requires expert knowledge and thus is not an option for traditional software developers.

Some FPGA families even allow dynamic partial reconfiguration so that only a part of the implemented circuit is exchanged at runtime. This enables a dynamic adaptation of the implemented circuit to the characteristics of the application. The tool support which is required to handle this design style is enormous, and thus the corresponding configurations are created offline and outside of the target system. The high flexibility to configure circuits at bit level also comes with the big drawback of the large amount of configuration information.

Coarse-grain reconfigurable arrays (CGRAs) as a contrary technology to FPGAs, try to solve this last problem by working on word level instead of bit level. The amount of configuration information is dramatically reduced, and also

the programming of such architectures can be considered more *software style*.

In general, all of the above solutions require a major restructuring and/or rewriting of the application code. Often only a complete new development will reach the full potential of the underlying implementation technology.

The aim of our research is to provide a new processor paradigm the AMIDAR class of processors [1], which makes code optimization or architecture knowledge by developers needless for performance improvement/energy saving.

Certainly, it is clear that knowledge of the underlying architecture or paying attention to best practices can improve the resulting performance even more. But for all that, no special knowledge beyond software development skills is necessary.

The AMIDAR model itself is an abstract processor model which is capable of targeting different platforms—for instance, it may be implemented by full-custom design or as a soft-core targeting FGPAs. According to the actual implementation, the model provides built-in runtime adaptivity at different levels, even if not all technologies are capable of implementing all adaptive operation.

The set of adaptive operations covers reorganization of communication structures, evolution of functional units, and the synthesis of application-specific functional units at runtime. For the sake of readability, Section 3 gives an overview of the AMIDAR model, although similar descriptions have been published in other papers. Furthermore, it provides additional references to further reading about selected aspects of AMIDAR.

In order to prove the capabilities of our model, we have implemented a cycle accurate simulator, which allows the analysis of individual aspects regardless of the underlying technology. In previous papers, we have already discussed the effects of bus-level adaptivity and evolution of functional units [2].

Currently, our research targets the synthesis of new application-specific functional units at runtime. These functional units are mapped to a CGRA which is part of the processor itself. Choosing CGRAs as target platform over fine-grained structures reduces the amount of configuration information considerably.

The most promising target for the synthesis of new functional units are the runtime-intensive kernels of the actual application. In order to determine these code sequences, a continuous runtime profiling of the executed code is done. We have already proposed a hardware circuit that provides capability to handle this profiling and all associated information [3]. Nevertheless, we will give a short introduction to the actual mechanism in Section 4.

The synthesis is triggered in case the profiling mechanism detects a code sequence that consumes more execution time than a given threshold. As the synthesis takes place at runtime of the application, all of our algorithms are designed to consume as little runtime as possible. Hence, we are trading quality of the synthesis results for a better performance and a smaller memory footprint. We have proposed our synthesis algorithms in previous work [4, 5]. Anyhow, Section 5 details the synthesis as the algorithms are fundamental to our whole concept and the contributions of this article.

The main focus of this article lies on the implementation of improved resource constraints in the synthesis algorithms. We evaluated the influence of different CGRA characteristics on the runtime of synthesized functional units. These characteristics cover the actual size of the CGRA regarding the number of its operators, the set of operations implemented by each operator, as well as the effect of dual ported memory access within the CGRA. The evaluation of all benchmark applications is presented in Section 6.

A major insight that we gain from our benchmarks is that almost all application kernels we tested can be mapped to an array of four or at most eight operators. Thus, it is possible to implement several different kernels on a CGRA with sixteen operators in parallel. The more detailed conclusion in Section 7 additionally provides a comparison of AMIDARs performance with an Intel Pentium Core2 Duo processor.

We are aiming at a further performance and quality improvement of both, the synthesis algorithms as well as the generated functional units. Further details on the research work we are planning to accomplish in the future is given in Section 8.

2. Related Work

Fine grain reconfigurable logic for application improvement has been used for more than two decades. Early examples are the CEPRA-1X which was developed to speed up cellular automata simulations. It gained a speedup of more than 1000 compared with state-of-the-art workstations [6]. This level of speedup still persists for many application areas, for example, the BLAST algorithm [7]. Unfortunately, these speedups require highly specialized HW architectures and domain-specific modelling languages.

Combining FPGAs with processor cores seems to be a natural idea. Compute-intense parts can be realized in the FPGA, and the control intense parts can be implemented in the CPU. GARP was one of the first approaches following this scheme [8]. It was accompanied by the synthesizing C compiler NIMBLE [9] that automatically partitions and maps the application.

Static transformation from high-level languages like C into fine grain reconfigurable logic is still the research focus of a number of academic and commercial research groups. Only very few of them support the full programming language [10, 11].

Also, Java as a base language for mapping has been investigated in the past. Customized accelerators to speed up the execution of Java bytecode have been developed [12]. In this case, only a small part of the bytecode execution is implemented in hardware and the main execution is done on a conventional processor. Thus, the effect was very limited.

CGRAs have also been used to speed up applications. They typically depend on compile time analysis and generate a single datapath configuration for an application beforehand: RaPiD [13], PipeRench [14], Kress-Arrays [15], or the PACT-XPP [16]. In most cases, specialized tool sets

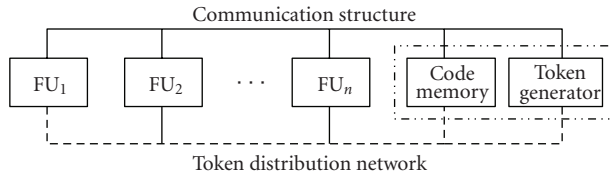


Figure 1: Abstract model of an AMIDAR processor.

and special-purpose design languages had to be employed to gain substantial speedups. Whenever general purpose languages could be used to program these architectures, the programmer had to restrict himself to a subset of the language and the speedup was very limited.

Efficient static transformation from high-level languages into CGRAs is also investigated by several groups. The DRESC [17] tool chain targeting the ADRES [18, 19] architecture is one of the most advanced tools. Yet, it requires hand-written annotations to the source code, and in some cases even some hand-crafted rewriting of the source code. Also, the compilation times easily get into the range of days.

The RISPP architecture [20] lies between static and dynamic approaches. Here, a set of candidate instructions are evaluated at compile time. These candidates are implemented dynamically at runtime by varying sets of so-called atoms. Thus, alternative design points are chosen depending on the actual execution characteristics.

Dynamic transformation from software to hardware has been investigated already by other researchers. Warp processors dynamically transform assembly instruction sequences into fine grain reconfigurable logic [21]. This happens by synthesis of bitstreams for the targeted WARP-FPGA platform. Furthermore, dynamic synthesis of Java bytecode has been evaluated [22]. Nonetheless, this approach is only capable of synthesizing combinational hardware.

The token distribution principle of AMIDAR processors has some similarities with transport triggered architectures [23]. Yet, in TTAs an application is transformed directly into a set of tokens. This leads to a very high memory overhead and makes an analysis of the executed code extremely difficult.

3. The AMIDAR Processing Model

In this section, we will give an overview of the AMIDAR processor model. We describe the basic principles of operation. This includes the architecture of an AMIDAR processor in general, as well as specifics of its components. Furthermore, we discuss the applicability of the AMIDAR model to different instruction sets. Afterwards, an overview of a minimum implementation of an AMIDAR-based Java machine is given. Finally, we discuss several mechanisms of the model that allow the processor to adopt to the requirements of a given application at runtime.

3.1. Overview. An AMIDAR processor consists of three main parts. A set of functional units, a token network, and a communication structure.

Two functional units, which are common to all AMIDAR implementations, are the code memory and the token generator. As its name tells, the code memory holds the applications code. The token generator controls the other components of the processor by means of tokens. Therefore, it translates each instruction into a set of tokens, which is distributed to the functional units over the token distribution network. The tokens tell the functional units what to do with input data and where to send the results. Specific AMIDAR implementations may allow the combination of the code memory and the token generator as a single functional unit. This would allow the utilization of several additional side effects, such as instruction folding. Functional units can have a very wide range of meanings: ALUs, register files, data memory, specialized address calculation units, and so forth. Data is passed between the functional units over the communication structure. This data can have various meanings: program information (instructions), address information, or application data. Figure 1 sketches the abstract structure of an AMIDAR processor.

3.2. Principle of Operation. Execution of instructions in AMIDAR processors differs from other execution schemes. Neither microprogramming nor explicit pipelining are used to execute instructions. Instead, instructions are broken down to a set of tokens which are distributed to a set of functional units. These tokens are 5-tuples, where a token is defined as $T = \{UID, OP, TAG, DP, INC\}$. It carries the information about the type of operation (OP) that will be executed by the functional unit with the specified id (UID). Furthermore, the version information of the input data (TAG) that will be processed and the destination port of the result (DP) are part of the token. Finally, every token contains a tag increment flag (INC). By default, the result of an operation is tagged equally to the input data. In case the TAG-flag is set, the output tag is increased by one.

The token generator can be built such that every functional unit which will receive a token is able to receive it in one clock cycle. A functional unit begins the execution of a specific token as soon as the data ports receive the data with the corresponding tag. Tokens which do not require input data can be executed immediately. Once the appropriately tagged data is available, the operation starts. Upon completion of an operation, the result is sent to the destination port that was denoted in the token. An instruction is completed, when all the corresponding tokens are executed. To keep the processor executing instructions, one of the tokens must be responsible for sending a new instruction to the token generator.

Table 1: Runtime of benchmarks on AMIDAR based Java processor and x86 Intel Core 2 Duo @ 2.66 GHz.

(a) Round key generation of encryption of cryptographic cipher benchmarks

Processor	Rijndael clock ticks	Twofish clock ticks	RC6 clock ticks	Serpent clock ticks
x86	≈8700	≈200000	≈28000	≈21000
AMIDAR	17760	525276	61723	44276

(b) Single block encryption of cryptographic cipher benchmarks

Processor	Rijndael clock ticks	Twofish clock ticks	RC6 clock ticks	Serpent clock ticks
x86	≈8800	≈4000	≈5600	≈14000
AMIDAR	21389	12864	17371	34855

(c) Hash algorithms and message digests

Processor	SHA-1 clock ticks	SHA-256 clock ticks	MD5 clock ticks
x86	≈9100	≈17200	≈4700
AMIDAR	23948	47471	11986

(d) Filter applications

Processor	Sobel filter clock ticks	Grayscale filter clock ticks	Contrast filter clock ticks
x86	≈7900	≈200	≈370
AMIDAR	21124	236	608

(e) JPEG encoding and its application kernels

Processor	JPEG encoder clock ticks	Color space transformation clock ticks	2-D DCT clock ticks	Quantization clock ticks
x86	≈17300000	≈2700000	≈11200	≈4800
AMIDAR	17368663	3436078	23054	7454

A more detailed explanation of the model, its application to Java bytecode execution, and its specific features can be found in [1, 24].

3.3. Applicability. In general, the presented model can be applied to any kind of instruction set. Therefore, a composition of microinstructions has to be defined for each instruction. Overlapping execution of instructions comes automatically with this model. Thus, it can best be applied if dependencies between consecutive instructions are minimal. The model does not produce good results, if there is a strict order of those microinstructions, since in this case no parallel execution of microinstructions can occur. The great advantage of this model is that the execution of an instruction depends on the token sequence, and not on the timing of the functional units. Thus, functional units can be replaced at runtime with other versions of different characterizations. The same holds for the communication structure, which can be adapted to the requirements of the running application. Thus, this model allows us to optimize global goals like performance or energy consumption. Intermediate virtual assembly languages like Java bytecode, LLVM bitcode, or the .NET common intermediate language are good candidates for instruction sets. The range of functional unit implementations and communication structures is especially wide, if the instruction set has a very high abstraction level and/or basic operations are sufficiently complex. Finally, the data-driven approach makes it possible to easily integrate new functional units and create new instructions to use these functional units.

3.4. Implementation of an AMIDAR Based Java Processor. The structure of an example implementation of an AMIDAR based Java processor is displayed in Figure 2. This section will give a brief description of the processors structure and the functionality of its contained functional units. The central units of the processor are the code memory and the token generator. In case of a Java processor, the code memory holds all class files and interfaces, as well as their corresponding constant pools and attributes. The Java runtime model separates local variables and the operand stack from each other. Thus, a functional unit that provides the functionality of a stack memory represents the operand stack. Furthermore, an additional functional unit holds all local variables.

A local variable may be of three different types. It may be an array reference type or an object reference type, and

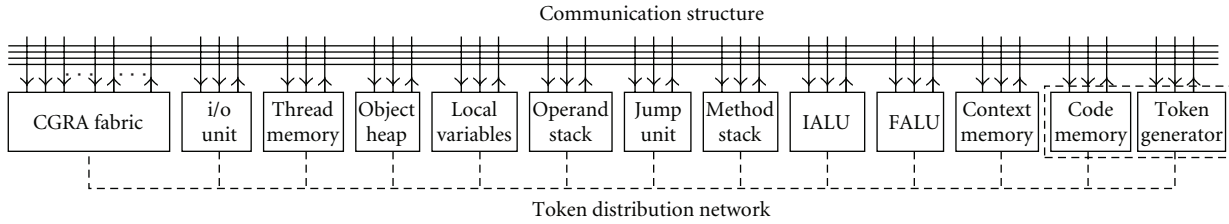


Figure 2: Model of a Java (non)Virtual Machine on AMIDAR basis.

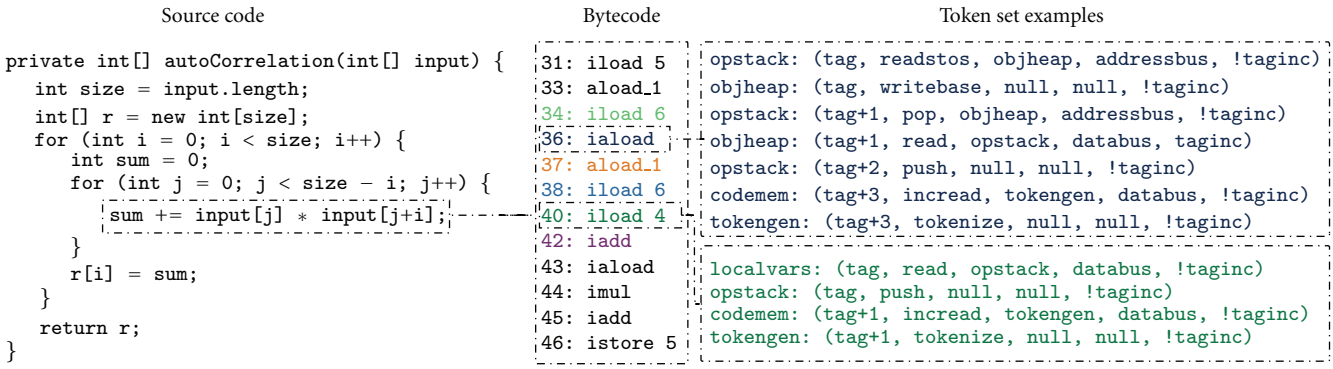


Figure 3: Example source code sequence and the resulting bytecode and exemplified token sequences.

furthermore, it may represent a native data type such as `int` or `float`. All native data types are stored directly in the local variable memory while all reference types point to an object or array located on the heap memory. Thus, the processor contains another memory unit incorporating the so called object heap. Additionally, the processor contains a method stack. This memory is used to store information about the current program counter and stack frame in case of a method invocation. The context of currently not running threads is stored in the context memory.

In order to process arithmetic operations, the processor will contain at least one ALU functional unit. Nonetheless, it is possible to separate integer and floating point operations into two disjoint functional units, which improves the throughput. Furthermore, the processor contains a jump unit which processes all conditional jumps. Therefore, the condition is evaluated, and the resulting jump offset is transferred to the code memory.

Instructions and data are distributed over the communication network. In the presented case, this structure consists of four equal busses of 32 bit width. The busses are assigned to the functional units via round robbin.

3.5. Example Token Sequence and Execution Trace. In order to give a more detailed picture of an actual applications execution on an AMIDAR processor, we have chosen an autocorrelation function as an example. The source code of the autocorrelation function, its resulting bytecode, and sample token sequences for two of its bytecodes are displayed in Figure 3. The `iaload` instruction at program counter 36 is focussed on the further descriptions.

The `iaload` bytecode loads an integer value from an array at the heap and pushes it onto the operand stack. Initially, the array's address on the heap and the offset of the actual value are positioned at the top of the stack. Firstly, the array's address is read from the second position of the stack and is sent to the heap where it is written to the base address register. Afterwards, the actual offset is popped of the stack and sent to the heap, and is used as address for a read operation. The read value is sent back to the operand stack and pushed on top of the stack.

Figure 4 shows an excerpt of the execution of the autocorrelation function. Each line of the diagram represents the internal state of the displayed functional units in the corresponding clock cycle. Furthermore, all operations that belong to the same instruction are colored identically, which visualizes the overlapping execution of instructions.

3.6. Adaptivity in the AMIDAR Model. The AMIDAR model exposes different types of adaptivity. All adaptive operations covered by the model are intended to dynamically respond to the running applications behavior. Therefore, we identified adaptive operations that adopt the communication structure to the actual interaction scheme between functional units. Furthermore, a functional unit may be the bottleneck of the processor. Hence, we included similar adaptive operations for functional units. The following subsections will give an overview of the adaptive operations provided by the AMIDAR model. Most of the currently available reconfigurable devices do not fully support the described adaptive operations (e.g., addition or removal of bus structures). Yet, the model itself contains these possibilities, and so may benefit from future hardware designs.

3.7. Adaptive Communication Structures. The bus conflicts that occur during data transports can be minimized

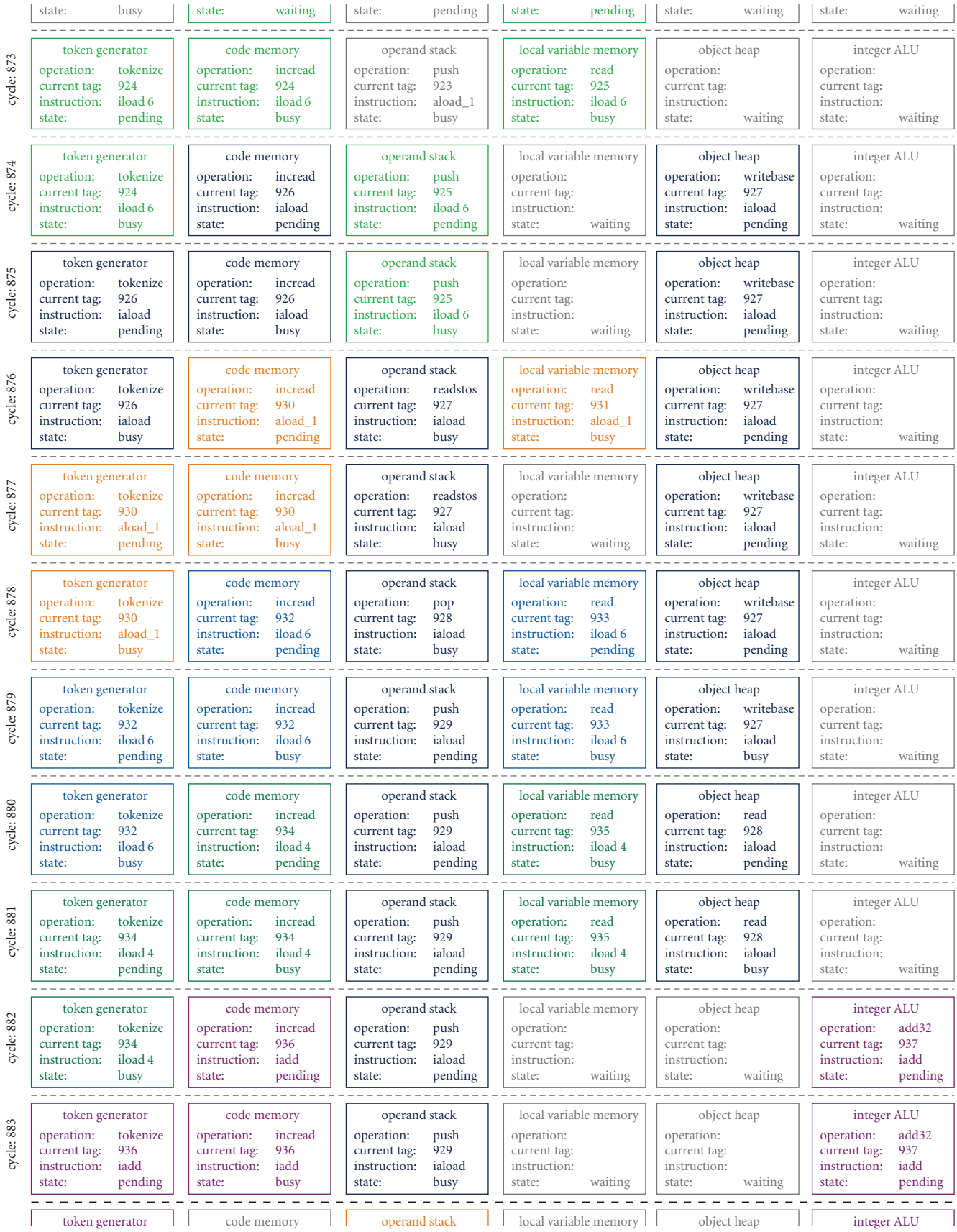


Figure 4: Visualized excerpt of an exeuction trace of the autocorrelation example.

Table 2: Runtime acceleration of benchmark applications.

(a) Round key generation

Configuration	Rijndael		Twofish		RC6		Serpent	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	17760	—	525276	—	61723	—	44276	—
4 operators	4602	3.86	43224	12.15	3725	16.57	6335	6.99
8 operators	4284	4.15	35130	14.95	3459	17.84	6245	7.09
12 operators	4337	4.09	34280	15.32	3459	17.84	6230	7.11
16 operators	4337	4.09	34112	15.40	3459	17.84	6230	7.11

(b) Single block encryption

Configuration	Rijndael		Twofish		RC6		Serpent	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	21389	—	12864	—	17371	—	34855	—
4 operators	6230	3.43	8506	1.51	2852	6.09	3278	10.63
8 operators	6181	3.46	8452	1.52	2810	6.18	3273	10.65
12 operators	6167	3.47	8452	1.52	2768	6.28	3273	10.65
16 operators	6167	3.47	8452	1.52	2768	6.28	3273	10.65

(c) Hash & digest algorithms

Configuration	SHA-1		SHA-256		MD5	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	23948	—	47471	—	11986	—
4 operators	4561	5.25	3619	13.12	1485	8.07
8 operators	4561	5.25	3484	13.63	1485	8.07
12 operators	4561	5.25	3484	13.63	1485	8.07
16 operators	4561	5.25	3484	13.63	1485	8.07

(d) Image Processing

Configuration	Sobel filter		Grayscale filter		Contrast filter	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	21124	—	236	—	608	—
4 operators	2246	9.41	59	4.00	90	6.76
8 operators	2246	9.41	59	4.00	90	6.76
12 operators	2246	9.41	59	4.00	90	6.76
16 operators	2246	9.41	59	4.00	90	6.76

(e) JPEG encoding

Configuration	JPEGEncoder		Color Space Transformation		2-D Forward DCT		Quantization	
	Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup	Clock Ticks	Speedup
Plain software	17368663	—	3436078	—	23054	—	7454	—
4 operators	4737944	3.67	323805	10.61	2743	8.40	1816	4.10
8 operators	4645468	3.74	292889	11.73	2572	8.96	1816	4.10
12 operators	4620290	3.76	277431	12.39	2545	9.06	1816	4.10
16 operators	4612561	3.77	269702	12.74	2545	9.06	1816	4.10

by adapting the communication structure. Therefore, we designed a set of several adaptive operations that may be applied to it. In [2], we have shown how to identify the conflicting bus taps and we have also shown a heuristic to modify the bus structure to minimize the conflicts.

In order to exchange data between two functional units, both units have to be connected to the same bus structure. Thus, it is possible to connect a functional unit to a bus in case it will send data to/receive data from another functional unit. This may happen if the two functional units do not have a connection yet. Furthermore, the two units may have an interconnection, but the bus arbiter assigned the related bus structure to another sending functional unit. In this case, a new interconnection could be created as well.

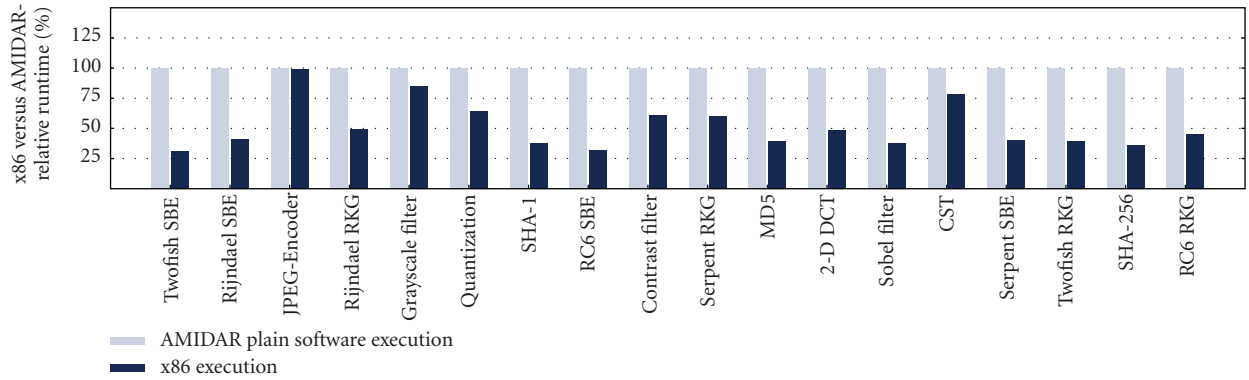


Figure 5: Runtime comparison of AMIDAR based Java processor and x86 Intel Core 2 Duo @ 2.66GHz.

As functional units may be connected to a bus structure, they may also be disconnected. For example, this may happen in case many arbitration collisions occur on a specific bus. As a result, one connection may be transfered to another bus structure by disconnecting the participants from one bus, and connecting them to a bus structure with sparse capacity.

In case the whole communication structure is heavily utilized and many arbitration collisions occur, it is possible to split a bus structure. Therefore, a new bus structure is added to the processor. One of the connections participating in many collisions is migrated to the new bus. This reduces collisions and improves the applications' runtime and the processors' throughput. Vice versa, it is possible to fold two bus structures in case they are used rarely. As a special case, a bus may be removed completely from the processor. This operation has a lower complexity than the folding operation, and thus may be used in special cases.

All of the described adaptive bus operations have been evaluated at topology level in the already mentioned paper. A hardware technique which allows the actual execution of these operations has not yet been part of our research.

3.8. Adaptive Functional Units. In addition to the adaptive operations regarding the communication structure, there are three different categories of adaptive operations that may be applied to functional units.

Firstly, variations of a specific functional unit may be available. This means, for example, that optimized versions regarding chip size, latency and throughput are available for a functional unit. The most appropriate implementation is chosen dynamically at runtime and may change throughout the lifetime of the application. The AMIDAR model allows the processor to adopt to the actual workload by substitution of two versions of a functional unit at runtime. In [1], we have shown that the characteristics of the functional units can be changed to optimally suit the needs of the running application.

Secondly, the number of instances of a specific functional unit may be increased or decreased dynamically. In case a functional unit is heavily utilized, but cannot be replaced by a specialized version with a higher throughput or shorter latency, it may be duplicated. The distribution of tokens has to be adapted to this new situation, as the token generator has to balance the workload between identical functional units. In contrary to the preceding and succeeding technique, this one has not been evaluated yet. Although the model itself offers this type of adaptivity, it should be noted that we do not further investigate it in this contribution.

Finally, dynamically synthesized functional units may be added to the processors' datapath. It is possible to identify heavily utilized instruction sequences of an application at runtime. A large share of applications for embedded systems rely on runtime-intensive computation kernels. These kernels are typically wrapped by loop structures and iterate over a given array or stream of input data. Both cases are mostly identical, as every stream can be wrapped by a buffer, which leads back to the handling of arrays by the computation itself. In [3], we have shown a hardware circuit that is capable of profiling an applications loop structures at runtime. The profiles gained by this circuit can be used to identify candidate sequences for online synthesis of functional units. These functional units would replace the software execution of the related code.

3.9. Synthesizing Functional Units in AMIDAR. AMIDAR processors need to include some reconfigurable fabric in order to allow the dynamic synthesis and inclusion of functional units. Since fine-grained logic (like FPGAs) requires a large amount of configuration data to be computed and also since the fine grain structure is neither required nor helpful for the implementation of most code sequences, we focus on CGRAs for the inclusion into AMIDAR processors. Successfully employing CGRAs in reconfigurable computing is shown in [25].

The model includes many features to support the integration of newly synthesized functional units into the running application. It allows bulk data transfers from and to data memories, it allows the token generator to synchronize with functional unit operations that take multiple clock cycles, and, finally, it allows synthesized functional units to inject tokens in order to influence the data transport required for the computation of a code sequence.

3.10. Latency of Runtime Adaptivity. Currently, we cannot fully determine the latencies regarding the runtime behavior of the adaptive features of the AMIDAR model. The feature

which is currently examined in our studies is the runtime synthesis of new functional units. Right now, the synthesis process itself is not executed as a separate Java thread within our processor, but only as part of the running simulator. Thus, the process of creating new functional units is transparent to the processor. Hence, a runtime prediction is not possible yet. It should be mentioned that the code currently used for synthesis could be run on the target processor as it is written in Java.

Nonetheless, the usefulness of synthesizing new functional units can be determined in two ways. In case there is no spare time concurrently to the executed task, the runtime of the synthesis process for new functional units slows down the current operation, but after finishing the synthesis, the functional units execute much faster. Thus, eventually, the runtime lost to the synthesis process will be gained back. In case there is enough spare time, the synthesis process did not slow down the application any way and there are no objections against this type of adaptation.

3.11. AMIDAR Performance Evaluation. We compared the AMIDAR based Java processor to an Intel Core 2 Duo in order to obtain an impression of its runtime performance. Therefore, we compiled a set of benchmarks to native code. The benchmarks that were used are described in detail in Section 6.1. The runtime of the benchmarks is displayed in Figure 5 and Table 1. Figure 5 depicts the relation between the two different benchmarks for all applications. The AMIDAR execution is used as baseline while the runtime of the x86 execution is displayed proportionately.

It can be seen, that the runtime on the basic AMIDAR processor is is up to three-times higher. Furthermore, the JPEG Encoder benchmarks as a whole application does not fall behind x86 execution, because of its high amount of memory accesses. Overall, it can be said that the execution of a program on an AMIDAR processor takes twice the time as the execution on an x86 processor. Standard interpreter Java Virtual Machines do not achieve such a good relative performance compared to natively compiled code.

4. Runtime Application Profiling

A major task in synthesizing hardware functional units for AMIDAR processors is runtime application profiling. This allows the identification of candidate instruction sequences for hardware acceleration. Plausible candidates are the runtime critical parts of the current application.

In previous work [3], we have shown a profiling algorithm and corresponding hardware implementation which generates detailed information about every executed loop structure. Those profiles contain the total number of executed instructions inside the affected loop, the loops start program counter, its end program counter, and the total number of executions of this loop. The profiling circuitry is also capable to profile nested loops, not only simple ones.

Profiling is based on the fact that the last instruction of loops is always branch with negative offset in Java bytecode. Also, negative branch offsets are only used for this purpose and do not occurat other places of the code. The value of an instructions counter is added to an associated loop register (one for each loop or loop nesting level). These loop registers are realized by a fully associative memory. The size of this memory depends on the maximum number of loops and loop nesting levels in a method. It is usually very small for real live applications (<16). The assiciative memory has to be saved during method calls and returns. This requires typically less time than the housekeeping of the method call itself. Thus, profiling does not introduce any runtime overhead.

A profiled loop structure becomes a synthesis candidate in case its number of executed instructions surmounts a given threshold. The size of this threshold can be configured dynamically for each application.

Furthermore, an instruction sequence has to match specific constraints in order to be synthesized. Currently, we are not capable of synthesizing code sequences containing the following instruction types, as our synthesis algorithm has not evolved to this point yet:

(i) memory allocation operations,

(ii) exception handling,

(iii) thread synchronization,

(iv) some special instructions, for example `lookupswitch`,

(v) access operations to multidimensional arrays,

(vi) method invocation operations.

From this group, only access to multidimensional arrays and method invocations are important from a performance aspect.

Multidimensional arrays do actually occur in compute kernels. Access operations on these arrays are possible in principle in the AMIDAR model. Yet, multidimensional arrays are organized as arrays of arrays in Java. Thus, access operations need to be broken down into a set of stages (one for each dimension), which is not yet supported by our synthesis algorithm. Nevertheless, a manual rewrite of the code is possible to map multidimensional arrays to one dimension. Reorganizing memory access patterns during the synthesis process could certainly improve the performance here, but the required dependency analysis is far too complex to be carried out online.

Similarly, method inlining can be used to enable the synthesis of code sequences that contain method invocations. Techniques for the method inlining are known from JIT compilers that preserve the polymorphism of the called method. Yet, these techniques require the abortion of the execution of the HW under some conditions, which is not yet supported by our synthesis algorithm.

5. Online Synthesis of Application-Specific Functional Units

The captured data of the profiling unit is evaluated periodically. In case an instruction sequence exceeds the given

runtime threshold the synthesis is triggered, and runs as a low-priority process concurrently to the application. Thus, it only occurs if spare computing time remains in the system, and also cannot interfere with the running application.

5.1. Synthesis Algorithm. An overview of the synthesis steps is given in Figure 6. The parts of the figure drawn in grey are not yet implemented.

Firstly, an instruction graph of the given sequence is created. In this graph, every instruction is represented by a node. The predecessor/successor relations are represented by the graphs edges. In case an unsupported instruction is detected the synthesis is aborted. Furthermore, a marker of a previously synthesized functional unit may be found. If this is the case, it is necessary to restore the original instruction information and then proceed with the synthesis. This may happen if an inner loop has been mapped to hardware before, and then the wrapping loop will be synthesized as well.

Afterwards, all nodes of the graph are scanned for their number of predecessors. In case a node has more than one predecessor, it is necessary to introduce specific Φ-nodes to the graph. These structures occur at the entry of loops or in typical if-else structures. Furthermore, the graph is annotated with branching information. This will allow the identification of the actually executed branch and the selection of the valid data when merging two or more branches by multiplexers. For if-else structures, this approach reflects a speculative execution of the alternative branches. The condition of the if-statement is used to control the selection of one set of result values. Loop entry points are treated differently, as no overlapping or software pipelining of loop kernels is employed.

In the next step, the graph is annotated with a virtual stack. This stack does not contain specific data, but contians the information about the producing instruction that would have created it. This allows the designation of connection structures between the different instructions as the predecessor of an instruction may not be the producer of its input.

Afterwards, an analysis of access operations to local variables, arrays, and objects takes place. This aims at loading data into the functional unit and storing it back to its appropriate memory after its execution. Therefore, a list of data that has to be loaded and a list of data that has to be stored are created.

The next step transforms the instruction graph into a hardware circuit. This representation fits precisely into our simulation. All arithmetic or logic operations are transformed into their abstract hardware equivalent. The introduced Φ-nodes are transfered to multiplexer structures. The annotated branching information helps to connect the different branches correctly and to determine the appropriate control signal. Furthermore, registers and memory structures are introduced. Registers hold values at the beginning and the end of branches in order to synchronize different branches. Localization of memory accesses is an important measure to improve the performance of potential applications. In general, SFUs could also access the heap to read or write array elements, but this access would incur an overhead of several clocks. The memory structures are connected to the consumer/producer components of their corresponding arrays or objects. A datapath equivalent to the instruction sequence is the result of this step.

Execution of consecutive loop kernels is strictly separated. Thus, all variables and object fields altered in the loop kernel are stored in registers at the beginning of each loop iteration.

Arrays and objects may be accessed from different branches that are executed in parallel. Thus, it is necessary to synchronize access to the affected memory regions. Furthermore, only valid results may be stored into arrays or objects. This is realized by special enable signals for all write operations. The access synchronization is realized through a controller synthesis. This step takes the created datapath and all information about timing and dependency of array and object access operations as input. The synthesis algorithm has a generic interface which allows to work with different scheduling algorithms. Currently, we have implemented a modified ASAP scheduling which can handle resource constraints, and additionally we implemented list scheduling. The result of this step is a finite state machine (FSM) which controls the datapath and synchronizes all array and object access operations. Also, the FSM takes care of the appropriate execution of simple and nested loops.

As mentioned above, we do not have a full hardware implementation yet. Thus, placement and routing for the CGRA are not required. We use a cycle accurate simulation of the abstract datapath created in the previous steps.

In case the synthesis has been successful, the new functional unit needs to be integrated into the processor. If marker instructions of previously synthesized FUs were found, the original instruction sequence has to be restored. Furthermore, the affected SFUs have to be unregistered from the processor, and the hardware used by them has to be released.

The synthesis process is depicted in Figure 7. It shows the initial bytecode sequence, the resulting instruction graph, as well as data dependencies between the instructions and the final configuration of the reconfigurable fabric. The autocorrelation function achieves a speedup of 12.42 on an array with four operators and an input vector of 32 integer values.

5.2. Functional Unit Integration. The integration of the synthesized functional unit (SFU) into the running application consists of three major steps: (1) a token set has to be generated which allows the token generator to use the SFU. (2) the SFU has to be integrated into the existing circuit, and (3) the synthesized code sequence has to be patched in order to access the SFU.

The token set consists of three parts: (1) the tokens that transport input data to the SFU, these tokens are sent to the appropriate data sources (e.g., object heap), (2) the tokens that control the operation of the SFU, that is, that start the operation (which happens once the input data is available) and emit the results, and (3) the token set that stores the results of the SFU operation in the corresponding memory.

In a next step, it is necessary to make the SFU accessible to the other processor components. This requires to register

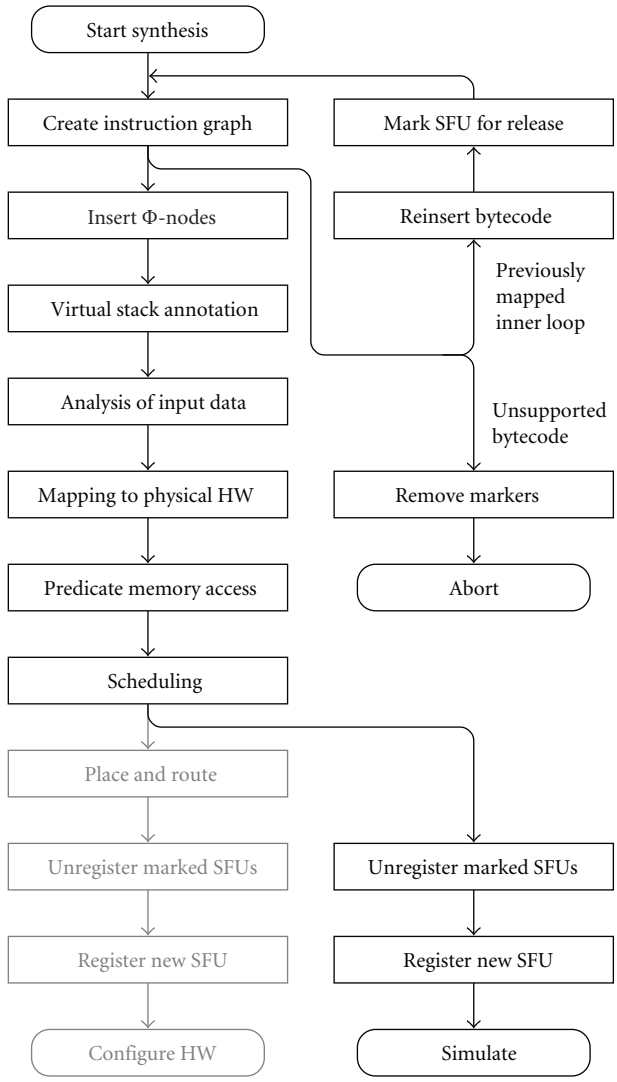


Figure 6: Overview of synthesis steps.

it in the bus arbiter and to update the token generator with the computed token sets. The token set will be triggered by a reserved bytecode instruction.

Finally, the original bytecode sequence has to be replaced by the reserved bytecode instruction. To allow multiple SFUs to coexist, the reserved bytecode carries the ID of the targeted SFU. Patching of the bytecode sequence is done in such a way that the token generator can continue the execution at the first instruction after the transformed bytecode sequence. Also, it must be possible to restore the original sequence in case an embracing loop nesting level will be synthesized.

Now, the sequence is not processed in software anymore but by a hardware SFU. Thus, it is necessary to adjust the profiling data of the affected code sequence.

In [26], we have given further information and a more detailed description of the integration process.

6. Evaluation

In previous research [27], we have evaluated the potential speedup of a simplistic online synthesis with unlimited resources. This is an unrealistic assumption. Thus, we are targeting an architecture based on a CGRA with a limited number of processing elements, and a single shared memory for all arrays and objects [5]. The scheduling of all operations is calculated by longest path list scheduling. The following dataset shows the characteristics of every benchmark and the influence of online synthesis at an applications runtime behavior:

(i) its runtime, and therewith the gained speedup,

(ii) the number of states of the controlling state machine,

(iii) the number of different contexts regarding the CGRA,

(iv) the number of complex operations within those contexts.

The reference value for all measurements is the plain software execution of the benchmarks without synthesized functional units. Note: the mean execution time of a bytecode in our processor is ≈4 clock cycles. This is in the same order as JIT-compiled code on IA32 machines.

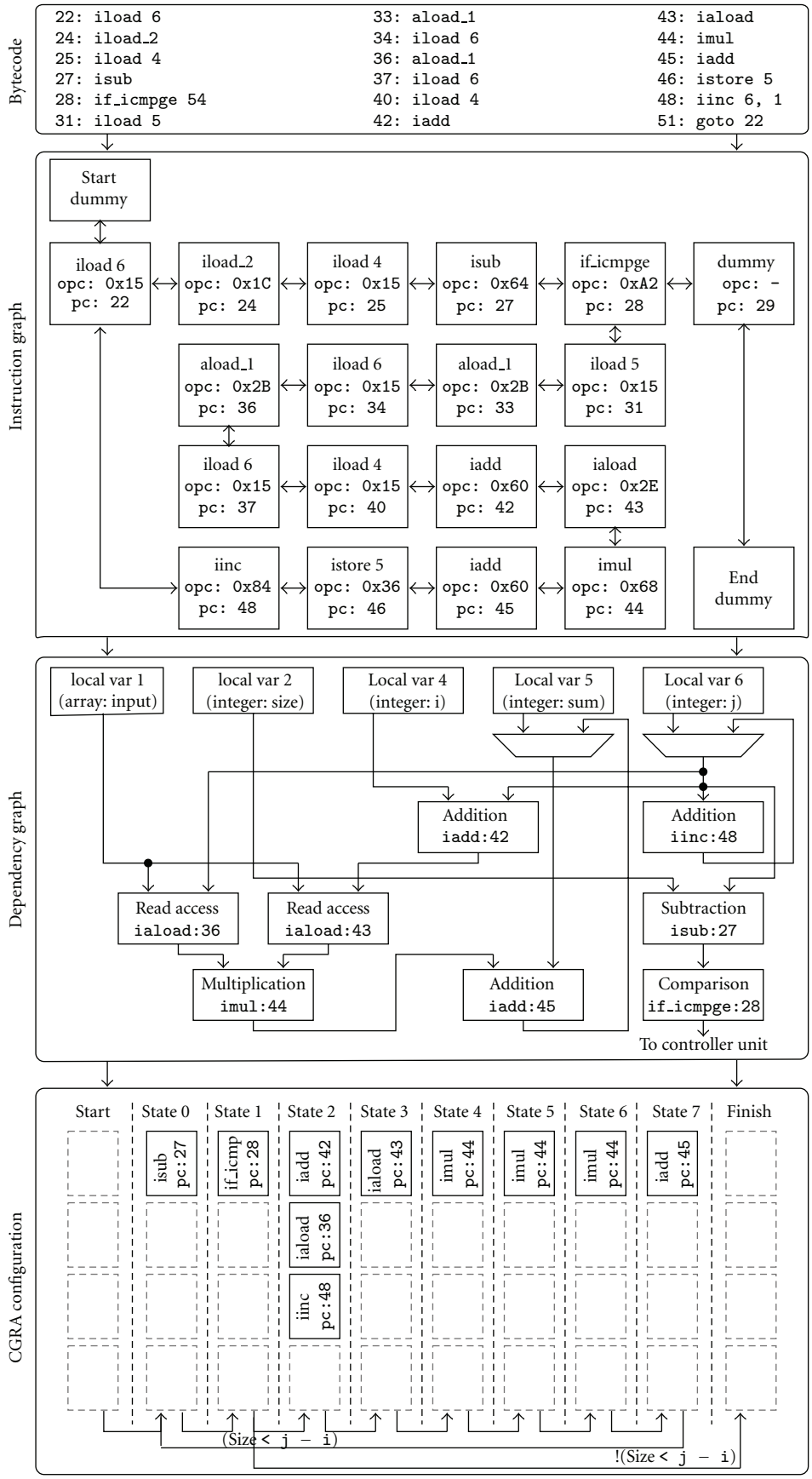


Figure 7: Example of intermediate synthesis results.

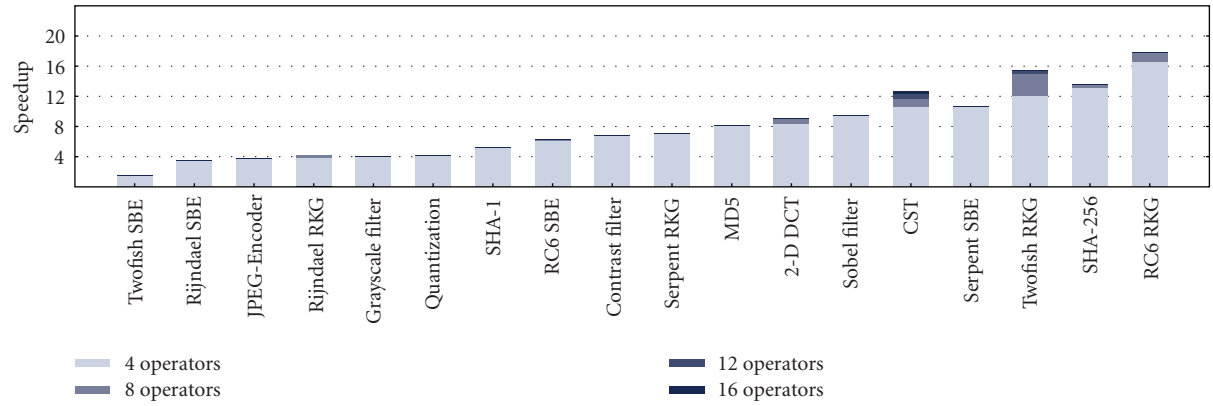


Figure 8: Diagram of runtime acceleration of benchmark applications.

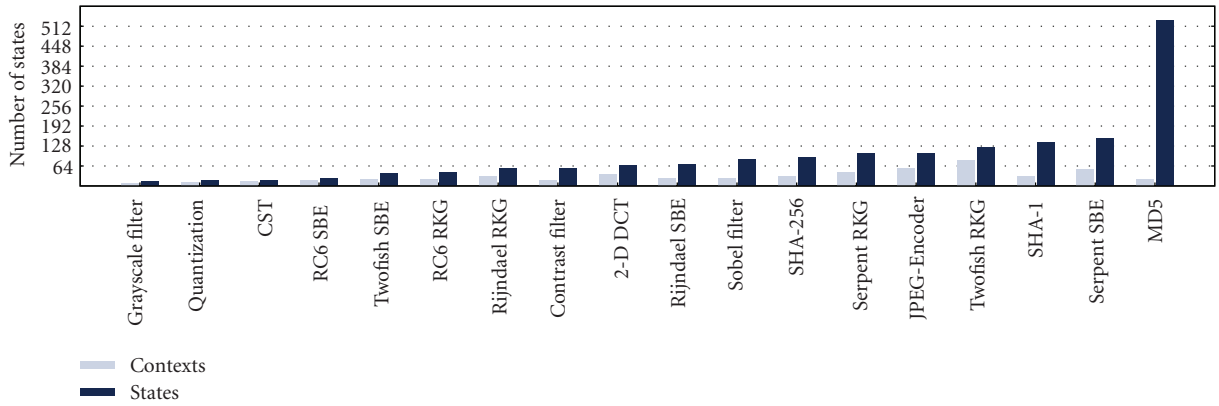


Figure 9: Diagram of complexity of the schedules of benchmark applications.

6.1. Benchmark Applications. We chose applications of four different domains to test our synthesis algorithm. Firstly, we benchmarked several cryptographic ciphers as the importance of security in embedded systems increases steadily. Additionally, we chose hash algorithms and message digests as a second group of appropriate applications, and furthermore evaluated the runtime behavior of image processing kernels. All of these benchmark applications are pure computation kernels. Regularly, they are part of a surrounding application. Thus, we selected the encoding of a bitmap image into a JPEG image as a benchmark application. This application contains several computation kernels, such as color space transformation, 2-D forward DCT, and quantization. Nonetheless, it also contains a substantial amount of code that utilizes those kernels, in order to encode a whole image.

The group of cryptographic cipher benchmarks contains the four block ciphers Rijndael, Twofish, Serpent, and RC6, which all were part of the Advanced Encryption Standard (AES) evaluation process.

We analyzed the runtime behavior of the round key generation out of a 256 bit master key, as this is the largest common key length of those ciphers. Furthermore, we reviewed the encryption of a 16 byte data block, which is the standard block size for all of them. We did not examine the decryption of data, as it is basically an inverted implementation of the encryption. Thus, its runtime behavior is mostly identical.

Another typical group of algorithms used in the security domain are hash algorithms and message digests. We chose the Message Digest 5 (MD5) and two versions of the Secure Hash Algorithm (SHA-1 and SHA-256) as representatives. For instance, these digests are heavily utilized during TLS/SSL-encrypted communication. We measured the processing of sixteen 32-bit words, which is the standard input size for those three algorithms.

Thirdly, we rated the effects of our synthesis algorithm onto image processing kernels. Therefore, we selected a discrete differentiation that uses the Sobel convolution operator as one of those tests. This filter is used for edge detection in images. Furthermore, a grayscale filter and a contrast filter have been evaluated. As its name tells, the grayscale filter transforms a colored image into a grayscale image. The contrast filter changes the contrast of an image regarding given parameters for contrast and brightness.

These three filters operate on a dedicated pixel of an image, or on a pixel and its neighbours. Thus, we measured the appliance of every filter onto a single pixel.

Finally, as we mentioned before, we encoded a given bitmap image into a JPEG image. The computation kernels of this application are the color space transformation, 2-D forward DCT, and quantization. We did not downsample the chroma parts of the image. The input image we have chosen has a size of 160×48 pixels, which results in 20×6 basic blocks of 8×8 pixels. Thus, every of the mentioned processing steps had been executed 120 times for each of

Table 3: Complexity of the schedules of benchmark applications.

(a) Round key generation

Configuration	Rijndael		Twofish		RC6		Serpent	
	States	Contexts	States	Contexts	States	Contexts	States	Contexts
4 operators	57	42	230	110	48	21	124	37
8 operators	55	31	148	113	44	20	106	43
12 operators	55	31	130	91	44	20	103	42
16 operators	55	31	122	83	44	20	103	42

(b) Single block encryption

Configuration	Rijndael		Twofish		RC6		Serpent	
	States	Contexts	States	Contexts	States	Contexts	States	Contexts
4 operators	78	37	46	33	25	17	153	54
8 operators	71	31	40	26	23	20	152	54
12 operators	69	26	40	22	23	19	152	54
16 operators	69	23	40	20	23	19	152	54

(c) Hash & digest algorithms

Configuration	SHA-1		SHA-256		MD5	
	States	Contexts	States	Contexts	States	Contexts
4 operators	138	29	107	28	531	20
8 operators	138	29	92	31	531	20
12 operators	138	29	92	31	531	20
16 operators	138	29	92	30	531	20

(d) Image processing

Configuration	Sobel filter		Grayscale filter		Contrast filter	
	States	Contexts	States	Contexts	States	Contexts
4 operators	86	23	13	9	56	18
8 operators	86	23	13	9	56	18
12 operators	86	23	13	9	56	18
16 operators	86	23	13	9	56	18

(e) JPEG encoding

Configuration	JPEG-encoder		Color space transformation		2-D forward DCT		Quantization	
	States	Contexts	States	Contexts	States	Contexts	States	Contexts
4 operators	132	64	22	17	89	43	16	11
8 operators	109	61	18	15	70	42	16	11
12 operators	110	60	17	15	67	39	16	11
16 operators	105	55	17	14	67	36	16	11

the three color components, which resulted in a total of 360 processed input blocks.

6.2. Runtime Acceleration. Except for the contrast and grayscale filter, all applications contained either method invocations or access to multidimensional arrays. As we mentioned above, the synthesis does not support these instruction types yet. In order to show the potential of our algorithm, we inlined the affected methods and flattened the multidimensional arrays to one dimension.

The subsequent evaluations have shown sophisticated results. Speedups between 3.5 and 12.5 were achieved for most kernels. Nonetheless, several applications, for example, SHA-256, gained better results originating from a benefiting communication/computation ratio. The JPEG-encoding application as a whole has gained a speedup of 3.77, which fits into the overall picture.

The encryption of the Twofish cipher is an outlier, being caused by a large communication overhead. This overhead can be reduced by caching objects and arrays inside the CGRA.

In case the cached values did not change since the last usage of a synthesized functional unit, they do not have to be transfered to the reconfigurable fabric again. We evaluated the usefulness of such a caching algorithm [28], but have not extended our synthesis to make use of it yet.

Table 4: Overall utilization of complex processing elements in synthesized functional units.

Configuration	Contexts	0		≤1		≤2		>2	
4 operators	1913	1269	66%	1616	84%	1751	92%	162	8%
8 operators	1748	1210	69%	1496	86%	1589	91%	159	9%
12 operators	1725	1215	70%	1499	87%	1583	92%	142	8%
16 operators	1710	1223	71%	1501	88%	1582	92%	128	8%

Table 5: Largest number of equal operation types executed within a single state on an array with 16 operators.

(a) Round key generation

Operation Type	Rijndael	Twofish	RC6	Serpent
Combinational	7	13	7	16
Multiplication	1	15	0	0
Type conversion	0	1	2	0
Division	0	0	0	0

(b) Single block encryption

Operation Type	Rijndael	Twofish	RC6	Serpent
Combinational	16	14	9	15
Multiplication	0	8	4	0
Type conversion	0	0	0	0
Division	0	0	0	0

(c) Hash/digest algorithms

Operation Type	SHA-1	SHA-256	MD5
Combinational	12	16	4
Multiplication	0	0	0
Type Conversion	0	0	0
Division	0	0	0

(d) Image processing

Operation Type	Sobel filter	Grayscale filter	Contrast filter
Combinational	3	5	5
Multiplication	1	3	3
Type Conversion	1	0	3
Division	1	0	3

(e) JPEG encoding

Operation type	JPEG-encoder	Color space transformation	2-D Forward DCT	Quantization
Combinational	7	13	7	16
Multiplication	1	15	0	0
Type conversion	0	1	2	0
Division	0	0	0	0

The runtime results for all benchmarks are shown in Figure 8 while the corresponding measurement values are given in Table 2.

6.3. Schedule Complexity. In a next step, we evaluated the complexity of the controlling units that were created by the synthesis. Therefore, we measured the size of the finite state machines that are controlling every synthesized functional unit. Every state is related to a specific configuration of the reconfigurable array. In the worst case, all of those contexts would be different. Thus, the size of a controlling state machine is the upper bound for the number of different contexts.

Afterwards, we created a configuration profile for every context, which reflects every operation that is executed within the related state. Accordingly, we removed all duplicates from the set of configurations. The number of remaining elements is a lower bound for the number of contexts that are necessary to drive the functional unit. The effective number of necessary configurations lies between

Table 6: Influence of a specialized operator set with 1 multiplication and 1 division operator on benchmark applications.

(a) Round key generation

Configuration	Rijndael		Twofish		RC6		Serpent	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	17760	—	525276	—	61723	—	44276	—
4 operators	4913	3.61	48509	10.83	3979	15.51	6429	6.89
8 operators	4480	3.96	48236	10.89	3580	17.24	6234	7.10
12 operators	4427	4.01	48236	10.89	3580	17.24	6234	7.10
16 operators	4427	4.01	48236	10.89	3580	17.24	6234	7.10

(b) Single block encryption

Configuration	Rijndael		Twofish		RC6		Serpent	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	21389	—	12864	—	17371	—	34855	—
4 operators	6303	3.39	8683	1.48	2995	5.80	3599	9.68
8 operators	6002	3.56	8629	1.49	2870	6.05	3219	10.83
12 operators	6002	3.56	8620	1.49	2870	6.05	3219	10.83
16 operators	6002	3.56	8620	1.49	2870	6.05	3219	10.83

(c) Hash & digest algorithms

Configuration	SHA-1		SHA-256		MD5	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	23948	—	47471	—	11986	—
4 operators	4676	5.12	4396	10.80	1485	8.07
8 operators	4561	5.25	3484	13.63	1485	8.07
12 operators	4561	5.25	3484	13.63	1485	8.07
16 operators	4561	5.25	3484	13.63	1485	8.07

(d) Image processing

Configuration	Sobel Filter		Grayscale Filter		Contrast Filter	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	21124	—	236	—	608	—
4 operators	2370	8.91	79	2.99	116	5.24
8 operators	2246	9.41	77	3.06	112	5.43
12 operators	2246	9.41	77	3.06	112	5.43
16 operators	2246	9.41	77	3.06	112	5.43

(e) JPEG encoding

Configuration	JPEG-Encoder		Color space transformation		2-D forward DCT		Quantization	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	17368663	—	3436078	—	23054	—	7454	—
4 operators	4868492	3.57	362450	9.48	2996	7.69	1886	3.95
8 operators	4847803	3.58	354721	9.69	2960	7.79	1816	4.10
12 operators	4847803	3.58	354721	9.69	2960	7.79	1816	4.10
16 operators	4847803	3.58	354721	9.69	2960	7.79	1816	4.10

those two bounds, as it depends on the place-and-route results of the affected operations.

The context informations, for the benchmarks are presented in Table 3 while a graphical representation is given in Figure 9. It shows the size of the controlling finite state machine (States), and the number of actually different contexts (Contexts) for every one of our benchmarks. It shows, that only three of eighteen state machines on an array with 16 processing elements consist of more than 128 states. Furthermore, the bigger part of the state machines contains a significant number of identical states regarding the executed operations. Thus, the actual number of contexts is well below the number of states.

Table 7: Influence of a specialized operator set with 3 multiplication operators and 1 division operator on benchmark applications.

(a) Round key generation

Configuration	Rijndael		Twofish		RC6		Serpent	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	17760	—	525276	—	61723	—	44276	—
8 operators	4480	3.96	37939	13.85	3580	17.24	6234	7.10
12 operators	4427	4.01	37582	14.15	3580	17.24	6234	7.10
16 operators	4427	4.01	37582	14.15	3580	17.24	6234	7.10

(b) Single block encryption

Configuration	Rijndael		Twofish		RC6		Serpent	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	21389	—	12864	—	17371	—	34855	—
8 operators	6002	3.56	8413	1.53	2828	6.14	3224	10.81
12 operators	6002	3.56	8404	1.53	2807	6.19	3219	10.83
16 operators	6002	3.56	8404	1.53	2807	6.19	3219	10.83

(c) Hash & digest algorithms

Configuration	SHA-1		SHA-256		MD5	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	23948	—	47471	—	11986	—
8 operators	4561	5.25	3601	13.18	1485	8.07
12 operators	4561	5.25	3502	13.56	1485	8.07
16 operators	4561	5.25	3502	13.56	1485	8.07

(d) Image processing

Configuration	Sobel filter		Grayscale filter		Contrast filter	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	21124	—	236	—	608	—
8 operators	2246	9.41	75	3.15	112	5.43
12 operators	2246	9.41	75	3.15	112	5.43
16 operators	2246	9.41	75	3.15	112	5.43

(e) JPEG Encoding

Configuration	JPEG-Encoder		Color space transformation		2-D forward DCT		Quantization	
	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup	Clock ticks	Speedup
Plain software	17368663	—	3436078	—	23054	—	7454	—
8 operators	4650780	3.73	300618	11.43	2563	8.99	1816	4.10
12 operators	4650780	3.73	300618	11.43	2563	8.99	1816	4.10
16 operators	4650780	3.73	300618	11.43	2563	8.99	1816	4.10

6.4. Resource Utilization. Another characteristic of the synthesized control units, is the distribution of multicycle operations like multiplication, type conversion, or division (complex operations) within the created contexts.

Table 4 shows the aggregate distribution of complex operations within the schedules. It shows a total number of 1913 contexts for all of our benchmarks, as we scheduled them for a reconfigurable array with four operators. Furthermore, it can be seen that a large set of 1269 contexts did not contain any complex operation. Furthermore, the bigger part of the remaining contexts utilized only one or two complex operations, which sums up to 1751 contexts containing two or less complex operations. Hence, only 162 contexts used more than two complex operators.

Entirely, it can be seen that the 1-quantile covers more than 84% of all contexts, regardless of the reconfigurable arrays size. Furthermore, the 2-quantile contains more than 91% of the contexts. Thus, it is reasonable to reduce the complexity of the reconfigurable array, as a full-fledged homogeneous array structure may not be necessary. Hence, the chip-size of the array would shrink. Nonetheless, this would also decrease the gained speedup. The following subsection shows the influence of such a limitation on the runtime and speedup, with the help of small modifications to the constraints of our measurements.

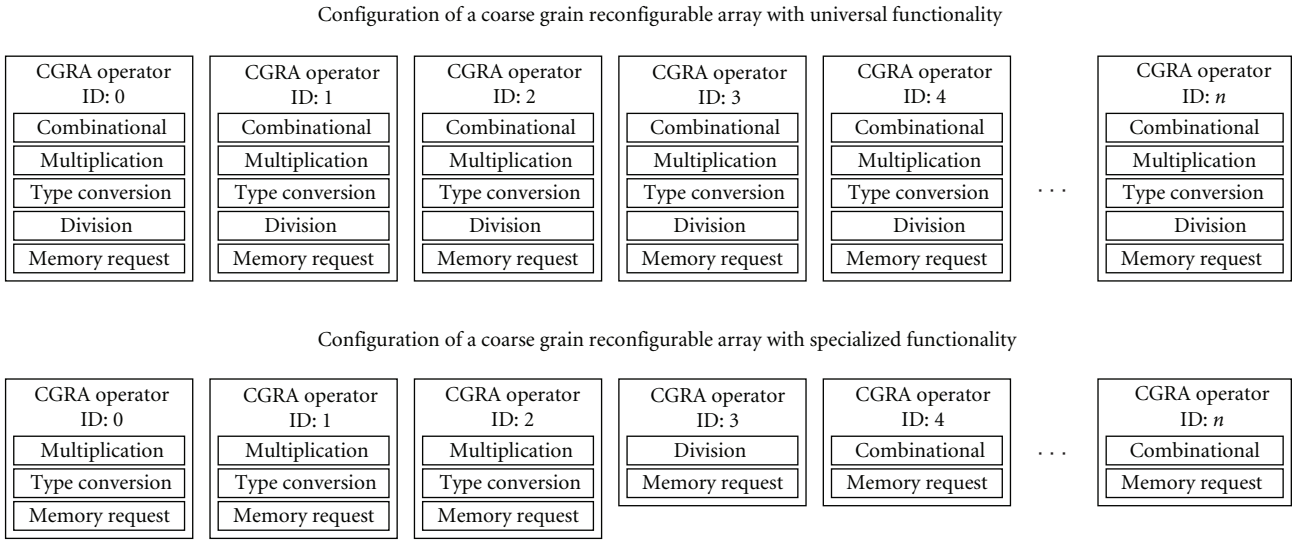


Figure 10: Specialization of an array with 3 multiplication operators and 1 division operator.

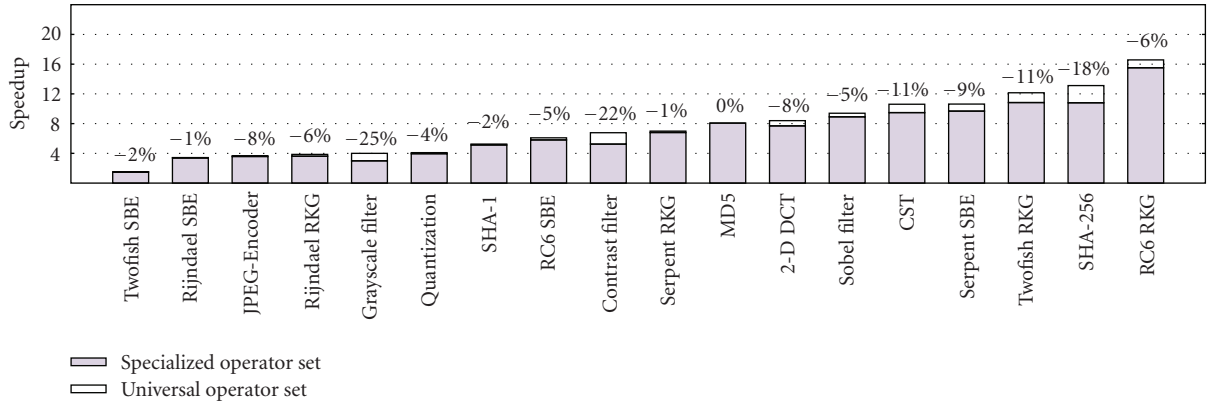


Figure 11: Speedup of benchmark applications on a specialized 4 operator array with 1 multiplication and 1 division operator.

6.5. Passing from Universal to Specialized Operator Sets. The results in the preceding subsections suggest the use of a heterogeneous array, as more than 90% of the contexts that were created by our synthesis algorithm used two or less complex operators. Single operators of this array would not provide the full functionality from the preceding measurements, but a specific subset. Thus, the functionality would be distributed all over the array while reducing the operators chip size and resource consumption significantly.

A well-informed decision about the structure of such specialized operators should be based on an analysis of the distribution of operations within the states of the controlling units. Table 5 shows the largest number of equally typed operations that are executed within a single state for all of our benchmarks. It can be seen that most benchmarks very seldom contain type conversion or division operations. Furthermore, a large subset of benchmarks does not utilize more than three multiplication operators in parallel.

This distribution of operations within the created schedules suggests the use of only a single dedicated division operator inside the array. The number of multiplication operations may be confined to one, as only six benchmarks utilize more than one operation at a time. Nonetheless, type conversion operations have to be executed as well. In case a combined operator for those two operation types is established, the number of its instances may be increased up to three.

All other operators up to the arrays size implement combinational operations, which are the most common instruction type inside the schedules. In addition to their specialized functions, all operators are able to generate a memory read/write request. The exemplified structure of a specialized array with a single division operator, three multiplication/type conversion elements, and one up to n combinational operators is sketched in Figure 10.

6.6. Runtime Impact of Specialized Operator Sets. In order to analyze the effects of the aforementioned specialization, we reconfigured our array to meet the given constraints. Firstly, we measured the runtime of our benchmarks on an array with a single division operator and a dedicated multiplication/type conversion operator.

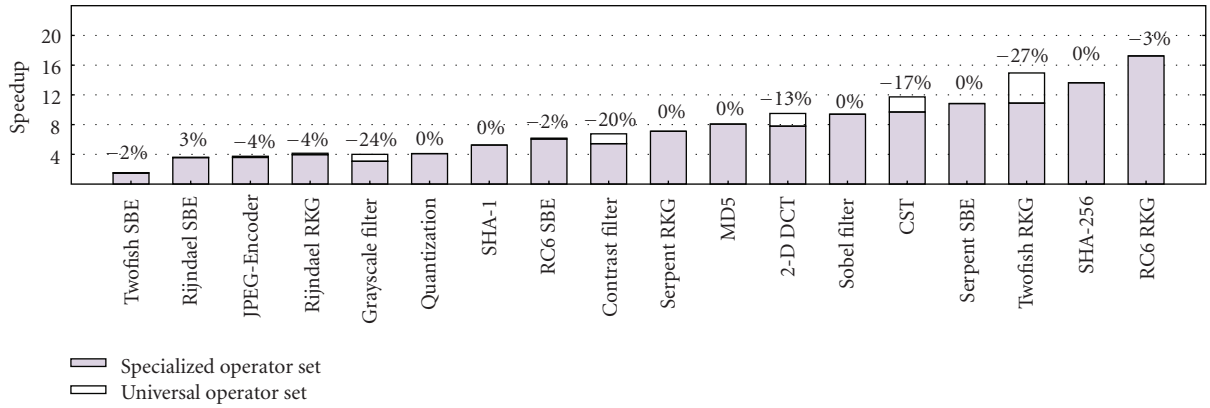


Figure 12: Speedup of benchmark applications on a specialized 8 operator array with 1 multiplication and 1 division operator.

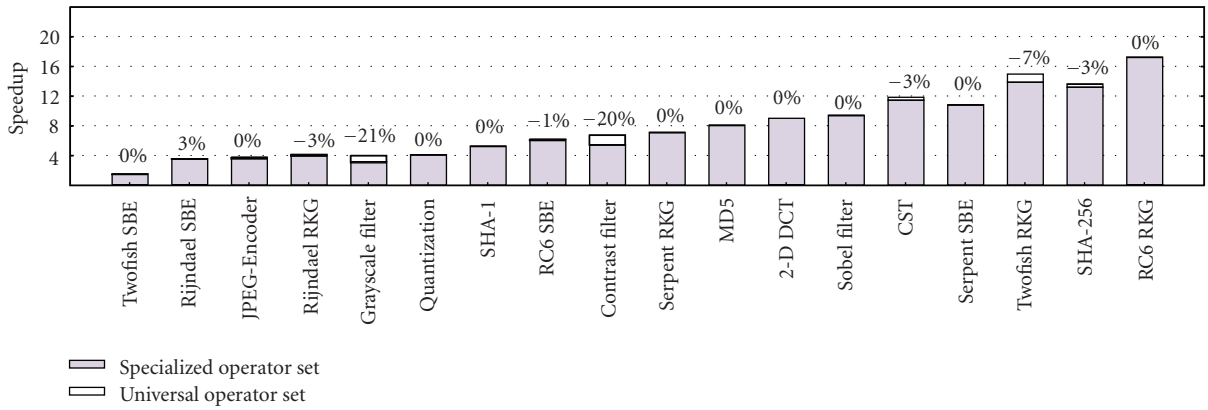


Figure 13: Speedup of benchmark applications on a specialized 8 operator array with 3 multiplication operators and 1 division operator.

The results of the corresponding measurements are shown in Table 6. Just as the measurements regarding an unconstrained reconfigurable array, it can be seen that an array size beyond eight operators does not provide further runtime improvement.

A comparison of the achieved speedups with the results regarding an unconstrained array are displayed in Figures 11 and 12. The annotated percentage numbers display the change of the speedup for a specific benchmark in comparison to its execution at the full-fledged array with the *corresponding size*.

It can be seen that most benchmarks slowed down slightly while only a small number of benchmarks (grayscale filter, contrast filter, color space transformation and twofish round key generation) took speedup losses of two-digit percentage numbers. Furthermore, only two benchmarks on an array with four operators and seven benchmarks on an array with eight operators did not slow down. Due to the heuristic character of the list scheduling, the Rijndael single block encryption slightly improved its runtime.

In a second evaluation iteration, we increased the number of multiplication/type conversion operators to three. Considering the resulting number of four non-combinational operators in this specific setup, it is not possible to evaluate an array of size four, as it does not contain any combinational operators. Thus, none of the benchmarks can be scheduled successfully.

The resulting measurements are displayed in Table 7. Expectedly, there has not been any major runtime improvement beyond an array size of eight operators. A comparison of the achieved speedups with the results regarding an unconstrained array are display in Figure 13. It can be seen that only two benchmarks lose more than 10% of their speedup. Furthermore, the number of lossless benchmark increased to ten out of eighteen.

6.7. Widening the Memory Bottle Neck. The previously shown characteristics of the benchmark applications have shown that most operations are executed parallel to others. As many of our benchmarks rely on array operations, it seems reasonable to allow more than one operation at a time to access the object/array memory. This can be achieved by using a dual ported memory inside the reconfigurable array.

We measured the effects of such an improved memory infrastructure on basis of an eight operator array with three multiplication/type conversion operators and a solely division operator. The achieved speedups in comparison to the execution on a similar array with a single ported memory are displayed in Figure 14, while the corresponding measurements are shown in Table 8. It can be seen that

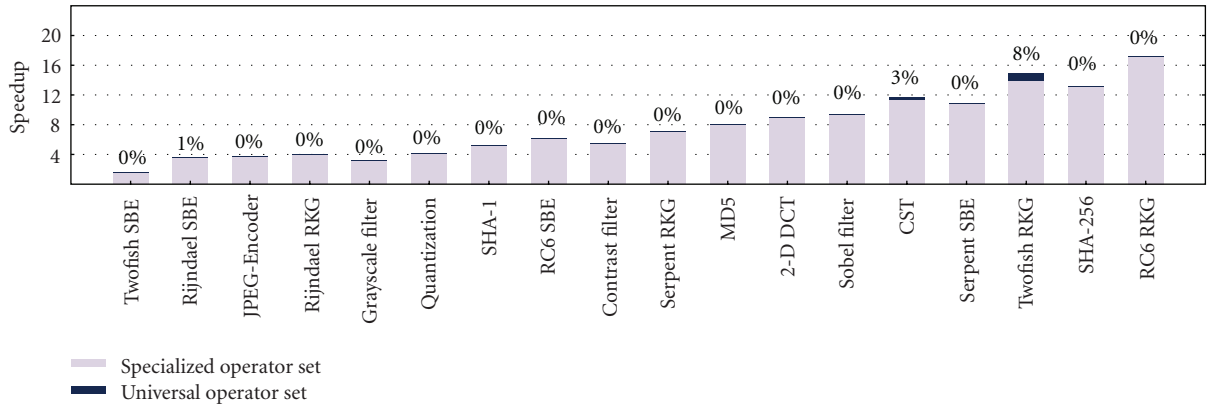


Figure 14: Speedup of Benchmark Applications on a Specialized 8 Operator Array With 3 Multiplication Operators and 1 Division Operator and Dual Ported Memory Acccess.

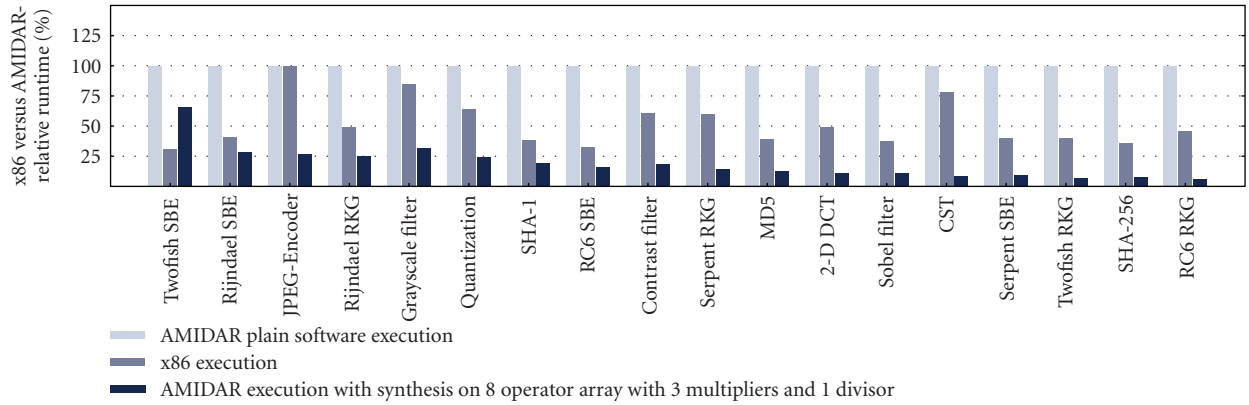


Figure 15: Comparison of AMIDAR plain software execution, x86 execution and AMIDAR execution with enabled synthesis.

only a small number of three benchmarks benefit from the additional memory access operation.

7. Conclusion

In this article, we have shown an online-synthesis algorithm for AMIDAR processors. The displayed approach targets maximum simplicity and runtime efficiency of all used algorithms.

It is capable of synthesizing functional units fully automated at runtime regarding given resource constraints. The target technology for our algorithm is a coarse-grain reconfigurable array. Initially, we assumed a reconfigurable fabric with homogeneously formed processing elements and one single shared memory for all objects and arrays. Furthermore, we used list scheduling as scheduling algorithm.

We evaluated our algorithm by examining four groups of benchmark applications. On average across all benchmarks, a speedup of 7.95 was achieved.

Comparing the runtime of the benchmarks, regarding the underlying reconfigurable fabrics size, shows notable results. An array of eight processing elements delivers the maximum speedup for most benchmarks. The improvements gained through the use of a larger array are negligible. Thus, the saturation of the speedup was achieved with a surprisingly moderate hardware effort.

Furthermore, we displayed the complexity of the synthesized finite state machines. This evaluation showed that most of our benchmarks could be driven by less than 128 states and that more than 90% of these corresponding contexts contained two or less complex operations.

Regarding this distribution of non-combinational operations, we proposed to scale down the full-fledged functionality of the reconfigurable array to a set of specialized operators. These operators are capable of executing a dedicated subset of operations.

Subsequently, we have shown the impact of a specialized operator set onto our benchmarks on four and eight operator arrays. Firstly, we reconfigured the array to contain a single multiplication/type conversion operator and a single division operator while all other operators were combinational.

This configuration resulted in speedup losses for nearly all benchmarks on a four-operator array and ranged up to 25%. Additionally, more than half of the benchmarks on an eight operator array were slowed down to 27%.

As a result of these measurements, we increased the number of multiplication/type conversion operators to three. Only two benchmarks have been affected significantly when

Table 8: Effects of dual ported memory access on an 8 operator array with 3 multiplication operators and 1 division operator.

(a) Round key generation

Benchmark	Clock ticks	Speedup
Rijndael	4480	3.96
Twofish	35264	14.90
RC6	3580	17.24
Sepent	6234	7.10

(b) Single block encryption

Benchmark	Clock ticks	Speedup
Rijndael	5981	3.58
Twofish	8413	1.53
RC6	2828	6.14
Sepent	3224	10.81

(c) Hashes & digests

Benchmark	Clock ticks	Speedup
SHA-1	4561	5.25
SHA-256	3601	13.18
MD5	1485	8.07

(d) Image processing

Benchmark	Clock ticks	Speedup
Sobel filter	2246	9.41
Contrast filter	112	5.43
Grayscale filter	75	3.15

(e) JPEG encoding

Benchmark	Clock ticks	Speedup
JPEG–Encoder	4653315	3.73
Color Sp. Trans.	292889	11.73
2-D DCT	2563	8.99
Quantization	1816	4.10

using this configuration while more than half of the benchmarks did not sustain any losses.

In a final test series, we assumed a dual ported memory inside the reconfigurable array, instead of a single ported memory. This allows an improved scheduling of memory access operations and is supposed to improve the benchmarks runtime. Nonetheless, this approach delivered negligible runtime improvements on only three of the eighteen benchmarks. All other applications have not been affected.

From the presented evaluation results, an array with eight specialized operators (three dedicated multipliers/type converters, one dedicated divider and five combinatorial operations) seems to be the best compromise between speedup and area. Only few applications seem to benefit from two concurrent memory access operations, so a single memory operation should be sufficient.

Furthermore, diagram shows the compared runtime of the AMIDAR plain software execution, the already mentioned benchmark execution on a x86 architecture, and, finally, the execution of an AMIDAR processor with the proposed CGRA extension. It can be seen that most benchmarks outperform the x86 execution. Regarding the achieved speedup of 7.95 across all applications and the plain software execution time, the AMIDAR execution with an enabled synthesis is approximately four times faster than the execution on an x86 processor.

8. Future Work

The full potential of online synthesis in AMIDAR processors has not been reached yet. Future work will concentrate on improving our existing synthesis algorithm in multiple ways. This contains the implementation of access to multidimensional arrays and automatic inlining of invoked methods at synthesis time. Additionally, we are going to explore the effects of instruction chaining in synthesized functional units, as well as the overlapping of a data transfer to a synthesized functional unit and its execution.

Larger numbers of processing elements within the CGRA currently do not seem to have a substantial effect. We hope to improve the usefulness of larger arrays by employing a simplified version of software pipelining.

Also, the interaction of simplified place & route tools and the underlying routing architecture of the CGRA will be an important field of research.

Currently, we are able to simulate AMIDAR processors based on different instruction set architectures, such as LLVM-Bitcode, .NET Common-Intermediate-Language, Dalvik-Executables, and Java Bytecode. In the future, we are planning to investigate the differences in execution of those instruction sets in AMIDAR-processors.

References

[1] S. Gatzka and C. Hochberger, "The AMIDAR class of reconfigurable processors," *Journal of Supercomputing*, vol. 32, no. 2, pp. 163–181, 2005.

[2] S. Gatzka and C. Hochberger, "The organic features of the AMIDAR class of processors," in *Proceedings of the International Conference on Automation, Robotics and Control Systems (ARCS '05)*, pp. 154–166, 2005.

[3] S. Gatzka and C. Hochberger, "Hardware based online profiling in AMIDAR processors," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, p. 144b, April 2005.

[4] S. Döbrich and C. Hochberger, "Low-complexity online synthesis for amidar processors," *International Journal of Reconfigurable Computing*, vol. 2010, Article ID 953693, 15 pages, 2010.

[5] S. Döbrich and C. Hochberger, "Practical resource constraints for online synthesis," in *Proceedings of the 5th International Workshop on Reconfigurable Communication-Centric Systems on Chip (ReCoSoC '10)*, pp. 51–58, 2010.

[6] C. Hochberger, R. Hoffmann, K.-P. Volkmann, and S. Waldschmidt, "The cellular processor architecture CEPRA-1X and its conguration by CDL," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pp. 898–905, 2000.

[7] E. Sotiriades and A. Dollas, "A general reconfigurable architecture for the BLAST algorithm," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 48, no. 3, pp. 189–208, 2007.

[8] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, pp. 12–21, April 1997.

[9] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 507–512, June 2000.

[10] N. Kasprzyk and A. Koch, "Advances in compiler construction foradaptive computers," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 26–29, Las Vegas, Nev, USA, 2001.

[11] A. Koch and N. Kasprzyk, "High-level-language compilation for reconfigurable computers," in *Proceedings of the International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC '05)*, pp. 1–8, 2005.

[12] Y. Ha, R. Hipik, S. Vernalde et al., "Adding hardware support to the HotSpot Virtual machine for domain specific applications," in *Proceedings of the International Conference on Field Programmable Logic (FPL '02)*, pp. 1135–1138, 2002.

[13] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD—reconfigurable pipelined datapath," in *Proceedings of the International Conference on Field Programmable Logic (FPL '96)*, pp. 126–135, 1996.

[14] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen, "PipeRench implementation of the instruction path coprocessor," in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '00)*, pp. 147–158, Monterey, Calif, USA, December 2000.

[15] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Mapping applications onto reconfigurable Kress Arrays," in *Proceedings of the International Conference on Field Programmable Logic (FPL '99)*, pp. 385–390, 1999.

[16] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—a self-reconfigurable data processing architecture," *Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[17] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architecture susing modulo scheduling," in *Proceedings of the Design, Automation and Test in Europe (DATE '03)*, pp. 10296–10301, 2003.

[18] F. Bouwens, M. Berekovic, B. De Sutter, and G. Gaydadjiev, "Architecture enhancements for the ADRES coarse-grained reconfigurable array," in *Proceedings of the 3rd International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC '08)*, pp. 66–81, Springer, Berlin, Germany, 2008.

[19] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the International Conference on Field Programmable Logic (FPL '03)*, pp. 61–70, 2003.

[20] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: rotating instruction set processing platform," in *Proceedings of the 44th Design Automation Conference (DAC '07)*, pp. 791–796, June 2007.

[21] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," *Transactions on Embedded Computing Systems*, vol. 8, no. 3, pp. 1–22, 2009.

[22] A. C. S. Beck and L. Carro, "Dynamic reconfiguration with binarytranslation: breaking the ILP barrier with software compatibility," in *Proceedings of the Design Automation Conference (DAC '05)*, pp. 732–737, 2005.

[23] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*, John Wiley & Sons, New York, NY, USA, 1997.

[24] S. Gatzka and C. Hochberger, "A new general model for adaptive processors," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, pp. 52–60, June 2004.

[25] S. Vassiliadis and D. Soudris, Eds., *Fine- and Coarse-Grain Reconfigurable Computing*, Springer, New York, NY, USA, 2007.

[26] S. Döbrich and C. Hochberger, "Towards dynamic software/hardware transformation in AMIDAR processors," *It—Information Technology*, vol. 50, no. 5, pp. 311–316, 2008.

[27] S. Döbrich and C. Hochberger, "Effects of simplistic online synthesis for AMIDAR processors," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 433–438, December 2009.

[28] S. Döbrich and C. Hochberger, "Predicting hardware acceleration through object caching in AMIDAR processors," in *Proceedings of the International Conference on Automation, Robotics and Control Systems (ARCS '10)*, pp. 162–171, 2006.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 962062, 15 pages
doi:10.1155/2011/962062

Research Article

A Self-Checking Hardware Journal for a Fault-Tolerant Processor Architecture

Mohsin Amin,[1] Abbas Ramazani,[2] Fabrice Monteiro,[1] Camille Diou,[1] and Abbas Dandache[1]

[1] *LICM Laboratory, University Paul Verlaine, Metz, 7 rue Marconi, 57070 Metz, France*
[2] *Electrical Engineering Department, Engineering Faculty Lorestan, University Khorramabad, Iran*

Correspondence should be addressed to Fabrice Monteiro, fabrice.monteiro@ieee.org

Received 1 August 2010; Revised 10 February 2011; Accepted 24 March 2011

Academic Editor: Gilles Sassatelli

We introduce a specialized self-checking hardware journal being used as a centerpiece in our design strategy to build a processor tolerant to transient faults. Fault tolerance here relies on the use of error detection techniques in the processor core together with journalization and rollback execution to recover from erroneous situations. Effective rollback recovery is possible thanks to using a hardware journal and chosing a stack computing architecture for the processor core instead of the usual RISC or CISC. The main objective of the journalization and the hardware self-checking journal is to prevent data not yet validated to be sent to the main memory, and allow to fast rollback execution on faulty situations. The main memory, supposed to be fault secure in our model, only contains valid (uncorrupted) data obtained from fault-free computations. Error control coding techniques are used both in the processor core to detect errors and in the HW journal to protect the temporarily stored data from possible changes induced by transient faults. Implementation results on an FPGA of the Altera Stratix-II family show clearly the relevance of the approach, both in terms of performance/area tradeoff and fault tolerance effectiveness, even for high error rates.

1. Introduction

For years, a substantial research effort has been successfully devoted to increase the performance of processor architectures, while making the best profit of the technological improvements predicted by Moore's law. However, the long-followed approach is reaching its limits. Indeed, the current technological boundaries are raising major constraints on future architectures, particularly in terms of reliability and fault tolerance, given the enlarging rates of physical defects and the increased sensitivity to external disturbances.

Fault tolerance aims at building systems that behave satisfactorily even in the presence of faults. The tolerance of a system is devised to some predefined set of fault types that may include transient, intermittent, or permanent faults, depending on the fault causes being addressed. What can be considered as a satisfactory behavior can vary according to the application domains: a simple error detection with an error alarm indication may be acceptable in some cases while in other cases, the system must ensure operation continuity with no visible impact on the service being delivered.

Transient faults, long while considered as a problem only in space applications and some other critical domains such as nuclear plants, are now becoming a significant threat at sea level. They are now a common source of errors in digital circuits, and their occurrence rate can be rather high compared to permanent faults. Actually, in the search of higher performance, transistors dimensions in each new technological generation have been continuously reduced, allowing higher integration and higher clock rates. However, smaller dimensions lead to lower charges being dealt with, particularly to hold logical states in registers, making new devices much more sensitive to external disturbances. This trend is leading towards an increasing number of soft errors in logic circuits [1, 2], affecting the reliability of those systems. The transient error rates will be far more in future [3].

Sources of disturbance include high energy particles coming from deep space (e.g., cosmic rays and solar wind), natural radioactivity (alpha particles produced by disintegration of radioactive isotopes), electromagnetic fields (e.g., electrical engines, radio frequency sources), and many others. Ratios of transient-to-permanent faults can vary from 2 : 1

to 100 : 1 or even higher [4]. This ratio has been continuously increasing due to higher complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels [5]. This trend may possibly change in the future due to the rapidly increasing rates of physical defects in circuit foundry processes. Still, the transient fault rates are not likely to diminish in the future. Single Event Upsets (SEUs), consisting in a bit state change caused by ions or electromagnetic radiation striking a memorization node, have been frequently addressed [2], using in particular techniques based on the use of error detecting and correcting codes (EDC).

Related work has been dedicated to improve the dependability of digital systems and, in particular, to design digital circuits displaying fault-tolerant characteristics. Hereafter, we reference some prior research that addresses tolerance to transient faults and may be used in processor architectures [6]. The approaches being employed usually rely on the capacity to detect and possibly correct the errors induced by the faults. Actually, one of the main problems that must be faced is to achieve the required levels of tolerance while keeping under acceptable limits the penalty in terms of area, cost, performance, and also power consumption, particularly in embedded systems. On real-time systems, this may be a rather difficult challenge to address.

Processor replication has been used since long as a fault tolerance technique against transient faults [7]. However, this is a costly solution requiring more than 100% of area overhead (and also power overhead), since duplication at least is required for error detection (triplication at least for error correction/masking) and additional voting circuitry. Practically, it is often a more effective compromise to detect errors at register level, specially when SEU are being considered. Triple Modular Redundancy (TMR) is such a classical hardware approach, in which all registers (or the critical ones) are triplicated in order to mask the effect of transient errors affecting one register among the three. Well-known fault-tolerant machines like the LEON FT [8] and the IBM S/390 mainframe processor [9] are TMR-based machines. Double Modular Redundancy (DMR), offering only error detection capability as been used also, for example in the DMR-based server developed by Tandem computers and presented in [10]. Razor [11] is another circuit-level correction mechanism addressing timing errors using supplementary shadow latches to detect delay errors. The DIVA [12] approach is tailored at the micro architecture level.

Software replication recovery [13, 14] with checkpointing [15–17] and re-execution [18] offers much cheaper solutions. However, these techniques tend to induce significant time overheads making severe time constraints hard to match in real-time designs. In redundant multithreaded architectures [19, 20], all instructions are executed twice to detect transient errors. The performance overhead of dual redundancy threads on the AR-SMT (Activestream/Redundant-stream Simultaneous Multi-Threading) pipeline can be significant due to resource contention and checking bandwidth. Dependence-based checking Elision [21, 22] allows to reduce the number of checks and, hence, to improve the overall performance.

A mixed hardware/software N-modular redundant approach is to run virtual machines on separate processors and compare the outputs [23, 24]. Another hardware/software codesign technique is addressed in [25] where little supplementary hardware is used to achieve fault tolerance.

Information redundancy is another classical approach to implement fault tolerance. It relies on the use of error detecting and correcting codes and requires extra circuitry to handle the later (encoders, decoders, and additional storage room). It is largely employed to protect memory devices [26, 27].

In this paper, we are proposing an alternative technique based on the design of a self-checking hardware journal (SCHJ) being used as a centerpiece in our strategy to devise a fault-tolerant processor against transient faults. This strategy relies on two main choices: having built-in hardware error detection capacity in the processor core and using software rollback execution for error recovery. Hardware implementation is the best choice for error detection as it needs to be done permanently and concurrently. Conversely, rollback error recovery is only required on error occurrence. It can be realized through software means if the time penalty remains acceptable, that is, the case when either one or both of the following are true: error rate is low; rollback recovery mechanism is fast. Error rate is an external parameter not under control of the designer that can possibly be high in harsh environments. Consequently, the rollback mechanism should be as fast as possible. Architectural choices for the processor core, and overall, the journalization mechanism and related SCHJ focused in this paper are all devised with this objective in mind.

Section 2 introduces the background motivation and the main architectural choices for the overall processor design. Section 3 presents the principles behind the journalization and fully discusses the architecture and operation modes of the SCHJ. Section 4 presents the main implementation results.

2. Overall Architectural Approach

In this section, we briefly present the basic principles on which the architectural design and the operation of our fault-tolerant processor are built. Actually, the approach we have chosen is largely dictated by our long-term objective: devising a new fault-tolerant massively parallel MPSoC (Multi-Processor System-on-Chip) architecture in which the current fault-tolerant processor design will be used as a building block (as a MPSoC processing node). It was clear from the beginning that severe design constraints concerning the processing node area should apply in order to match the massively parallel objective, yet preserving as much as possible the individual node performance.

Our basic choice for the core processor was to select an architecture in the MISC family (Minimal Instruction Set Computer) instead of the more classic RISC (Reduced Instruction Set Computer) or CISC (Complex Instruction Set Computer) ones. Our architecture [28, 29], inspired from that of the canonical stack processor [30], is able to offer a rather good level of performance with only a limited

amount of hardware being required. Another interesting characteristic is the great compactness of the code running on this kind of processor. On control-oriented applications, in which the quantity of data being manipulated is low and only a small amount of memory is required, it is possible to implement all the required storage in fast memory devices and avoid complex memory cache structures.

The other major reason for choosing an MISC stack computer architecture is related to our approach on how to provide transient fault tolerance to the system. Actually, the very little amount of logical resources and internal storage keeping the state of the processor core allows:

(i) little hardware area overhead to be used to concurrently check its correct operation and, hence, to make the processor core self-checking,

(ii) little time overhead to be used to periodically backup the internal state and to restore it when required on error recovery.

Among other underlying hypotheses, we are supposing that the processor core is to be connected to a dependable memory in which data is supposed to be kept safe fully without any risk of corruption. Actually, a lot of work has been dedicated in the past to the protection of memory devices [26, 27] making this hypothesis pertinent.

Having a self-checking processor core (SCPC) and a dependable memory (DM) device able to safe-fully store data is not enough to build an effective fault-tolerant system. Indeed, the processor may generate undue or corrupted data in sequence to a transient fault. The built-in hardware means will detect the error but will not correct it. Hence, erroneous data may flow into the DM (see Figure 1), and this later can no more be considered as a place where data is to be trusted. In this case, implementing a software recovery mechanism can be a rather painful task with a lot of data redundancy being necessary in the memory device.

The underlying idea behind the journalization mechanism presented in this paper is to prevent untrustable data to flow into the DM and to allow an easy recovery from faulty situations. The basic idea is to implement some hardware device on the path between the SCPC and the DM controlling the way data flows from one side to the other and preventing untrustable data to end up in the DM, as suggested in Figure 2.

Our strategy to implement fault recovery is based on rollback execution, a classic software technique employed in real-time embedded systems [31–33], and relies on the following usual behavior:

(i) program (or thread) execution is split in sequences of fixed maximal length;

(ii) each sequence must reach its end without any error being detected to be validated;

(iii) data generated during a faulty sequence must be discarded and execution restarted from the beginning of the faulty sequence.

For an effective implementation of the above scenario, several points must be taken into consideration. Discarding

Figure 1: Untrusted data flowing into DM.

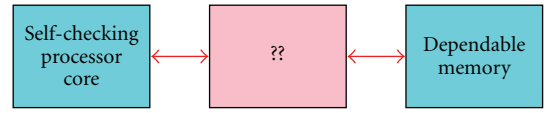


Figure 2: Principle of the overall organization scheme.

data from a faulty sequence, requires the state (data) prevailing at the end of the previous sequence being available somewhere. The DM, as a trustable location, seems to be the ideal location. But there is the need to store the data not yet validated from the current sequence under execution. DM is not the better location as in that case, it would contain untrustable data. Thus, a temporary location for not-yet-validated data is required.

Data stored inside this temporary location can also be corrupted in the consequence of transient faults affecting it, such as SEUs (see Figure 3). Hence, an error detecting and correcting mechanism is required to ensure the reliable operation of this temporary data storage.

In this regard, we are proposing a Self-Checking Hardware Journal (SCHJ) as the dependable temporary storage (DTS) location (see Figure 4). This SCHJ has built-in mechanisms to detect and correct transients errors affecting its content, allowing an effective protection of data during its temporary stay. The basic role of this SCHJ is to hold the new data being generated during the sequence being currently executed until it can be validated (at the end of the current sequence). If sequence is validated, this data can be transfered to the DM. Otherwise, in case of error detection during the current sequence, this data is simply dismissed and the current sequence can restart from the beginning using the trustable data hold in the DM and corresponding to the state prevailing at the end of the previous sequence.

Apart from the data generated by the program/thread code being executed during the current sequence, additional data is to be saved at the end of the sequence and corresponding to the internal state of the processor core, that is, to its internal registers or State determining Elements (SE). The interesting point in choosing an MISC stack computer architecture is the very few bytes of data (corresponding to the SE) that need to be saved. This makes our strategy very effective on control flow applications compared to other fault-tolerant techniques [34]. Indeed, the incurred time penalty being very low, most of the sequence duration (SD) is used for active program/thread instruction execution. Consequently, SD can be reduced and hence the journal size be smaller, leading to a simplified hardware and a shorter average time for not-yet-validated data to remain in the SCHJ. This limits the risk that errors cumulate and become undetectable in the SCHJ. To conclude this analysis, rollback

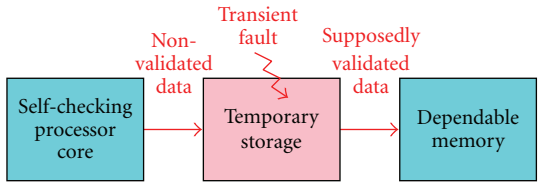


Figure 3: Data corruption in temporary storage.

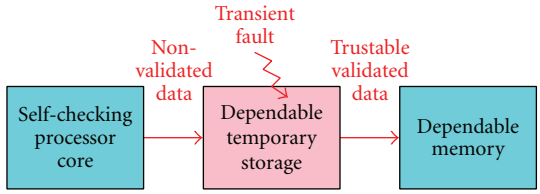


Figure 4: DTS protecting DM from contamination.

error recovery is a favorable choice to be made with our combination of a self-checking MISC stack processor core and a self-checking hardware journal.

Section 3 describes in detail the architecture and operation of the SCHJ in its environment (between the SCPC and the DM).

3. Journal Architecture and Operation

The journal storage space is internally split into two parts, as shown in Figure 6. The upper part contains the not-yet-validated data being generated in the current sequence. This data will be called unvalidated data (UVD) in the remaining of this paper. The lower part holds the data validated at the end of the previous sequence. This validated data (VD) is being transfered to the DM during the execution of the current sequence. At the end of the current sequence, if the sequence is validated UVD turns into VD and, thus, the virtual line separating the upper part from the lower part shifts up to denote the new situation.

Each row in the SCHJ is 41 bits long as shown in Figure 5. The v and w bits will be discussed later. Together with the 16 address bits and the 16 data bits, they represent the information corresponding to a single element being stored in the SCHJ. In order to trust the data temporarily stored in SCHJ, we need a built-in mechanism to detect and correct errors that may occur due to transient faults. Here, we have chosen to rely on error control coding, a classic and effective approach to protect storage devices [27]. The remaining bits in a row, that is, the parity bits, represent the information redundancy related to the error correcting code and protecting the other bits. In the present case, we selected a Hsiao (41, 34) code, a systematic single-error-correction and double-error-detection (SEC-DED) code, Hsiao codes being more effective than Hamming codes in terms of cost and reliability [26].

The overall organization of system composed of the SCPC, the SCHJ, and the DM is depicted in Figure 6. Thanks to the parallel access on reading to the SCHJ and the DM, data can be simultaneously checked in both places. If the data corresponding to a given address is found in the SCHJ, it will be preferred to that present in the DM as it is the most recently written data. If the address is not found in the SCHJ, the data will be read from the DM. As both accesses to SCHJ and DM are done at the same time, there is no time overhead due to an MISS in finding the data in the SCHJ. Actually, whether data comes from the SCHJ or the DM is totally transparent to the processor.

Data is not written directly to the DM in order to insure the trustability of DM: data to be written is always written first in the SCHJ. The corresponding address is always searched in unvalidated area so no two data elements in this area correspond to the same address. If the address is found, the data element is updated. Else, a new row is initialized in the unvalidated area with $w = 1$ and $v = 0$ and the address, data and parity-bit fields filled with the adequate values.

Before transferring to the DM, data awaits for the validation of the current sequence at the validation point (VP). The waiting delay depends on the number of instructions being executed in a sequence, that is, the sequence duration (SD). If no error is found at the end of the current sequence, the processor validates the sequence (sending the validation signal to the SCHJ). All the UVD in the SCHJ is validated by switching the corresponding v bit to 1. Otherwise, if any error is detected, the sequence is not validated and the UVD data in the SCHJ is disclosed by switching the corresponding w bits to 0. As one can easily imagine, the w and v bits are used to denote written and validated data, respectively. Only data having $v = 1$ can be transferred to the DM.

The last instructions in a sequence are used to write the SE to the SCHJ. In our processor core, SE includes the following internal registers: Program Counter (PC), Data Stack Pointer (DSP), Return Stack Pointer (RSP), Top of Stack (TOS), Next of Stack (NOS), and Top of Return Stack (TORS), which is few compared to modern RISC and CISC based processors. On sequence validation, all the data written to the SCHJ during the current sequence get the v bit set to 1 and are consequently sent to the DM. In case the sequence is not validated (see Figure 7), the SE data is restored from memory on rollback as the UVD in the SCHJ is dismissed, and execution is restarted from previous VP. Further explanation on the rollback operation can be found in [28].

As stated before, the on-chip DM is supposed to be fast enough to fulfill the performance requirements of our SCPC. Our strategy of using a SCHJ aims not only to improve fault tolerance but also to allow the rollback mechanism to be used with very little time penalty, and much less area overhead compared to a full hardware approach.

Each row in the SCHJ is protected by a SEC-DEC code as shown in Figure 8. This protection is used the following way:

(i) any error detected in the UVD area will result in the sequence invalidation. One can argue that having a strategy to correct more than a single error is possible but, practically, the hardware overhead necessary to implement it would be too high as all the rows must be checked concurrently;

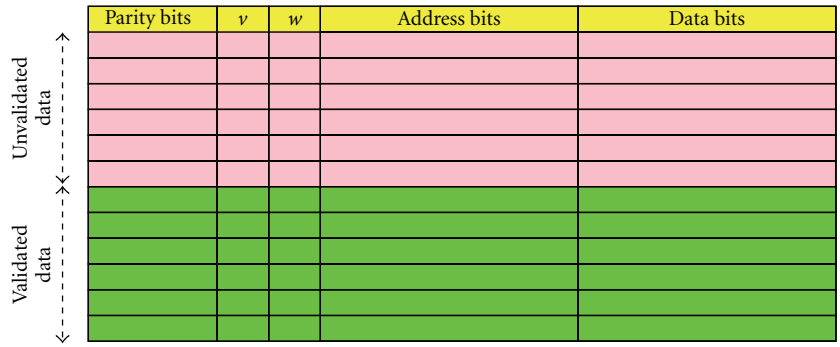


Figure 5: SCHJ structure.

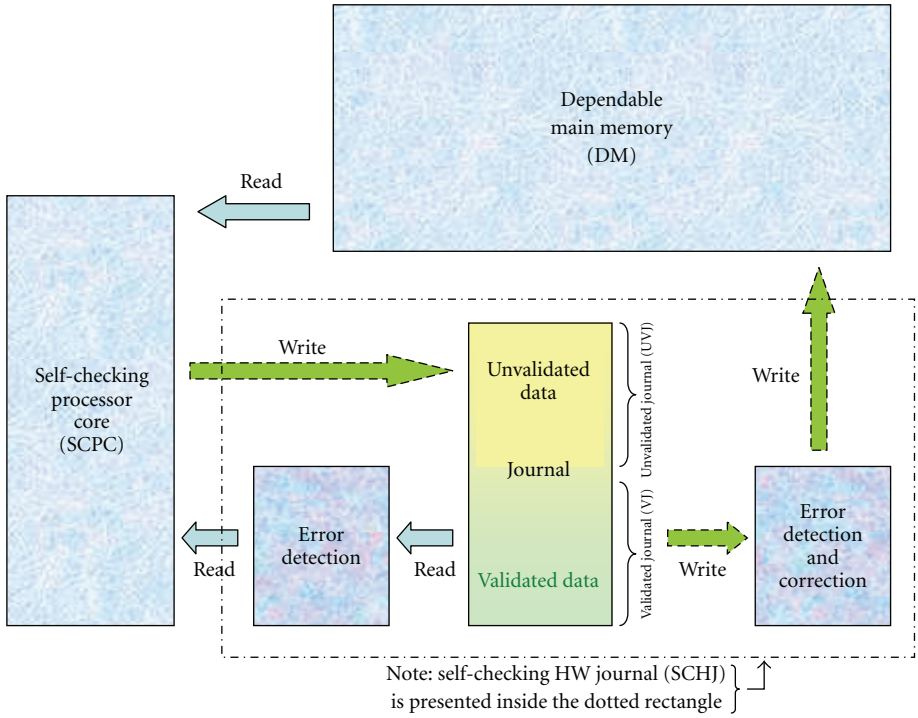
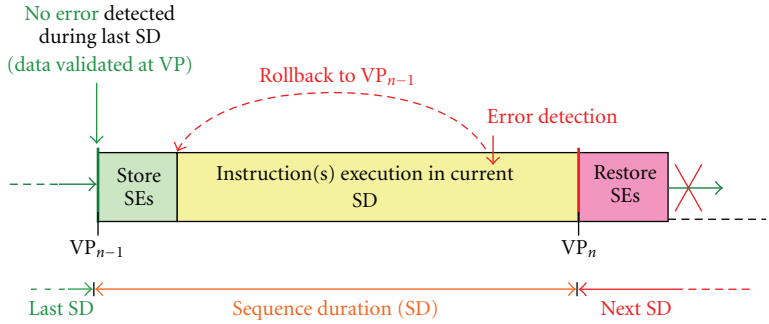


Figure 6: Overall Architecture.



Note: VP: validation point
SE: state-determining element(s) of the processor

Figure 7: Rollback mechanism on error detection.

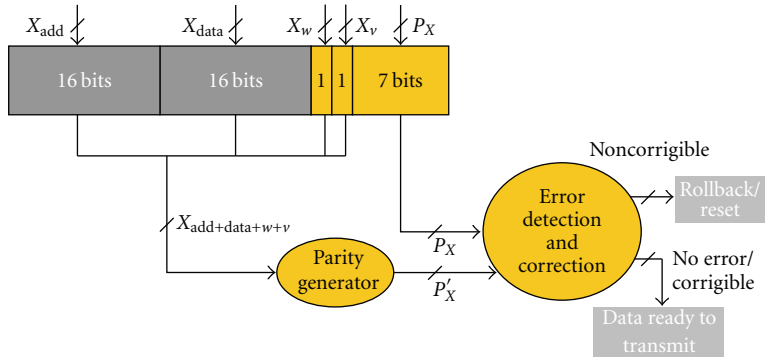


Figure 8: Error detection and correction in journal.

(ii) VD area is written row by row to the DM. The overhead related to the hardware mechanism necessary to correct a single VD row before it flows to the DM is thus acceptable. It is to be noticed that the VD still in the SCHJ is the only copy of the latest validated sequence. Thus, throwing away this data would avoid correct completion of the program/thread execution and require a system reset. This may still happen if an error is detected that overpasses the correction capacity of the code (e.g., a double error in a single VD row).

The overall operation of the SCHJ is depicted in the self-explanatory flow chart of Figure 9. Four modes of operation can be distinguished which are summarized in Table 1. The traffic signals in Figures 10, 12, 14, 15 are with respect to write-operation.

Mode 00. This mode is active on start of program or whenever a noncorrigible error is detected in a VD of the SCHJ. In this mode, the processor is reset, the execution starting (or restarting) from default initial values. All the data stored in the journal is cleared, with all the w and v bits being set to 0. There is no data exchange between the SCPC, the SCHJ, or the DM, as shown in Figure 10.

Mode 01. This is a normal read or write mode depending on the active instruction in the SCPC (rd = 1 or wr = 1). In this mode, the SCPC can write directly into the SCHJ but not into the DM, in order to avoid any risk of data contamination in the DM. Read access is allowed both from the SCHJ and the DM (not shown in Figure 12 to avoid complexity). The data read from the SCHJ is checked for possible errors. On error detection, the processor enters mode 11 in which rollback mechanism is activated without waiting for the VP of the current sequence.

A deeper analysis of this mode is useful because most of the execution time corresponds to this mode (in average, more than 80% execution time is in mode 01). As shown in Figure 11, when the processor wants to read from the SCHJ, the address tags are checked to match the requested data element (arrow *a* in Figure 11). If the corresponding address is found, the related data is checked for possible errors before its transfer towards the SCPC. The checking is done comparing the stored parity bits (of the employed Hsiao code) to regenerated parity bits in the error detection unit (shown in Figure 11).

Table 1: SCHJ operating modes.

Modes	Operation
00	Initialization
01	Read/Write
10	Validate ($v = 1$)
11	Invalidate (rollback)

(i) If an error is detected (shown in Figure 11), the rollback mechanism is invoked because data contents in UVJ contains data generated during the current sequence (denoted by the v field set to 0). The *Enable* signal on the data bus is then set to 0 to forbid further data transfers from the SCHJ to the SCPC. All the data contents written during this sequence are considered as garbage values ($w \leq 0$).

(ii) If no error is detected, the requested data is sent to the SCPC. The data bus between the SCHJ and the SCPC is activated while the data bus between the SCHJ and DM is temporary unactivated. Note in Figure 11: the arrow 1 shows the $w = 1$ which indicates the data written in SCHJ while the arrow 2 shows the data validated during the previous sequence.

Mode 10. This mode relates to the transfer of validated data (VD) from the SCHJ to the DM (see Figure 14). This mode is active at VP when no error has been detected during the current sequence. The sequence is thus to be validated, and all the data written into the SCHJ during this sequence must now have its v bit set to 1 ($v \leq 1$). In consequence, the transfer of this freshly validated data to the DM is allowed, and will be processed on following clock cycles, based on the availability of the data bus.

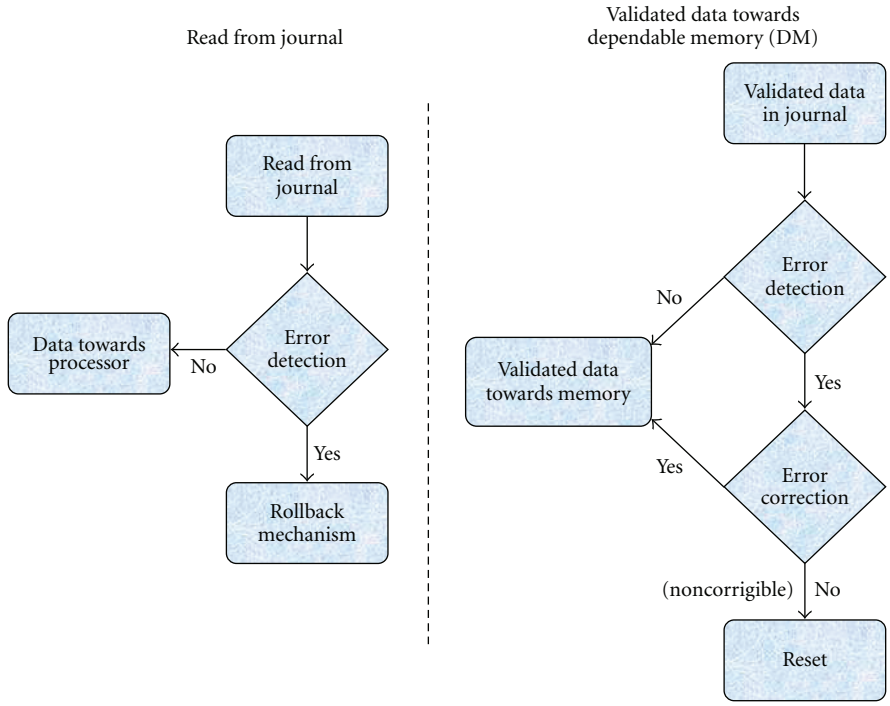


Figure 9: SCHJ operation flow chart.

The VD elements being written to DM are checked for errors affecting them as a consequence of transient faults that may have occurred during their stay in the SCHJ.

(i) If a corrigible error is detected, then data is send to the DM after correction;

(ii) if a noncorrigible error is detected (the probability for this to happen is very low normally), for example, a double error affecting the same memory block (see arrow b in Figure 13), then an unrecoverable situation arises as there is no way to rollback to a previous VP (the corresponding data being no more available in the DM). Resetting the SCPC and switching to mode 00 (and possibly raising some alarm indicator) is the usual behavior in this situation.

Mode 11. This mode is invoked when an error is detected during the read/write-operation as shown in Figure 15, and it has been partially discussed with mode 01. In this mode, all the data written in UVJ of SCHJ (i.e. all the data generated during the current sequence) is invalid and discarded ($w \leq 0$) as shown by the cross in Figure 11. Rollback is invoked to restart execution of the current sequence from the begin (after restoring the last valid SE). On successful restore, the mode 01 (read/write-mode) is activated.

4. Results

The SCPC and SCHJ have been modeled in VHDL at the Register Transfer Level (RTL) and implemented on a Altera Stratix II EP2S15F484C3 device and the Altera QuartusII.

The area dedicated to the SCHJ depends on the number of writes to different addresses that can be supported in a single sequence. This depends both on the sequence duration and on the rate of "writes" in the sequence. The later depends largely on the program being executed. Figure 16 depicts two extreme cases:

(i) a sequence of Nb "DUP" instructions will generate Nb "writes" to different addresses ("DUP" duplicates the top stack data element). Indeed, on each "DUP," the stack grows of one additional position generating a "write" to a new address in the SCHJ ("TOS" → "NOS" → SCHJ shift). Hence, Nb positions are necessary to hold the UVD of the sequence in the SCHJ. In the worst case, where two sequences of this kind follows one another, $2 * Nb$ positions could be required to hold the UVD and VD together. The practical value is closer to Nb than $2 * Nb$, as VD will be progressively written to the DM;

(ii) a sequence of any length of "SWAP" instruction will not generate any "write" to the SCHJ. Indeed, on each "SWAP", only "TOS" and "NOS" contents are swapped without any occurrence of a "write to" or "read from" the SCHJ.

In practice, considering the worst case is not very realistic and an average case is more indicated, which can be evaluated from representative benchmarks of the target applications to be executed on the processor.

Finding the optimal depth for the journal is a key issue in obtaining an acceptable node size for the future MPSoC architecture. Hence, we have investigated the impact of the

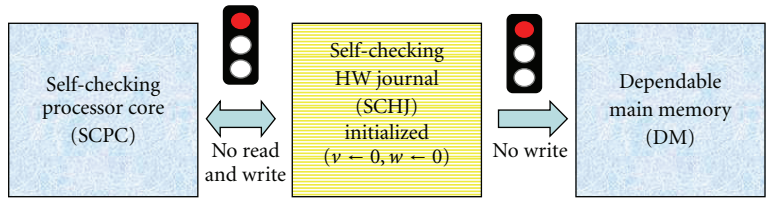


Figure 10: SCHJ relations in mode 00.

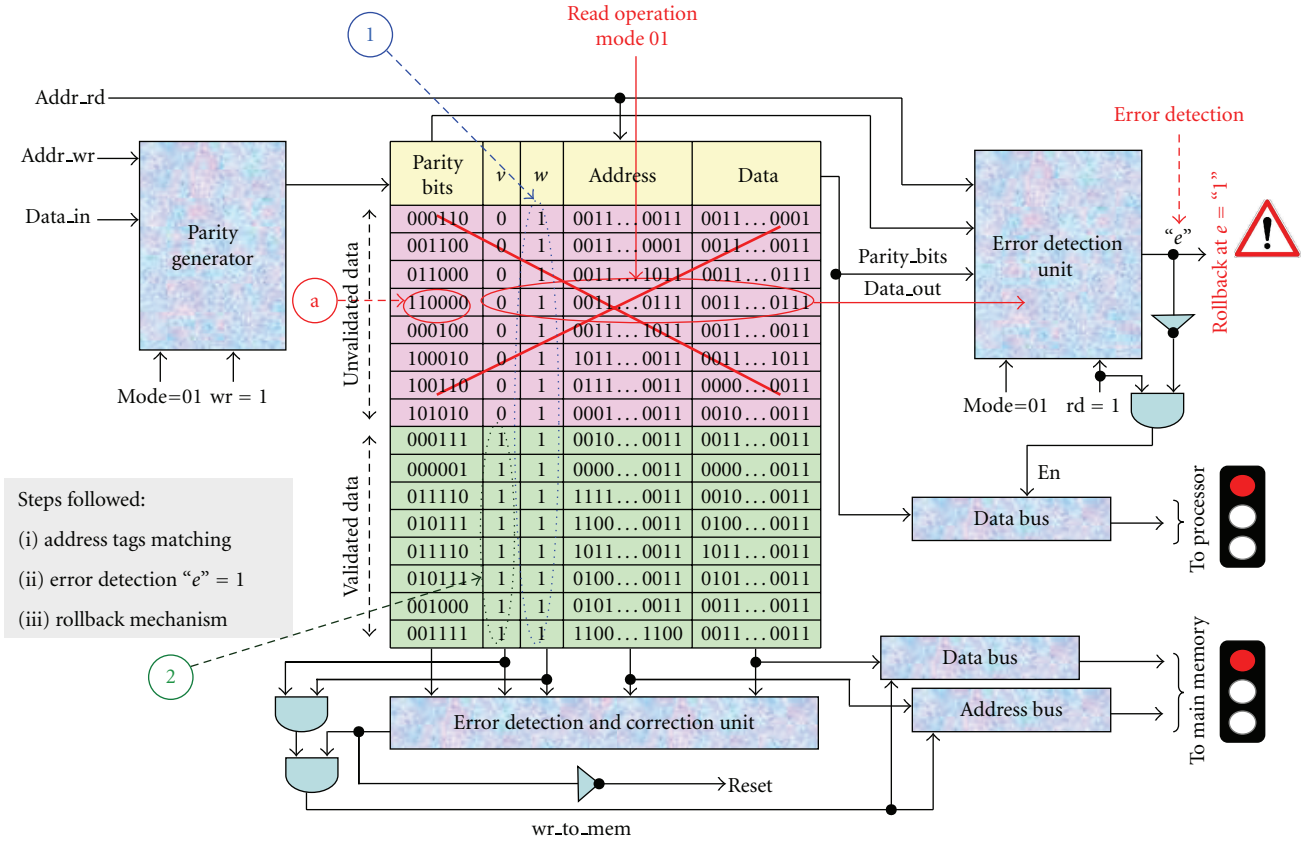


Figure 11: Reading UVD from SCHJ in mode 01.

SCHJ depth on the total area (SCHJ + SCPC). The results are reported in Figure 17 where vertical axes scale is in hundreds of combinatorial ALUTs.

The case of a journal depth of 50 is analyzed in Table 2. Here, the area dedicated to the SCHJ is about 54%. However, the total area required for the SCPC and the SCHJ together is reasonably small making it suitable for integration as an active node of an MPSoC. The analysis of varied algorithms shows that, most of the time, a depth of 50 is more than enough, due to data locality. Using a stack computing-oriented programming style enforces even more data locality on the stack, which in turn results in a smaller journal depth being required.

4.1. Validation by Error Injection. The overall architecture operation has been checked with fault injection techniques.

Table 2: Implementation area.

	Comb. ALUTS	(Ded. Logic)
SCPC and SCHJ	2400	(1295)
SCHJ alone	1305	(726)

Fault injection is a method to observe and evaluate the behavior of a system in a controlled experimental environment in which faults are introduced voluntary in the system [35]. In our case, we are injecting transient faults by simulation, that is, the faults are injected altering the logical values during the simulation. Simulation-base injection is as special case of error injection that has been widely used at different levels of abstraction such as logic, architectural, or functional [36].

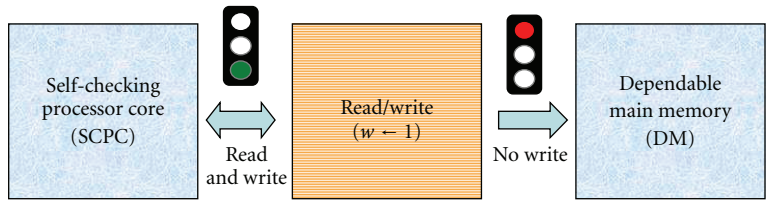


Figure 12: SCHJ relations in mode 01.

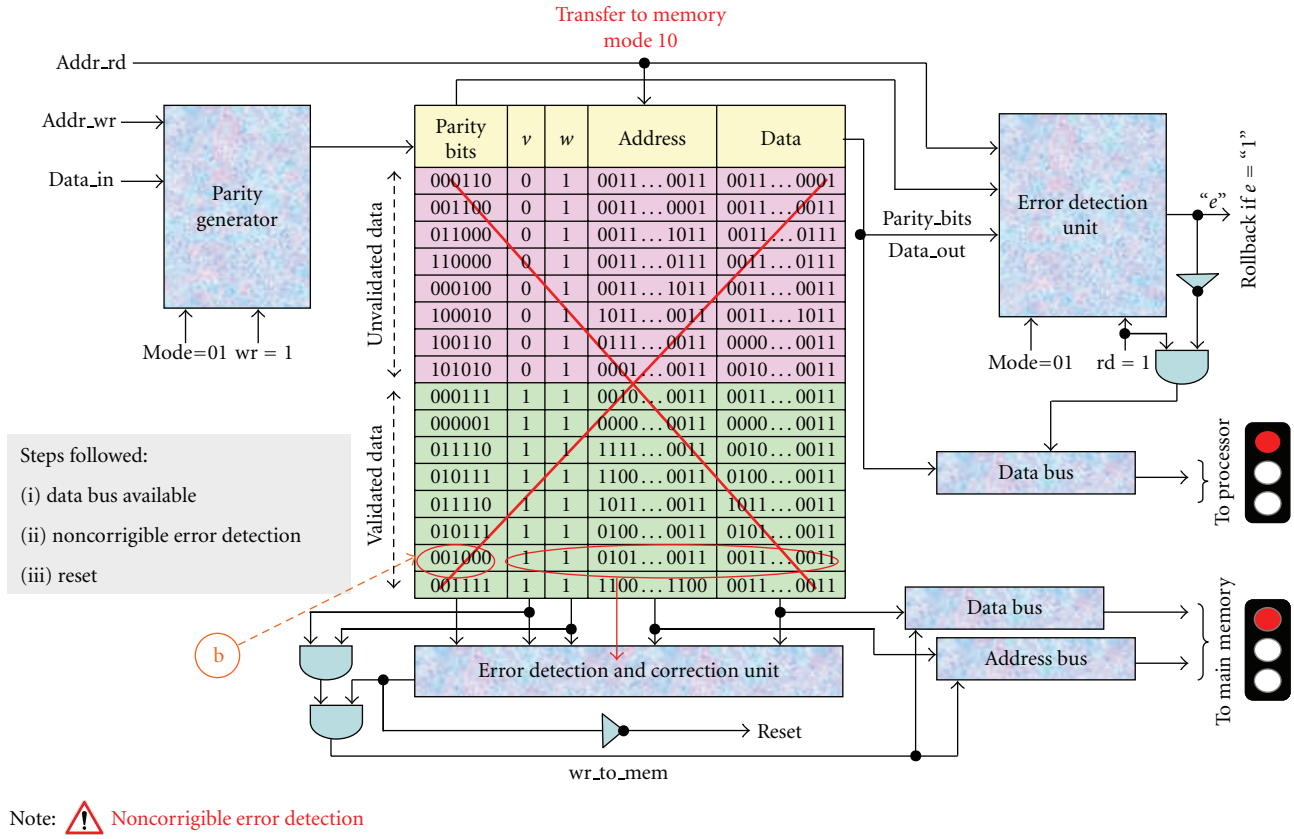


Figure 13: Mode 10 of SCHJ operation.

In order to allow very fast simulation (and hence, allow a large number of simulation campaigns to be conducted in a reasonable delay), dedicated C++ tools have been developed to replace the usual "discrete event driven" simulation model on which VHDL relies, by the faster "cycle driven" simulation model that fits very well synchronous designs [37]. For the simulation, strictly equivalent C++ "cycle drive models" replaced the original VHDL models at RTL level used for synthesis.

Our goal in this section is to check the fault tolerance properties of the full system, that is, to verify its correct operation in the presence of faults, as expected from the specifications.

Figures 18 and 19 show two different simulation situations: a recoverable and a unrecoverable error, respectively. In the first case, a detectable error occurrence in the unvalidated part of the journal (UVD). As seen on Figure 18, on error detection, the system rollbacks to the previous saved sure state. In Figure 19, a noncorrigible error is detected in the validated part of the journal (VD). Rollback cannot occur as it cannot recover the error, the data to recover being no more present in any place (DM or SCHJ). The only choice is hence to reset the processor and clear the SCHJ.

The faults considered are SEUs (Single Event Upsets: only one bit changes in a single register) and MBUs (multiple bits change at once in one register), injected at a randomly chosen moment. The corresponding error patterns are shown in Figure 20: (a) SEU (single bit error); (b) 1 up to 2-bit MBU; (c) 1 up to 3-bit MBU; (d) 1 up to 8-bit MBU. These fault patterns are commonly used ones in RTL models [35, 38].

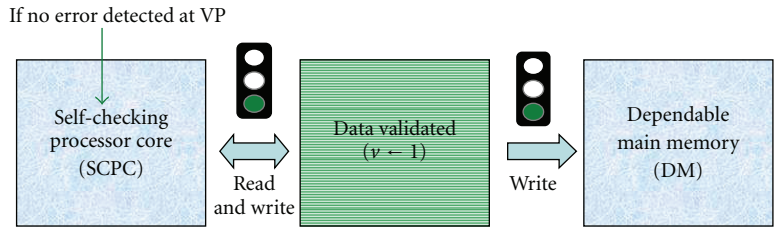


Figure 14: SCHJ relations in mode 10.

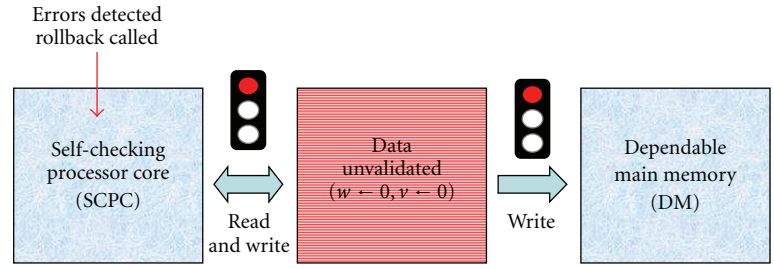


Figure 15: SCHJ relations in mode 11.

Note: the affected bits can be any, not necessarily those shown in the figure.

Different campaigns were conducted for each of the different error patterns. The results are presented in Figures 21, 22, 23, and 24, respectively. On error injection, the simulation is tracked to check whether the error is detected or not within a maximum of 2 cycles (the maximal detection latency of the processor). The simulation is hence stopped after two cycles in either cases, whether the error is detected or not. Then, a new simulation starts. The figures compile the cumulated results for each kind of error pattern.

The graph in Figure 21 shows that the proposed journalized processor architecture is 100% effective in detecting single bit random errors. The results remain reasonable, being higher than 60% and 78% for 2-bit and 3-bit errors, as shown in Figures 22 and 21. Despite the hard job the architecture must face with the last experimental conditions, using wide (1 to 8 bit) random error patterns, the error detection rate remains reasonable, with values greater than 36% for all the configurations. This clearly shows the effectiveness of the architectural approach being used to protect the processor against transient errors.

4.2. Performance Evaluation. There are two main limiting factors to the speed performance of the proposed fault-tolerant processor architecture.

(i) The encoding and decoding blocks in the journal, limiting the maximal operation frequency, as they are located in the critical path of the data incoming and outcoming the SCHJ. Some additional optimization is actually possible using some pipelining strategies in the journal (particularly on the "write to DM" path') and some advanced techniques for the error control coding implementation.

(ii) The delay induced by the rollback mechanism when triggered by the errors being detected. This delay is directly related to the amount of code requiring to be executed again. This in turn, depends on the injected error rate and on the length of sequences. Indeed, longer sequences are more prone to errors than shorter ones. Thus, for a given error rate, sequences are more likely to be faulty and require re-execution. It is to be noticed, however, that the journalization mechanism requires some execution overhead to backup the SCPC internal state. Thus, reducing too much the length of sequences will also have a negative impact. Theoretically, it is possible to determine the optimal sequence duration minimizing the delay due to re-execution [39]. However, in real-life conditions, it is generally not easy to estimate the the actual error rate.

To evaluate the impact of transient errors on speed performance, we have measured the degradation of execution time for different error injection rates (using the previously indicated error patterns) and different sequence duration on 3 different sets of benchmarks. In the corresponding graphs (cf. Figures 25, 26, and 27), time is measured as "number of Clock Per Operation" (CPO) and is plotted against the "Error Injection Rate" (EIR). The "no injected error" cases are the reference cases for each sequence duration SD, and match the fixed CPO value of "1."

The 3 sets of benchmarks are the following: group I contains algorithms where logic and arithmetic operations dominate; group II addresses memory manipulation algorithms (e.g., permutation of memory elements); group III is representative of typical control dominated algorithms. Table 3 summarizes, for each group of benchmark, the percentage profiles of "read from"/"write to" memory (SCHJ

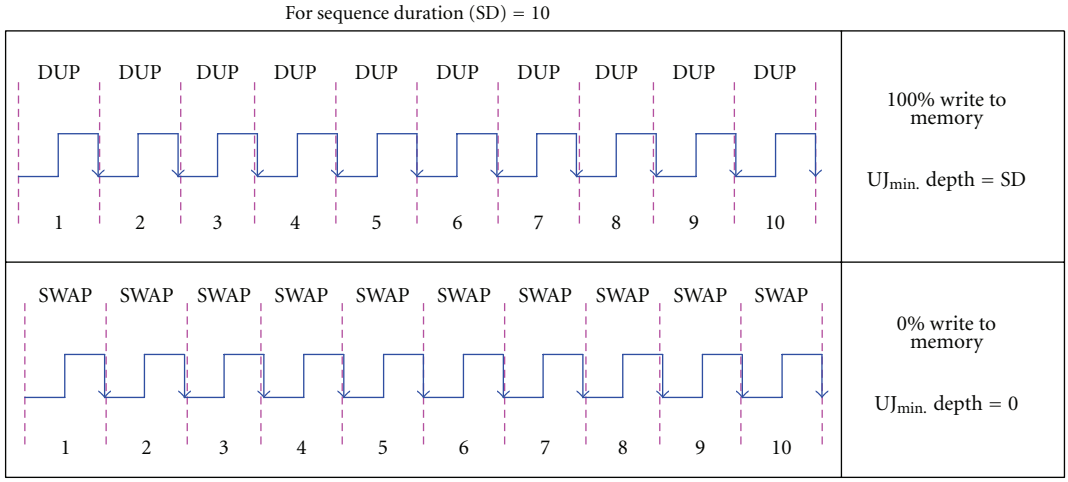


Figure 16: "Writes" rate for 2 different benchmarks.

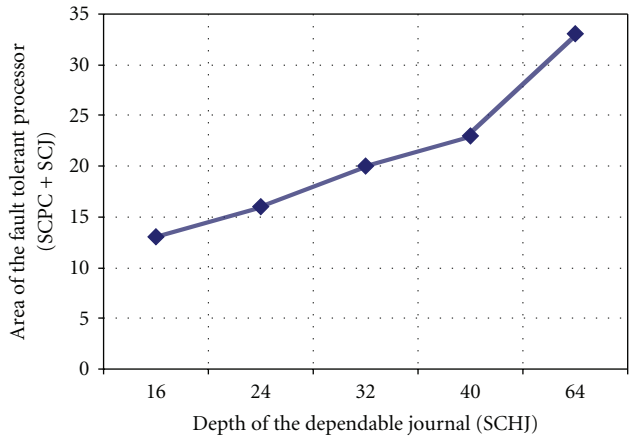


Figure 17: SCHJ/SCPC area ratio to SCHJ depth.

Table 3: Read/Write profiles in benchmarks groups.

Group	Read	Write
I	45%	39%
II	57%	38%
III	50%	38%

and DM) induced by the instructions running on the SCPC. Note that the instruction set of the SCPC has 36 instructions among which a majority (23 instructions) involve reading or writing from/to the memory.

Figures 25, 26, and 27 summarize the results for the benchmarks in Group I, Group II, and Group III, respectively. The analysis of the graphs show that the curves tend to overlap for the lower values of EIR. This is logic as, in absence of error, no time penalty due to rollback is induced whatever the benchmark being used.

In Figure 25, moving from point A to B incurs an increase of 10× in the error rate. The corresponding increase in CPO remains negligible for SD = 10 and 20. It remains low for SD = 50 with only a 10% overhead. The overhead becomes relatively important for SD = 100 with a value of about 60%. Moving from B to point C, a new error rate increase of 10× is applied. Now, the overhead is more important. Yet, it remains lower than 100% for SD = 20 and SD = 50, and still negligible for SD = 10. Similar observations can be seen in Figures 26 and 27.

In order to observe more directly, the effect of the error rate on the rollback mechanism, we have compiled the results for Group III in the graph of Figure 28, where rollback re-execution rate is measured for different error rates and different sequence durations. As expected, the rollback rate (and hence, the rollback penalty) is lower for the lower error rates. Furthermore, the impact of higher error rates is much larger for longest sequences than for the shortest ones.

With higher EIR, the smaller SD are the ones that denote the lower time penalty being incurred. This is also coherent with predicted results. Indeed, for a given error rate, the probability for a sequence to be invalidated is higher for a longer SD, hence leading to a higher rollback rate.

Taking into account that the architecture chosen for the SCPC requires little time to save the SE, it is possible to select a short SD and still have a good level of performance. Furthermore, this allows a lower SCHJ depth to be chosen with a reduced area consumption and diminishes the risk that errors cumulate in the SCHJ and induce a nonrecoverable error.

4.3. Comparison with FT-LEON-3. The LEON-FT is one of the most popular fault-tolerant processors of the last decade. It uses TMR (triple modular redundancy) to implement fault tolerance. In this section, we have chosen LEON FT-3 (the latest FT-version from LEON) to make some comparision with the approach proposed in this paper. It is the successor of the ERC32 and LEON processors, developed for the European Space Agency (ESA). The LEON FT-3 has been designed for operation in the harsh space environment and includes functionality to detect and correct (SEU) errors

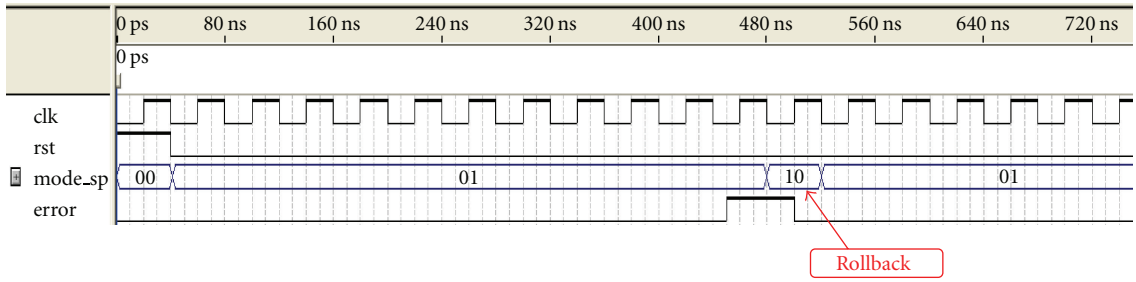


Figure 18: Recoverable error detection.

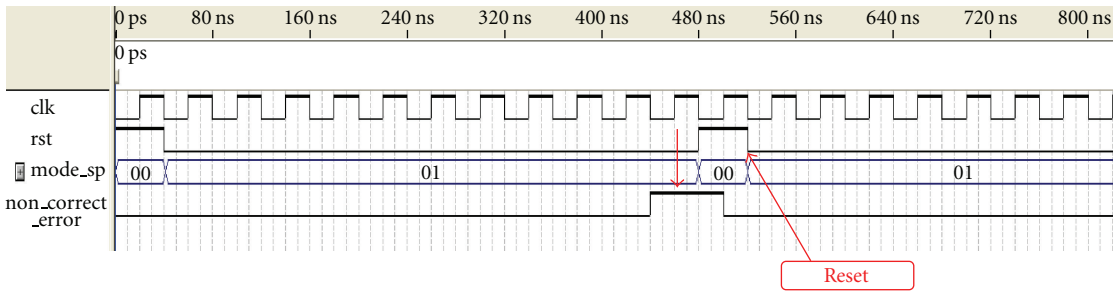


Figure 19: Unrecoverable error detection.

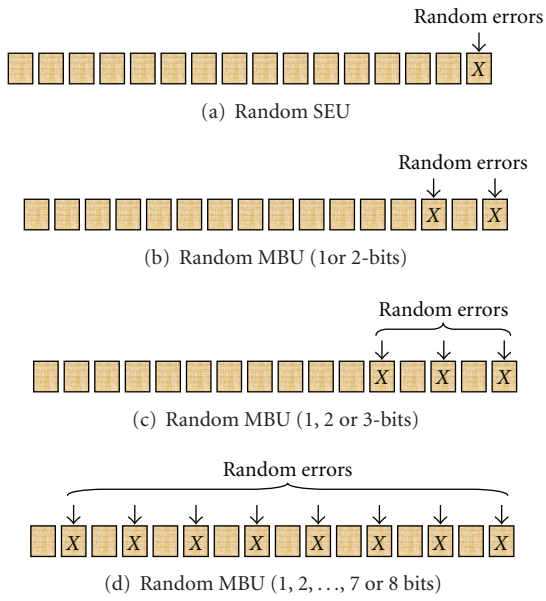


Figure 20: Error patterns for fault injection.

in all on-chip RAM memories [40]. We have chosen this processor for comparison because: (i) it is a well-established FT-processor, (ii) it has been used in MPSoC designs, and (iii) it is an open source machine.

Although it is difficult to compare the fault-tolerant strategies employed for these two very different processor architectures (a simple "low-budget" MISC stack processor and a rather complex and sophisticated RISC processor), it is interesting to check the validity of our approach compared to the well established approach of the LEON FT-3.

Table 4: Comparison of LEON-3 FT and Journalized Stack Processor.

	LEON-3 FT	Journalized Stack
Max. MHz	150	110
Pipelining	7-Stages	2-Stages
Register size	32-bits	16-bits
FT Method	Triple Modular Redundancy	Error Correcting Codes
100% Error Detection	4 bits out of 32 bits	1 bit out of 16 bits
Area	3500	1151

For fair comparison, both were implemented on Altera Stratix-III. The implementation results are reported in the Table 4. The parameters chosen for the comparison are the maximum clock frequency, the area, the error detection/correction approach, and the number of pipeline stages. Note that the area overhead related to the journal has not been included. Actually, this overhead can remain rather low for a journal tailored for a sequence duration of less than 16 (which is actually very effective).

The comparison shows that the LEON 3-FT requires more than three times the area of our processor. On the other hand, the maximum frequency of LEON3 FT is higher. Practically, there is still a margin of optimization available in our design, particularly in the journal, where additional pipelining can be used to reduce the critical path in the encoding and decoding circuitry for the error correcting code.

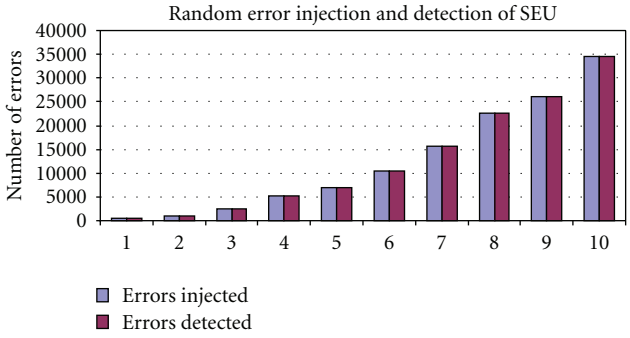


Figure 21: Results for single bit error injection.

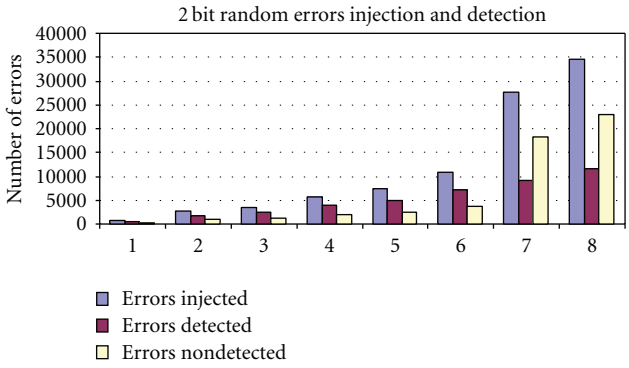


Figure 22: Results for 2 bit error injection.

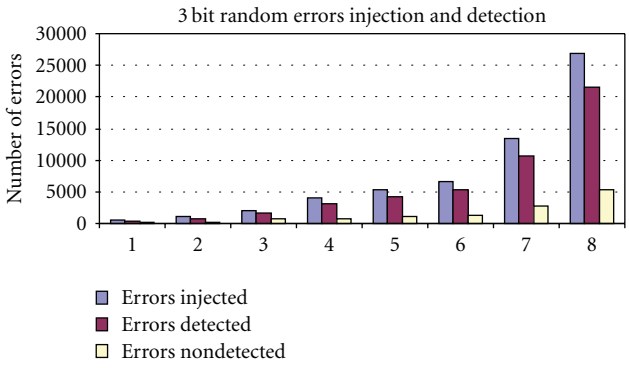


Figure 23: Results for 3 bit error injection.

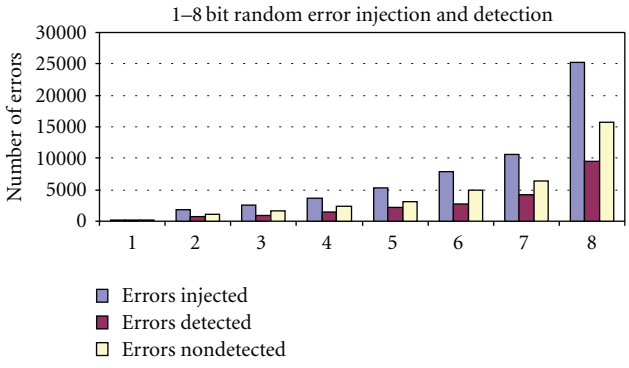


Figure 24: Results for 1–8 bit error injection.

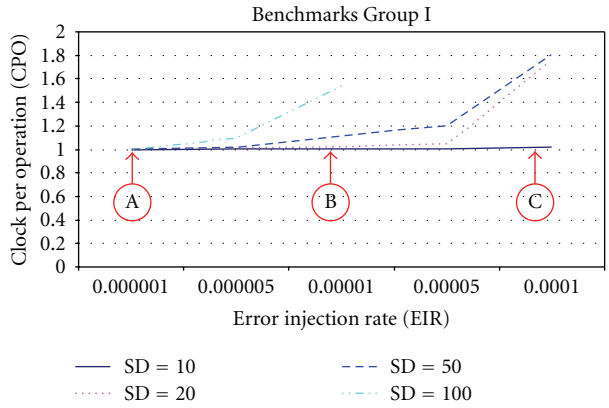


Figure 25: Simulation Curves for Group I.

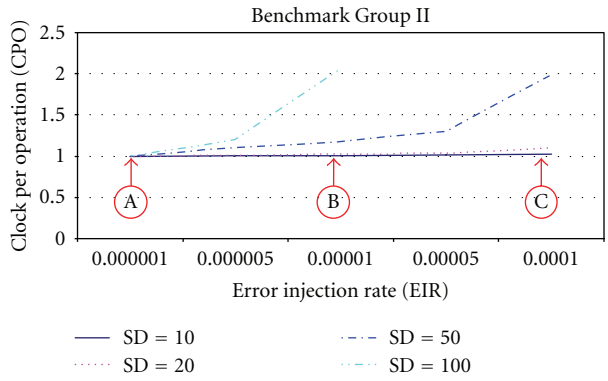


Figure 26: Simulation Curves for Group II.

LEON 3-FT is able to detect all error patterns up to 4-bits whereas the capability of the present version of our processor is only guarantied to single bits in each SCPC's internal register, and up to 2-bits in each word of the SCHJ.

However, this short comparison demonstrates that the principle of journalization can be rather effective on a stack computing-based processor core architecture and deserves more research effort to enhance the performances and protection capability.

5. Conclusion

In this paper, we present a specialized self-checking hardware journal being used as a centerpiece in a design strategy to build a transient fault-tolerant processor later to be used as a building block in massively parallel fault-tolerant MPSoC architecture. Together with the choice of the MISC stack computer architecture for the processor core (instead of RISC or CISC), it allows the combination of hardware error detecting techniques and error recovery through software rollback recovery to be a very effective approach to fault tolerance. The self-checking hardware journal is central to the journalization, the key functionality for fast rollback. This journalisation scheme is made possible thanks to the simple memory organization permitted by the processor core architecture choice.

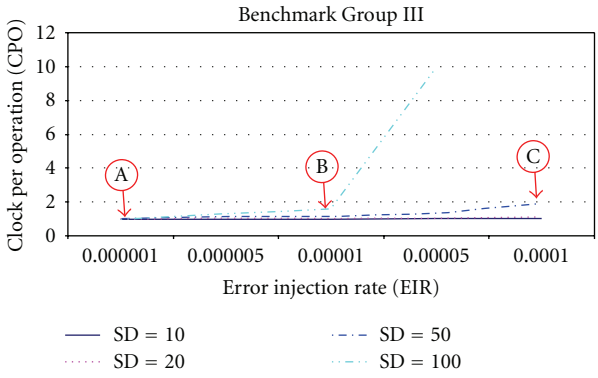


Figure 27: Simulation Curves for Group III.

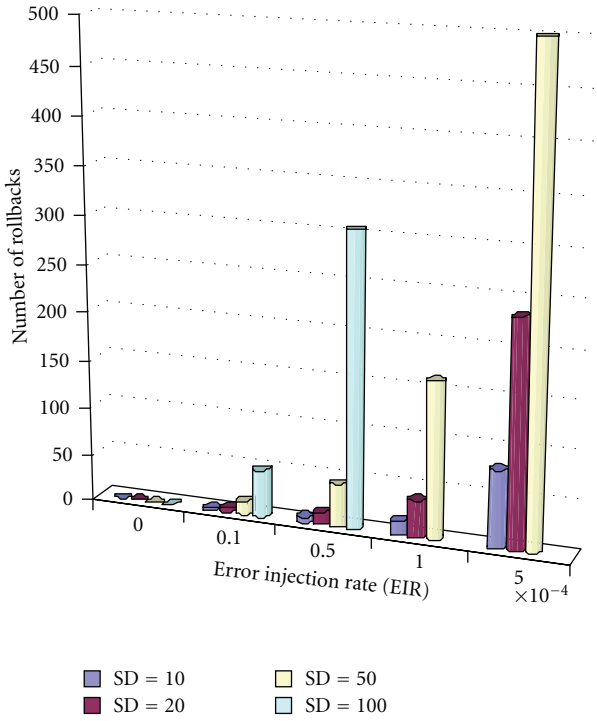


Figure 28: E.I.R effect on rollback (benckmarks of group III).

Implementation results show that area requirements are small and that speed performance degradation remains low under transient error injection, even for high error rates. For injection of simple errors, about 100% of the injected errors are detected and recovered for several experimental configurations. Similarly, for a double and triple bit error pattern injection, recovery capacity is about 60% and 78%. According to the results, the recovery is still possible, even for error patterns of up to 8 bits where recovery goes up to 36%. Therefore, the proposed approach can be effectively used in applications requiring a reasonable level of protection against transient errors at low HW cost.

In summary, the experiments presented in this section demonstrate beyond doubt that the proposed architecture is an interesting alternative approach to implement fault tolerance in processor architecture.

References

[1] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld et al., "IBM experiments in soft fails in computer electronics," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 3–16, 1996.

[2] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–315, 2005.

[3] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 389–398, June 2002.

[4] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri, "From a federated to an integrated architecture for dependable embedded real-time systems," Tech. Rep. 22, TU, Vienna, Austria, 2003.

[5] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[6] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[7] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger, "Tolerating transient faults in MARS," in *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS '90)*, pp. 466–473, June 1990.

[8] J. Gaisler, "Preparations for next-generation SPARC processor," in *Proceedings of the Workshop on Spacecraft Data Systems*, May 2003.

[9] T. J. Slegel, "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, 1997.

[10] D. McEvoy, "The architecture of tandem's nonstop system," in *Proceedings of the Association for Computing Machinery Conference (ACM '81)*, p. 245, 1981.

[11] D. Ernst, S. Das, S. Lee et al., "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[12] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '99)*, pp. 196–207, November 1999.

[13] P. Chevochot and I. Puaut, "Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies," in *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '99)*, pp. 356–363, 1999.

[14] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Reliability-aware co-synthesis for embedded systems," in *Proceedings of the 15th IEEE International Conference on Applications-Specific Systems, Architectures and Processors*, pp. 41–50, September 2004.

[15] J. Xu and B. Randell, "Roll-forward error recovery in embedded real-time systems," in *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS '96)*, pp. 414–421, June 1996.

[16] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, no. 1, pp. 83–102, 2001.

[17] Y. Zhang and K. Chakrabarty, "A unified approach for fault tolerance and dynamic power management in fixed-priority

real-time embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 1, pp. 111–125, 2006.

[18] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 113–125, 2003.

[19] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS '99)*, pp. 84–91, June 1999.

[20] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pp. 25–36, June 2000.

[21] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 87–98, May 2002.

[22] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Proceedings of the30th Annual International Symposium on Computer Architecture*, pp. 98–109, June 2003.

[23] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault-tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 80–107, 1996.

[24] A. Shye, T. Moseley, V. J. Reddi, J. Blomstedt, and D. A. Connors, "Using process-level redundancy to exploit multiple cores for transient fault tolerance," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pp. 297–306, June 2007.

[25] B. P. Dave and N. K. Jha, "COFTA: hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance," *IEEE Transactions on Computers*, vol. 48, no. 4, pp. 417–441, 1999.

[26] M. Y. Hsiao, "A class of optimal minimum oddweight-column SEC-DED codes," *IBM Journal of Research and Development*, vol. 25, no. 5, 1970.

[27] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: a state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, 1984.

[28] A. Ramazani, M. Amin, F. Monteiro, C. Diou, and A. Dandache, "A fault tolerant journalized stack processor architecture," in *Proceedings of the15th IEEE International On-Line Testing Symposium (IOLTS '09)*, pp. 201–202, June 2009.

[29] M. Amin, F. Monteiro, C. Diou, A. Ramazani, and A. Dandache, "A HW/SW mixed mechanism to improv. The dependability of a stack processor," in *Proceedings of the 16th IEEE International Conference on Electronics, Circuits and Systems (ICECS '09)*, pp. 976–979, December 2009.

[30] P. H. J. Koopman, *Stack Computers: The New Wave*, Mountain View Press, La Honda, Calif, USA, 1989.

[31] D. B. Hunt and P. N. Marinos, "A general purpose cache-aided rollback error recovery (CARER) technique," in *Proceedings of the 17th Annual Symposium on Fault-Tolerant Computing*, pp. 170–175, 1987.

[32] N. S. Bowen and D. K. Pradhan, "Virtual checkpoints: architecture and performance," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 516–525, 1992.

[33] F. Liberato, R. Melhem, and D. Mossé, "Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems," *IEEE Transactions on Computers*, vol. 49, no. 9, pp. 906–914, 2000.

[34] D. K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall, New York, NY, USA, 1996.

[35] J. Arlat, A. Costes, Y. Crouzet, J. C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 913–923, 1993.

[36] J. A. Clark and D. K. Pradhan, "Fault injection: a method for validating computer-system dependability," *Computer*, vol. 28, no. 6, pp. 47–56, 1995.

[37] W. T. Chang, S. Ha, and E. A. Lee, "Heterogeneous simulation—mixing discrete-event models with dataflow," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 15, no. 1-2, pp. 127–144, 1997.

[38] P. Vanhauwaert, *Fault-injection based dependability analysis in a FPGA-based environment*, Ph.D. thesis, Institut National Polytechnique de Grenoble, 2008.

[39] M. Väyrynen, V. Singh, and E. Larsson, "Fault-tolerant average execution time optimization for general-purpose multiprocessor system-on-chips," in *Proceedings of the Conference on Design Automationand Test in Europe*, pp. 484–4899, April 2009.

[40] http://www.gaisler.com/doc/leon3ft-rtax.pdf.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 591983, 28 pages
doi:10.1155/2011/591983

Research Article

Static Scheduling of Periodic Hardware Tasks with Precedence and Deadline Constraints on Reconfigurable Hardware Devices

Ikbel Belaid,[1] Fabrice Muller,[1] and Maher Benjemaa[2]

[1] *LEAT-CNRS, University of Nice Sophia Antipolis, 06560 Valbonne, France*
[2] *Research Unit ReDCAD, National Engineering School of Sfax, University of Sfax, 3038 Sfax, Tunisia*

Correspondence should be addressed to Ikbel Belaid, ikbel.belaid@unice.fr

Received 27 August 2010; Revised 12 January 2011; Accepted 10 February 2011

Academic Editor: Michael Hübner

Task graph scheduling for reconfigurable hardware devices can be defined as finding a schedule for a set of periodic tasks with precedence, dependence, and deadline constraints as well as their optimal allocations on the available heterogeneous hardware resources. This paper proposes a new methodology comprising three main stages. Using these three main stages, dynamic partial reconfiguration and mixed integer programming, pipelined scheduling and efficient placement are achieved and enable parallel computing of the task graph on the reconfigurable devices by optimizing placement/scheduling quality. Experiments on an application of heterogeneous hardware tasks demonstrate an improvement of resource utilization of 12.45% of the available reconfigurable resources corresponding to a resource gain of 17.3% compared to a static design. The configuration overhead is reduced to 2% of the total running time. Due to pipelined scheduling, the task graph spanning is minimized by 4% compared to sequential execution of the graph.

1. Introduction

An important trend in real-time applications implemented in reconfigurable computing systems consists in using reconfigurable hardware devices to increase performances and to guarantee temporal constraints. These reconfigurable devices provide a high density of heterogeneous resources in order to satisfy application requirements and especially to enable parallel computing. Furthermore, the devices employ the pertinent concept of run-time partial reconfiguration which allows reconfiguration of a portion of available resources without interrupting the remainder parts running in the same device. Consequently, the concept increases resource utilization and application performance.

Periodic partially ordered activities represent the major computational demand in real-time systems such as real-time control and digital signal processing. This category of repetitive computation is described by directed acyclic graphs (DAGs). Implementation of these DAGs in reconfigurable hardware devices consists in scheduling tasks to a limited number of nonidentical units shaped on the area of reconfigurable resources, while respecting the four constraints described as follows. (1) The periodicity constraint: each task is repeated periodically according to its ready times in the graph. Thus, if task A has a period P_A, then for all $i \in \mathbb{N}$, $(r_{A_{i+1}} - r_{A_i}) = P_A$, where A_i and A_{i+1} are the ith and the $(i+1)$th repetitions of task A, and r_{A_i} and $r_{A_{i+1}}$ are their start times. (2) The precedence constraint: to maintain the rightness of task precedences, in each iteration, a task can be executed only if all its predecessors in the graph have finished their executions. Therefore, each task A must start execution after the completion of executions of its predecessors defined by the subset Π_A, thus for all $i \in \mathbb{N}$, $s_{A_i} \geq s_{B_i} + C_B$, for all $B \in \Pi_A$, where s_{A_i}, s_{B_i} are the start times of task A and task B, respectively, during their ith iteration, and C_B is the execution time of task B. (3) The dependence constraint: the execution of each task in DAG is launched when all the data resulting from all its predecessors are available. This constraint guides the choice of task periods as detailed in Section 3. (4) The deadline constraint: as this paper focuses on hard real-time systems, each task in the DAG must finish its execution before its hard deadline. Thus, within iteration i, if task A has an execution time C_A and an absolute deadline d_{A_i}, then $s_{A_i} + C_A \leq d_{A_i}$.

Figure 1 illustrates an example of the targeting task graph. As can be seen in Figure 1, the tasks are repeated according to their fixed periods. Each task with precedence link launches its execution only when its predecessors achieve their executions and only when it is required. For example, the third iterations of T_2 and T_3 of periods 8 do not need a third execution of their predecessor T_1 as it is less repetitive than T_2 and T_3 (period of T_1 is equal to 12). At each repetition, to enable the task execution, the dotted lines ensure the data transfer between interdependent tasks. The issue of data dependence is detailed later in the paper. Finally, at each iteration, the real-time tasks must respect their hard deadlines.

As shown in Figure 2, this paper proposes a new methodology comprising three main stages to achieve the scheduling of these DAGs with the predefined constraints on reconfigurable devices.

Task Clustering. This stage is technology dependent. It targets the partitioning of tasks requiring the same types of resources into the same cluster.

Mapping/Scheduling of Tasks in Clusters. This stage starts by performing spatial and temporal analyses mentioned in Figure 2 by DAG validity, Ready Times, and a set of heuristics. Subsequently, based on a predefined preemption model, it deals with simultaneous resolution of mapping tasks to the obtained clusters and global scheduling of tasks in clusters respecting the periodicity, precedence, dependence, and deadline constraints. This stage aims at optimizing scheduling quality.

Cluster Placement on the Reconfigurable Device. This stage is also technology dependent. It involves searching for the most suitable physical location partitioned on the reconfigurable device for each cluster obtained at the second stage. This stage aims at optimizing placement quality.

The resolution of these three stages results in static scheduling of tasks in the DAGs into a limited number of reconfigurable units partitioned on the device, respecting the periodicity, precedence, dependence, and deadline constraints. This is a fundamental problem in parallel computation, equivalent to determining static multiprocessor scheduling for DAGs in a software context. As is well known, static multiprocessor task graph scheduling is a combinatorial optimization problem, and it is formulated in this paper through mixed integer programming and solved by means of powerful solvers.

The paper details the spatial and temporal analyses required to check scheduling task graph feasibility and aims at determining the optimal solution in terms of schedule length, waiting time, parallel efficiency, resource efficiency, and configuration overhead. Schedulability analysis is not the focus of the present paper. However, before dealing with DAG scheduling on reconfigurable device, the rightness of the precedences and dependences between tasks within the graph and the accuracy of real-time functioning are analyzed, and a set of heuristics are performed to provide the number of reconfigurable physical units needed to ensure the existence of valid DAG scheduling. The analyses are expressed by some constraints to ensure the validity of the chosen task graph.

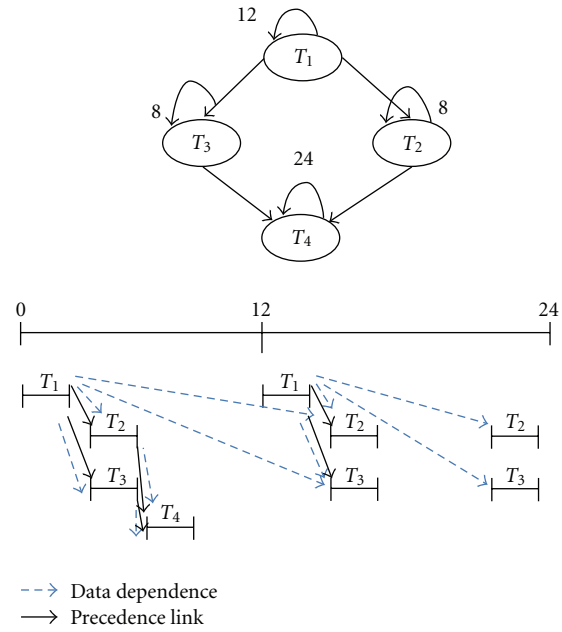


Figure 1: The targeted acyclic task graph.

The paper is organized as follows: Section 2 presents related works of the DAG scheduling problem. Section 3 details the methodology we propose to achieve the placement and scheduling of DAGs on reconfigurable devices. Section 4 describes an illustration of our proposed methodology on a given DAG and evaluates the obtained enhancements by metric measuring of placement and scheduling quality. The conclusion and future works are presented in Section 5.

2. Related Works

Static multiprocessor scheduling techniques using task graphs have matured over the last years, and many powerful scheduling strategies have emerged. As this problem is known to be NP-hard [1], the main research efforts in this area focus on heuristic methods and few of them propose analytic resolutions. We have studied static and dynamic multiprocessor scheduling using DAGs in both the software and hardware contexts.

In [2], Clemente et al. implement a static hardware scheduler employing efficient techniques which greatly reduce reconfiguration latencies and schedule length. Taking into account that configuration latency drastically reduces the efficiency of hardware multitasking systems, they introduce a new hardware scheduler communicating directly with the reconfigurable units and using optimization techniques: prefetch, reuse and replace while guaranteeing the precedence constraints. The prefetch technique manages in advance the reconfigurations and replacements required to improve task reuse. Reference [2] presents three algorithms

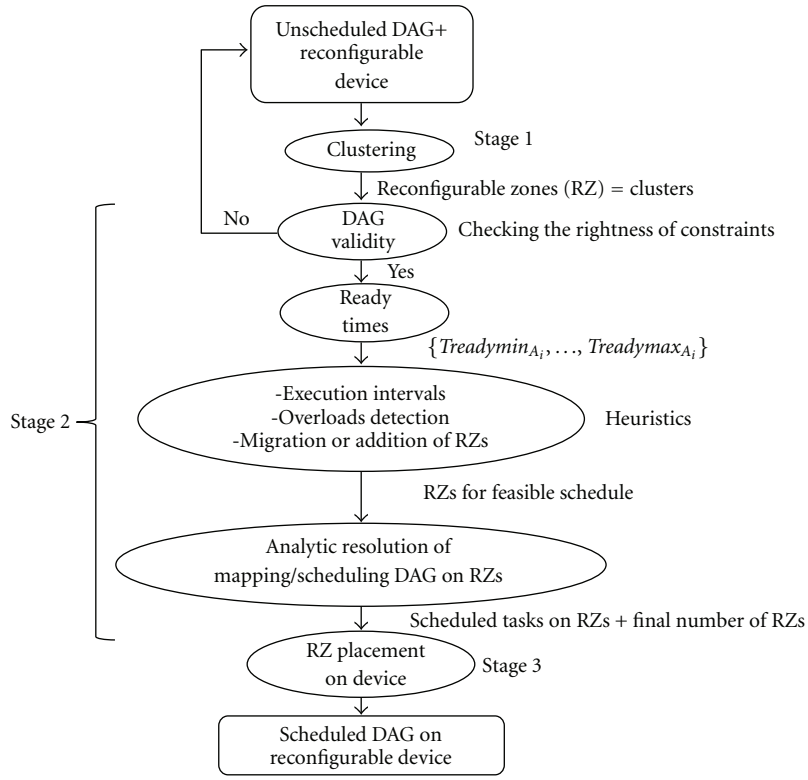


Figure 2: The proposed methodology.

for the replace technique, that is, the *Least Recently Used*, *Longest Forward Distance*, and *Look Forward + Critical* (*LF + C*) algorithms. The paper focuses especially on *LF + C* to schedule graph tasks on several reconfigurable units. In order to maximize the task reuse that reduces reconfiguration latency, *LF + C* classifies tasks from the most critical ones in terms of required reconfiguration to the least critical ones and tries to replace the latter tasks, so that their reconfiguration does not generate any overhead during task graph execution. This advantage is ensured by the prefetch technique which reconfigures a given task during execution of its predecessors.

Static multiprocessor scheduling is a very difficult problem, but genetic algorithms have successfully been applied to the search for acceptable solutions. In [3], the authors investigate scheduling for cyclic task graphs using genetic algorithms by transforming the cyclic graph into several alternate DAGs. To create an efficient schedule, the paper considers both the intracycle dependencies and the dependencies across different cycles. After unfolding the cyclic task graph for two cycles by incorporating the intercycle dependencies, the paper presents an algorithm investigating all the subgraphs extracted from a two-cycle task graph. Based on measurements such as the height and the width of the task graph, connection degree, degree of simultaneousness, and independent parts in the graph, the method evaluates the resulting subgraphs to select the configuration that best suits a chosen application and the available hardware configuration. Suitable allocation to processors is obtained by the *earliest start time* heuristic where tasks are assigned to the processor that offers the earliest start time. The employed genetic algorithm tries to optimize the schedule length which is expressed by the finishing time on all processors.

In [4], Abdeddaim et al. describe a method based on a timed automaton model for solving the problem of scheduling partially ordered tasks on parallel identical machines. The proposed method formulates for each task in the graph a 3-state automaton consisting of the waiting, active, and finish states. Therefore, by searching the tasks related by a partial order in the graph, the possible disjoint chains in the graph are extracted. The automaton of every chain is constructed using the individual task-specific automaton. The global automaton is then composed by the chain-specific automata and takes care that the transitions do not violate the precedence, and resource constraints. Thus, optimal scheduling consists in finding the shortest path in the timed automaton. The proposed methodology is also extended to include two additional features in the task-specific automaton which are the release and the deadline times.

Integer linear programming (ILP) formulation is exploited in some works of static multiprocessor scheduling using task graphs. The authors of paper [5] propose an exact ILP formulation to perform task graph scheduling on dynamically and partially reconfigurable architectures and that minimizes schedule length. In addition, this work proposes a dynamic reconfiguration-aware heuristic scheduler called *NAPOLEON*, which adopts an ALAP (as late as possible)

order of tasks and exploits configuration pre-fetching, module reuse, and antifragmentation techniques. Both methods are extended to the Hw/Sw codesign. The ILP formulation is based on nonpreemptive tasks, allows the execution of tasks on software processors or on the FPGA, and respects the FPGA physical constraints as well as the precedence and temporal constraints in the graph. Both methods provide a solution for the complete scheduling of the DAG and determine for each task its Sw or Hw execution unit, its time of reconfiguration start, its position on FPGA, and its execution starting time.

Another exact ILP formulation for performing task mapping and scheduling on multicore architectures is presented in [6]. The technique of these authors incorporates loop level task partitioning, task transformations by using loop fusion and loop splitting, and it aims at reducing system execution time. The paper focuses on an ILP-based approach for task partitioning, task mapping, and pipelined scheduling while taking data communication between processors into consideration for DSP applications on the multicore platform. The authors in [6] divide the problem into two parts. The first assigns and schedules tasks on processors by including the task merging on the same batch and the replication of batches to several processors. The second step conducts a mapping of data to memory architecture by minimizing memory access latencies.

In [7], Sandnes and Sinnen consider the scheduling of iterative computing that can be represented by cyclic task graphs. In order to avoid costly classic graph unfolding and to shorten the makespan during scheduling, the authors propose a new strategy for transforming cyclic task graphs into acyclic task graphs; an efficient scheduling from the literature named *Critical Path/Most Immediate Successor First* (*CP-MISF*), proposed by Kasahara and Narita in 1985, is then applied to the transformed graph. The strategy is based on a decyclification step involving three parts: (1) a decyclification algorithm for transforming the cyclic graph into an acyclic graph based on a given start node and depth first search (DFS) strategy, (2) by assuming that the critical path in the graph is a good estimator for its schedule length, it searches the start node that yields the shortest critical path in the transformed graphs, and (3) quantifying the acyclic graph quality in terms of makespan. In addition, based on an adjacency matrix representing the graph dependencies and simplifying the unfolding formulation, the paper presents a new intuitive graph unfolding formulation which decomposes the adjacency matrix into two matrices, one for intraiteration dependences and another for interiteration dependences. The unfolded graph is then scheduled using a genetic algorithm approach.

In [1], Djordjević and Tošić propose a new compile-time single-pass scheduling technique applied for task graphs onto fully connected multiprocessor architectures called *chaining* and which takes into account communication delays. The proposed technique consists in a generalized list scheduling with no preconditions concerning the order in which tasks are selected for scheduling. The main idea is to build an heuristic providing a trade-off between maximizing parallelism on processors, minimizing communication overheads, and minimizing overall execution time of the task graph. *Chaining* technique uses nonpreemptive tasks and constructs a scheduled task graph incrementally by scheduling one task at each step. The intermediate partially scheduled task graphs are obtained by selecting a nonscheduled task at each step and by placing it on the most appropriate precedence edge. The policy of selection of tasks to be scheduled is based on a *Task Selection First* heuristic, and the selection of the most suitable valid edge where the task will be placed is guided by the critical path and edge width criteria. The tasks encompassed within the same chain are scheduled on the same processor.

In [8], the authors aim at improving the performance of hardware tasks on the FPGA. Intertask communication and data dependences between tasks are analyzed in order to reduce configuration overhead, to minimize communication latency, and to shorten the overall execution of tasks. The work exploits the proposed works in reconfigurable computing and addressing resource efficiency to present three algorithms. *Reduced Data Movement Scheduling* (*RDMS*) is the most efficient dynamic algorithm for reducing configuration and communication overheads and provides the optimal performance for scheduling tasks in DAGs on the FPGA. *RDMS* uses the total reconfiguration of the FPGA and tries to minimize the number of reconfigurations by grouping communicating tasks in the same configuration. By conducting a width search, *RDMS* schedules tasks while respecting their data dependences. *RDMS* is based on dynamic programming algorithm and ensures that each configuration includes the combination of tasks that exploits the hardware resources to the maximum and that encompasses the highest possible number of task dependences.

In [9], Fekete et al. consider the optimal placement of hardware tasks in space and in time on the FPGA. Tasks are presented as three-dimensional boxes in space and time. The authors integrate intertask communication expressing data dependence and use a graph-theoretical characterization of the feasible packing determined by means of a decision of an orthogonal packing problem with precedence constraints. By searching the transitive orientations and by performing projections, the authors of paper [9] transform the 3D boxes representing tasks into 3×1D arrangements and then verify whether the three obtained arrangements referred to as packing classes satisfy the conditions of the feasible packing and determine the optimal spatial and temporal packing. This work enhances the makespan of the graph and optimizes the used reconfigurable space on the FPGA.

The major contribution in [10] is the development of a multitasking microarchitecture to perform a dynamic task scheduling algorithm on reconfigurable hardware for nondeterministic applications with intertask dependences which are not known until runtime. The task system is modeled as a modified directed acyclic graph which contains directed data edges and directed control edges labeled with scalar values indicating the probability of occurrence of the corresponding sink task in multiple task graph iterations [10]. Based on dynamic priority assignment for nonpreemptive tasks, the dynamic scheduler assigns each task to a software or hardware processing element, schedules

the contexts (bitstreams) and the data, and the fetch and the prefetch reconfigurations, and activates task execution. In order to minimize reconfiguration overhead, the dynamic scheduler uses the configuration prefetching technique to prefetch the task bitstream ahead of time or exploits the previous context configured in the logic cell. In addition, it aims to minimize application execution time by employing a local optimization technique.

In [11], Kohler defines a new heuristic to schedule DAGs on a system of independent identical processors. The author describes a simple critical path priority method which is shown to be optimal or near optimal in the most randomly generated computation graphs compared to the Branch and Bound method. This heuristic aims to minimize the finishing time of the computation graph. Critical path scheduling is based on a list (L) containing permutation of the tasks. Any time a processor is idle, it instantaneously scans the list L from the beginning and begins to execute the first free task which may validly be executed because all its predecessors have been completed [11]. The construction of the list is based on the critical path of tasks which is defined by the longest path from a given task to a terminal node. The paper also presents the exact Branch and Bound method used to obtain optimal scheduling, and the results obtained are compared to the critical path heuristic to prove the high quality of the latter method.

Table 1 provides a summary of the optimization parameters and employed techniques described in the cited works.

The major common drawback of most described techniques is that they do not address real-time constraints. Further-more, as shown in Table 1, most of them seek to optimize the makespan of the graph and neglect reconfiguration overhead and resource inefficiency or do not optimize the three parameters simultaneously. The works described in [8, 9] that conduct scheduling of DAGs on FPGA devices are based on successive total configurations of the device. Their resource efficiency consists only in the packing the maximum of tasks in the DAG on the FPGA in order to efficiently exploit the reconfigurable resources as well as to perform the minimum of total configurations. These works therefore do not consider the internal fragmentation caused by task placement on the FPGA which represents resource efficiency in our work.

In the context of hardware task scheduling, in the proposed works, the placement of scheduled tasks is not considered. Either the placement of the task is allowed in whatever position in the device (in this case, they do not take into account device heterogeneity) or the task is fixed to a unique reconfigurable unit which will reduce application flexibility. Contrary to these works, our strategy may be generalized for all types of devices, that is, both homogeneous and heterogeneous devices; the placement problem is considered an important stage, that is, highly interlinked with the scheduling of task graphs on the reconfigurable device. With our strategy, the task may be executed on several reconfigurable units according to its resources and according to the analyses conducted during the clustering stage.

Moreover, some of the described works do not exploit the relevant concept of run-time partial reconfiguration afforded by recent reconfigurable devices and employ the total configuration of FPGAs.

Based on these observations, our challenge is to utilize the benefits of the run-time partial reconfiguration concept for recent heterogeneous devices. The concept opens up the possibility of developing a hardware multitasking system by dividing the reconfigurable area into smaller reconfigurable units and by customizing them as required by the running application.

Considering multitasking, scheduling of task graphs on reconfigurable hardware devices is similar to heterogeneous multiprocessor scheduling in the software context. With full knowledge of the characteristics of DAG tasks and technology features, our methodology targets constrained applications and endeavors to provide pipelined scheduling in multi-reconfigurable-unit system while optimizing schedule length, waiting time, parallel efficiency, resource efficiency, and configuration overhead.

3. Proposed Methodology for Placement and Scheduling of Dags on Reconfigurable Devices

Our methodology can be viewed as two separate subproblems: (i) the mapping and scheduling of hardware tasks on predefined clusters by satisfying periodicity, precedence, dependence, and deadline constraints and (ii) the placement of obtained clusters on reconfigurable device taking into account its heterogeneity. Our resource and task management is essentially based on features of hardware tasks and reconfigurable hardware devices. The most recent Xilinx's Virtex FPGA was used as a reference for the reconfigurable hardware device to perform the placement and scheduling of the DAGs. Virtex SRAM-based FPGAs are characterized by a column-based architecture, a high density of heterogeneous resources, and several parallel configuration ports functioning at a high speed.

3.1. Terminology and Definitions. Throughout the paper, NT refers to the number of tasks in the graph, NZ the number of clusters, and NP the number of resource types in the chosen technology. The directed acyclic task graph is denoted by the pair (N, E). N is the set of nodes representing tasks in the DAG, and E is the set of edges linking the dependent tasks, $E \subseteq N \times N$.

As shown in Figure 3, on each edge, the outgoing value $(x_{A,B})$ from the source node A to the sink node B depicts the amount of data that A must produce at each repetition for B. The incoming value $(y_{B,A})$ represents the amount of data that must be consumed by the sink node B at each execution iteration after completion of the repetition of its predecessor A.

Each task in the graph has three models as follows.

3.1.1. Functional Model. Each hardware task (A) is represented by a set of parameters fixed at compile time and which are kept static throughout the DAG execution. A is characterized by its worst-case execution time (C_A), its

Table 1: Optimization parameters and techniques for scheduling works.

References	Makespan/Speedup/ Parallel efficiency	Resource efficiency	Configuration overhead	Techniques
[2]	x		x	Prefetch/replace/reuse of reconfigurations
[3]	x			Graph unfolding/genetic algorithm/*earliest start time* heuristic
[4]	x			Timed automaton model/shortest path
[5]	x		x	(1) ILP/reuse (2) *NAPOLEAN*/ALAP/prefetch/reuse/ antifragmentation
[6]	x			ILP/loop level task partitioning/task transformation (loop fusion + loop splitting)/task mapping and scheduling (task merging on batches + batch replication)/data mapping
[7]	x			(1) Graph decyclification (DFS + shortest critical path)/*CP-MISF* (2) Graph unfolding/genetic algorithm
[1]	x			*Chaining*/*task selection first* heuristic/critical path/edge width/communication latencies
[8]	x	x	x	Dynamic programming algorithm/FPGA total configuration
[9]	x	x		Orthogonal packing problem/packing classes
[10]	x		x	Prefetch/local optimization/reuse
[11]	x			Critical path heuristic/Branch and Bound

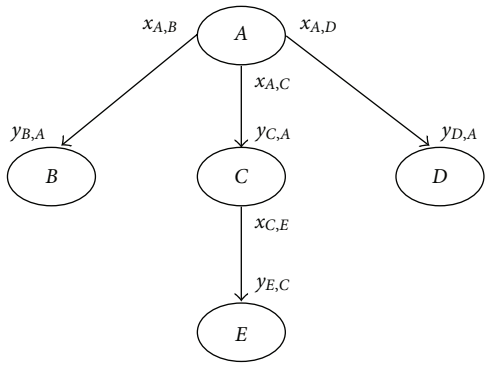


Figure 3: Directed acyclic graph.

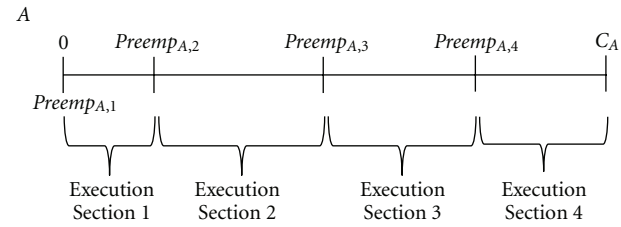


Figure 4: Predefined preemption points in Task A.

period (P_A), which is equal to its relative deadline (D_A), and a set of preemption points ($Preemp_{A,l}$). The preemption points are instants of the time taken throughout the worst-case execution time as shown in Figure 4. The number of preemption points of task A is denoted by $NbrPreemp_A$. This number also includes the first point of execution of the task. The set of preemption points is determined by the designer according to the known states in the behavioral model and according to possible data dependences between these states. The predefinition of preemption points gives rise to the execution sections within the hardware task. Our methodology is based on preemptive modeling to create a reactive system, to increase flexibility towards application needs, and consequently to enhance the respect of real-time constraints.

3.1.2. Behavioral Model. This includes the finite state machine controlling each task and which handles a set of registers or a small memory bank useful for context switching during preemption. The latency required to preempt and to resume the execution of tasks is disregarded as in the worst case; the time to access the system bus and memory is negligible. With the preemptive model, we do not use the classical method of readback and load modified bitstream since latency with this method is significant; it complicates preemption and requires a large space memory. With the classical method, a new readback bitstream must be saved at each preemption. With our model, the number of preemptions within tasks is limited by specifying the

possible preemption points according to task states outside of which preemption is not allowed. Thus, we resort to saving/loading the current state of the finite state machine with an acceptable amount of data by maintaining the same bitstream for each task within a given reconfigurable unit when the task needs to be preempted or resumed. Preemption points of hardware tasks are set in a way that reduces the data dependences that could exist between two states. In fact, maintaining a preemption point between two states processing the same data must be avoided because these data need to be saved into an external memory which might increase the preemption overhead at runtime. Otherwise, it is recommended to insert a preemption point when the task is in a blocked state, waiting to receive an external resource to enable the ready tasks to be executed in the reconfigurable unit. In the finite state machine, the longest execution time between two states must be considered in order to deduct the worst-case execution time.

3.1.3. RB-Model. At the physical level, tasks are presented as a set of reconfigurable resources called reconfigurable blocs (RB). RBs correspond to the physical resources in the reconfigurable hardware device required to achieve execution of the hardware task, and they define the RB model of the task as expressed by (1). Determination of the RB-model of hardware tasks is well detailed in our work described in [12]. The number of RB types is equal to the number of resource types in the chosen technology. The RBs are the smallest reconfigurable units in the hardware device. They are determined according to the available reconfigurable resources in the device, and they closely match its reconfiguration granularity. Each type of RB is characterized by a specified cost, $RBCost_k$, defined according to its frequency in the device, its power consumption, and the importance of its functionality,

$$A\_\mathrm{RB} = \{\alpha_{A,k}\mathrm{RB}_k\}, \quad \alpha_{A,k} \in \mathbb{N}, \quad 1 \le A \le \mathrm{NT}, \quad 1 \le k \le \mathrm{NP}. \tag{1}$$

The reconfigurable device is also characterized by its RB model as shown in our work described in [12] to enable the placement of hardware task clusters at a later stage.

The three main stages of the methodology used for static scheduling of DAGs on multi-reconfigurable-unit system with predefined constraints are described below.

3.2. Hardware Task Clustering. This stage comprises two steps that are performed consecutively: (i) reconfigurable zone type search and (ii) cost D computing. Bearing in mind that the concept of run-time partial reconfiguration had to be used, our main objective at this first stage was to partition tasks constituting the graph into cluster types determined according to their required RB types in order to enhance resource utilization.

3.2.1. Reconfigurable Zones Types Search. This step takes as input the RB model of each task in the DAG, and by performing Algorithm 1 of the worst-case complexity $O(\mathrm{NT}^2 * \mathrm{NP})$, it groups tasks sharing the same types of RBs under the same type of cluster by taking the maximum number of RBs between these tasks. With our methodology, the obtained types of clusters are denoted as reconfigurable zones (RZs). The upper bound of the possible RZs is NT. Thus, RZs are virtual units customized by Algorithm 1 to model the classes of hardware tasks in terms of RB types. RZs separate hardware tasks from their execution units on the reconfigurable device. In the last stage of our proposed methodology, RZs will be placed on their suitable reconfigurable units respecting the heterogeneity of the device and optimizing resource efficiency as well as configuration overhead. After the completion of Algorithm 1, each RZ is represented by its RB model as expressed by

$$\mathrm{RZ}_j\_\mathrm{RB} = \left\{\beta_{j,k}\mathrm{RB}_k\right\}, \quad \beta_{j,k} \in \mathbb{N}, \quad 1 \le j \le \mathrm{NZ}, \quad 1 \le k \le \mathrm{NP}. \tag{2}$$

Algorithm 1 processes the tasks of the DAG as follows. It scans the RB model of each hardware task and checks whether an already inserted type of RZ that closely matches the required types of RBs in the task exists in the RZ types list, *list-RZ* (line 6). Should this be the case, Algorithm 1 updates the number of RBs within this type of RZ by the maximum between the number of RBs in the task and that in the RZ (line 9).

If the required types of RBs in the task do not match any type of RZ included in the *list-RZ*, the algorithm of the search of RZ types decides on the creation of a new type of RZ as required by the task (lines 12, 13) and inserts it in *list-RZ* (line 14).

Figure 5 is an example of the execution of Algorithm 1 for the RZ types search for DAG comprising five tasks. Figure 5 illustrates the search for RZ types resulting from five tasks in a technology including four types of RBs (RB_1, RB_2, RB_3 and RB_4). A and C are grouped in the same type of RZ (RZ_1) as both need RB_1 and RB_2, and the number of each RB type within RZ_1 is adjusted by the maximum number of RBs between A and C. Similarly, RZ_2 is created by B and D, and E defines the third type of RZ (RZ_3).

After searching for the set of RZs, the configuration overhead for each obtained RZ is computed and denoted by $Config_j$. $Config_j$ corresponds to the configuration overhead to fit RZ_j in the target technology. This configuration overhead is computed by the floorplanning of each RZ_j on the chosen device and by conducting the whole partial reconfiguration flowup to the creation of the partial bitstream. $Config_j$ is determined by (3) according to configuration frequency and configuration port,

$$Config_j = \frac{\text{size of bit stream}}{(\text{Configuration frequency} \times \text{configuration port width})}. \tag{3}$$

3.2.2. Cost D Computing. This step commences by computing cost D between tasks and each RZ type resulting from the

(1) $RZ\text{-}type = 0$ // RZ types
(2) $List\text{-}RZ$ // list of RZ types
(3) n // natural
(4) **for all** tasks A **do**
(5) // $A\_\mathrm{RB} = \alpha_{A,k}\mathrm{RB}_k$
(6) **if** $((RZ\text{-}type \neq 0)$ **and** $(\exists n, 1 \leq n \leq RZ\text{-}type)/\forall k((\alpha_{A,k} \neq 0$ **and** $\beta_{n,k} \neq 0)$ **or** $(\alpha_{A,k} = 0$ **and** $\beta_{n,k} = 0)))$ **then**
(7) // this test checks whether the task matches with an RZ type that already exists in $list\text{-}RZ$
(8) **for all** k **do**
(9) $\beta_{n,k} = \max(\alpha_{A,k}, \beta_{n,k})$ // update RB number of RZ_n
(10) **end for**
(11) **else**
(12) Increment $RZ\text{-}type$
(13) $\mathrm{RZ}_{RZ\text{-}type} = Create\ new\ \mathrm{RZ}(\alpha_{A,k})$ // new type of RZ, $\mathrm{RZ}_{RZ\text{-}type} = \{\alpha_{A,k}\mathrm{RB}_k\}$
(14) Insert $(list\text{-}RZ, \mathrm{RZ}_{RZ\text{-}type})$
(15) **end if**
(16) **end for**

Algorithm 1: RZ types search or hardware task classes search.

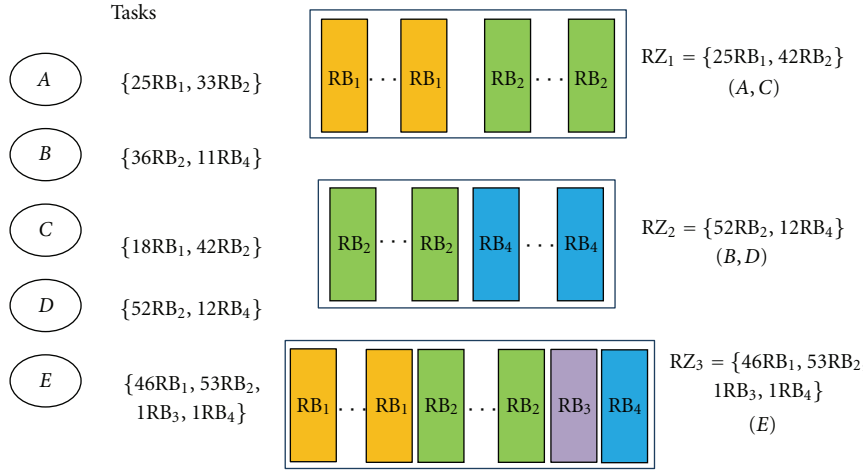


Figure 5: Example of the search for RZ types.

first step. Cost D represents the differences in RBs between tasks and RZs; consequently, it expresses resource wastage when a task is mapped to an RZ. Based on RB models of task A and RZ RZ_j, cost D is computed according to two cases as follows. Firstly, we define by

$$d_{A,j,k} = \alpha_{A,k} - \beta_{j,k}, \quad 1 \leq A \leq \mathrm{NT}, \quad 1 \leq j \leq \mathrm{NZ}, \quad 1 \leq k \leq \mathrm{NP}. \tag{4}$$

Case 1. For all k, $d_{A,j,k} \leq 0$, RZ_j contains a sufficient number of each type of RB (RB_k) required by A, and cost D is equal to the sum of the differences in the numbers of each RB type between A and RZ_j weighted by $RBCost_k$ as expressed in

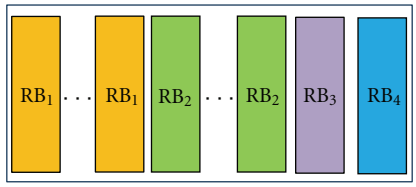
$$D\left(A, \mathrm{RZ}_j\right) = \sum_{1 \leq k \leq \mathrm{NP}} RBCost_k \times \left| d_{A,j,k} \right|. \tag{5}$$

Case 2. There exists k, $d_{A,j,k} > 0$, the number of RBs required by A exceeds the number of RBs in RZ_j or A needs RB_k which is not included in RZ_j. In this case, cost D is infinite,

$$D\left(A, \mathrm{RZ}_j\right) = \infty. \tag{6}$$

Cost D is exploited during the stage of task mapping and scheduling on RZs and the RZ placement stage to optimize the utilization of costly resources to the device. The execution of a given task in an RZ is allowed only when the cost D between them is finite. Figure 6 illustrates the computing of costs D between the five tasks and RZ_3 described in Figure 5.

3.3. Mapping and Scheduling of Hardware Tasks on RZs. It is well known that task mapping and scheduling are highly interdependent. The two issues therefore need to be addressed together for mapping and scheduling to be efficient. In order to analyze the scheduling of a given DAG



$RZ_3 = \{46RB_1, 53RB_2, 1RB_3, 1RB_4\}$ (E)

$RBCost_1 = 20$, $RBCost_2 = 80$, $RBCost_3 = 192$, $RBCost_4 = 340$

$D(A, RZ_3) = 20 \times |25\text{–}46| + 80 \times |33\text{–}53| + 192 \times |0\text{-}1| + 340 \times |0\text{-}1|$

$D(B, RZ_3) = \infty$ (Lack of $10RB_4$ in RZ_3 for B)

$D(C, RZ_3) = 20 \times |18\text{–}46| + 80 \times |42\text{–}53| + 192|0\text{-}1| + 340 \times |0\text{-}1|$

$D(D, RZ_3) = \infty$ (Lack of $11RB_4$ in RZ_3 for D)

$D(E, RZ_3) = 0$

Figure 6: Example of computing cost D with RZ_3.

of periodic tasks, it is sufficient to study its behavior for a time interval equal to the least common multiple of all the task periods, called the hyperperiod (HP). Consequently, the possible iterations of execution of each task A during the HP may be determined according to its period P_A, which is equal to $\lfloor HP/P_A \rfloor$. To resolve the subproblem of mapping and scheduling of hardware tasks on RZs, our methodology conducts three steps of spatial and temporal analyses. The first one checks the rightness of task precedences, dependences, and real-time functioning in the DAG by means of three essential constraints. Consequently, the DAG will be validated to launch the following analyses. The second analysis determines the lists of ready times of each task for each possible iteration during the hyperperiod. These lists take into account the periodicity, precedence, dependence, and deadline constraints. The third straightforward analysis takes as input the lists of ready times, searches at each iteration the possible execution intervals for each task, and therefore detects possible conflicts due to overlapping between execution intervals of parallel tasks on the same RZ. The third analysis pursues its processing to solve the detected overloads within RZs by performing either the migration of execution sections of some tasks respecting their predefined preemption points or by increasing the number of overloaded RZs.

3.3.1. Checking of Precedence, Dependence, and Real-Time Rightness in DAG. The first temporal analysis does not take into account spatial constraints and considers that there is an available RZ for each task. It also considers the periodicity of tasks. The main objective of this analysis is only to validate the correctness of task precedence and dependences and real-time constraints in the studied DAG. It is conducted by means of three essential constraints.

(a) Dependence Checking. More than precedence, we consider that the tasks related by an edge of precedence are also dependent. This means that the execution of a given task requires the data resulting from the execution of all its predecessors. This dependence is expressed by (7). Equation (7) focuses on periods and the amount of interchanged data between dependent tasks. It guarantees, when $P_A \geq P_B$ and $A \in \Pi_B$, that each data $(x_{A,B})$ produced by task A in its repetition A_i is consumed by its successor B during its iterations of execution B_i or $B_{j>i}$. When $P_A \leq P_B$ and $A \in \Pi_B$, at each iteration of execution of B, it ensures that there are the sufficient data $(y_{B,A})$ for B to be executed.

$$y_{B,A} \times \left\lfloor \frac{HP}{P_B} \right\rfloor = x_{A,B} \times \left\lfloor \frac{HP}{P_A} \right\rfloor, \quad \forall B, A \in \pi_B. \tag{7}$$

The previous equation eliminates the problem of data wastage and ensures that the data produced by all the iterations of each task are consumed by all the iterations of its successors. In this work, we focus on the case where $P_A \geq P_B$, for all $B, A \in \pi_B$.

(b) Precedence Checking. As we work in the case of $P_A \geq P_B$, for all $B, A \in \pi_B$, considering the periodicity constraint, this constraint claims that each iteration of execution A_i of a given task A may include only the iterations B_i or $\{B_{j<i}\}$ of its successors $\{B\}$ to ensure the correctness of the precedence constraint. During repetition A_i of a given task A, this constraint prohibits that B_i and $\{B_{j>i}\}$ repetitions of its successors $\{B\}$ coexist. In this way, during the HP, each ith execution of any task is preceded by the ith execution iterations of all its predecessors. To guarantee these rules, the following constraint expressed by (8) was developed to guide the selection of execution times and periods for tasks in the DAG:

$$\begin{aligned} &Tready_B + k \times P_B > Tready_A + (k-1) \times P_A, \\ &\forall B, A \in \pi_B, \quad k \in \mathbb{N}, \quad 1 \leq k \leq NbrIter_A. \end{aligned} \tag{8}$$

$NbrIter_A$ depicts the possible number of execution iterations of A during the HP. $Tready_A$ and $Tready_B$ are the ready times for tasks A and B without considering the spatial constraints. They are determined by searching the critical path corresponding to the task in the graph by using the ASAP (as soon as possible) technique. For each task B having a set of predecessors $\{A\}$, its ready time $Tready_B$ is computed as follows

$$\begin{aligned} Tready_B &= \max_{\{A \in \pi_B\}} \left(Tready_A + C_A \right), \\ Tready_B &= 0, \quad \text{if } \pi_B = \varnothing. \end{aligned} \tag{9}$$

(c) Real-Time Checking. Considering the periodicity constraint, this constraint analyses the respect of real-time constraints in the best case of spatial conditions in terms of number of RZs. By respecting the precedence and dependence constraints, (10) checks at each iteration whether each task may complete its execution before its strict absolute deadline. If the absolute deadline of the task for its last

iteration exceeds the HP, the deadline then turns into the HP. As we work in the case $P_A \geq P_B$, for all $A \in \pi_B$, the second expression of (10) satisfies the deadline constraint for the remaining repetitions of task B when its predecessors $\{A\}$ achieve all their execution iterations

$$\begin{aligned}
&\max\left(Tready_A + k \times P_A + C_A, Tready_B + k \times P_B\right) + C_B \\
&\quad \leq \min\left(Tready_B + (k+1) \times P_B, \mathrm{HP}\right), \quad \forall B, A \in \pi_B, \\
&\qquad k \in \mathbb{N}, \quad 0 \leq k \leq NbrIter_A - 1, \\
&Tready_B + k \times P_B + C_B \leq \min\left(Tready_B + (k+1) \times P_B, \mathrm{HP}\right) \\
&\forall B, k \in \mathbb{N}, \quad \max_{A \in \pi_B}(NbrIter_A) \leq k \leq NbrIter_B - 1.
\end{aligned} \tag{10}$$

Respect of the three previous constraints validates the selection of periods and execution times for periodic tasks in the graph for the scheduling with precedence, dependence constraints, and under strict real-time constraints on an unlimited number of reconfigurable units. Nevertheless, the following temporal and spatial analyses will extract the corresponding number of RZs that will satisfy these constraints. When the previous constraints are unreal, the DAG is considered invalid and consequently it will be rejected or the features of its tasks will be modified and evaluated again till it respects the constraints expressed by (7), (8), and (10).

Consequently, with our methodology, we depart from the temporal analysis to construct a suitable physical architecture allowing a feasible schedule. As a final step at this stage, after fixing the physical architecture of the multi-reconfigurable-unit system, analytic resolution of mapping/scheduling provides the optimal scheduling of DAGs on target technology.

3.3.2. Determination of Lists of Ready Times. The temporal analysis does not consider the physical constraints and searches all the ready times in each execution iteration for all tasks in the graph by respecting the precedence, dependence, periodicity, and real-time constraints. This analysis yields, by means of (11), the lists $\{Treadymin_{B_i}, \ldots, Treadymax_{B_i}\}$ during which task B may start execution, where B_i denotes the ith iteration of task B within the HP. Outside this list, task B might not respect the predefined constraints, which would lead to unfeasible scheduling.

For each task B, $Treadymin_{B_i}$ is the lower bound time to start task execution at its iteration i in order to respect its precedence, dependence, and periodicity constraints. $Treadymax_{B_i}$ is the upper bound time from which the task can no longer start execution if its strict deadline and the data dependency and precedence of its successors are to be respected. To compute the *Treadymin* of tasks, we start computing from the top of the DAG. For the *Treadymax* calculation, we start from the bottom of the DAG, and for both, we proceed using the breadth-first search strategy

$$\begin{aligned}
&Treadymin_{B_1} \\
&\quad = 0 \quad \text{if } \pi_B = \varnothing, \\
&Treadymin_{B_i} \\
&\quad = \max\left(\max_{A \in \pi_B}\left(Treadymin_{A_i} + C_A\right), rmin_{B_i}\right), \\
&Treadymax_{B_1} \\
&\quad = \min\left(\min_{A \in \varphi_B}\left(Treadymax_{A_1}\right), P_B\right) - C_B, \\
&\qquad \varphi_B \text{ is the set of successors of } B, \\
&Treadymax_{B_i/i>1} \\
&\quad = \min\left(\min_{A \in \varphi_B}\left(Treadymax_{A_i}\right), rmax_{B_{i+1}}, \mathrm{HP}\right) - C_B.
\end{aligned} \tag{11}$$

$rmin_{B_i}$ and $rmax_{B_i}$ are the start times of the ith repetition of B according to its first ready times ($Treadymin_{B_1}$ and $Treadymax_{B_1}$). Hence, they are determined by incrementing $Treadymin_{B_1}$ and $Treadymax_{B_1}$ by P_B. For example, if we have a task B with a period = 8, a hyperperiod HP = 24, and we consider $Treadymin_{B_1} = 3$ and $Treadymax_{B_1} = 4$, then $rmin_{B_1} = Treadymin_{B_1} = 3$, $rmin_{B_2} = 3 + 8 = 11$, $rmin_{B_3} = 11 + 8 = 19$, $rmax_{B_1} = Treadymax_{B_1} = 4$, $rmax_{B_2} = 4 + 8 = 12$, $rmax_{B_3} = 12 + 8 = 20$.

3.3.3. Determination of Task Execution Intervals and the Number of RZs. This temporal and spatial analysis considers the RZ types resulting from the task clustering stage and searches the possible parallelism between tasks to study execution conflicts on the RZs. When an overload is detected in some RZs, the analysis starts by solving this problem through a migration mechanism; if migration does not produce a solution, it increments the number of overloaded RZs as required. This analysis searches first, by means of Algorithm 2, the execution intervals of each task for each possible iteration during the HP, then, using Algorithm 3, it deals with the overlapping execution intervals on the RZs to search the possible overloads, and finally, it uses Algorithm 4 to try to solve the found overloads by the migration mechanism; when this latter mechanism fails, additional RZs are inserted in the architecture of the multi-reconfigurable-unit system to solve the persisting overloads.

(a) Search of Execution Intervals. This step uses the functional model of tasks and determines their execution intervals by means of Algorithm 2 of worst-case temporal complexity equal to $O(\mathrm{NT} * \mathrm{HP}^3)$. An execution interval for a given task at a given iteration is an interval during which the task can be executed while satisfying all the predefined constraints.

For each task A in the DAG, Algorithm 2 produces the set of its possible execution intervals expressed by

(1) A, B // Tasks
(2) i // Natural, iterations of execution of task A
(3) $List\text{-}Tready_{A_i}$ // the list of ready times for task A during ith iteration
(4) $Tready_{A_i}$ // the ready times for task A during ith iteration
(5) $Tready_A$ // the ready times for task A from the current $Tready_{A_i}$
(6) $Execution\text{-}Interval_{A_i} = \oslash$ // the set of execution intervals for task A during ith iteration
(7) **for all** tasks A **do**
(8) **for all** execution iterations i of task A **do**
(9) $List\text{-}Tready_{A_i} = \{Treadymin_{A_i}, \ldots, Treadymax_{A_i}\}$
(10) **if** $i = 1$ **then**
(11) $Execution\text{-}Interval_{A_i} = \{[Tready_{A_i}, \min(Tready_{A_i} + P_A, \min_{B \in \varphi_A}(Treadymax_{B_i}), \mathrm{HP})], \forall Tready_{A_i} \in List\text{-}Tready_{A_i}\}$
(12) **else**
(13) **for all** $Tready_{A_1} \in List\text{-}Tready_{A_1}$ **do**
(14) $Tready_{A_i} = \mathrm{First}\ (List\text{-}Tready_{A_i})$
(15) **for all** $Tready_A \in \{Tready_{A_i}, \ldots, Treadymax_{A_i}\}$ **do**
(16) $Execution\text{-}Interval_{A_i} = Execution\text{-}Interval_{A_i} \bigcup \{[Tready_A, \min(Tready_{A_1} + i * P_A, \min_{B \in \varphi_A}(Treadymax_{B_i}), \mathrm{HP})]$, if $Tready_{A_1} + i * P_A$ is chosen as upper bound for this interval then it must respect: $Tready_A + C_A \leq Tready_{A_1} + i * P_A$, if HP is chosen then it must respect: $Tready_A + C_A \leq \mathrm{HP}\}$
(17) **end for**
(18) $List\text{-}Tready_{A_i} = \mathrm{next}\ (List\text{-}Tready_{A_i})$
(19) **end for**
(20) **end if**
(21) **end for**
(22) **end for**

Algorithm 2: Search of execution intervals.

$Execution\text{-}Interval_{A_i}$ at each possible iteration i during the HP. When this algorithm processes the first iteration of task A, the set of its possible execution intervals is determined directly (line 11) considering the precalculated ready times in $List\text{-}Tready_{A_1}$, and the periodicity and dependence constraints. For the next iterations, considering each possible $Tready_{A_1}$ for the purpose of respecting the periodicity constraint (line 13), at each iteration i, Algorithm 2 searches all the corresponding execution intervals starting with each possible $Tready_{A_i}$ (line 14, line 15), and considering the dependence and the deadline constraints, and the HP to determine the upper bound for each execution interval (line 16). In line 18, in order to guarantee the periodicity and dependence constraints, progressing from a $Tready_{A_1}$ to another in list $List\text{-}Tready_{A_1}$, Algorithm 2 must shift the list $List\text{-}Tready_{A_i}$ by omitting its first element, since ready times in each iteration i are also linked to the ready times of the first iteration.

(b) Search of Overlapping between Execution Intervals of Tasks in the Same RZ. For the first iteration, the parallel tasks on a given RZ are extracted directly from the DAG; hence, there is no parallel execution between a given task and its successors. However, in the next iterations i, searching parallel tasks must respect some rules. For the next iterations i, for parallel efficiency, a given task could overlap with its successors during their iterations j ($j < i$) on the same RZ. Moreover, this step must also consider the execution conflicts on the same RZ for a given task in its iteration i and other tasks that are in the iterations j ($j \leq i$) when there are no dependence constraints between them. This step prohibits simultaneous execution of several iterations of the same task.

The step defines for each task the possible RZs allowing its execution in terms of types and number of RBs based on computed cost D and then searches the execution conflicts on the RZs using Algorithm 3. Algorithm 3 has a worst-case temporal complexity equal to $O(\mathrm{NZ} * 2^{\mathrm{NT} * \mathrm{HP}})$.

During each iteration i, for each RZ, in order to find the parallel tasks with a finite cost D with the RZ, Algorithm 3 searches all the possible combinations of sets of execution intervals $\{Execution\text{-}Interval_{A_j}\}$ of all the tasks that can be executed on the current RZ and produce overlapping execution intervals respecting the rules described above (line 11). Then, from the resulting sets of execution intervals, Algorithm 3 extracts the execution intervals causing the conflicts in the current RZ which will result in *Comb* (line 12). *Comb* may contain two or more tasks. Each obtained *Comb* is processed individually to study the load of the RZ (line 13–line 24). For each *Comb*, we start the study only at the time at which all the tasks coexist to check for possible conflict and its consequence on the current RZ. Thus, Algorithm 3 searches the latest tasks $\{B\}$ in *Comb* (line 14) and for tasks $\{D\}$ that either have already started their executions and still running after $\{B\}$ arriving or have been ready before $\{B\}$ arriving; it searches their remaining execution times and periods (line 15–line 17) by promoting the tasks with the earliest deadlines and using the ASAP technique, especially in the case there are more than two tasks within the *Comb*. Finally, Algorithm 3 computes the

(1) A, B, D // Tasks
(2) i, j, k, m // Natural, iterations of execution of tasks
(3) *Execution-Interval*$_{A_i}$ // The set of execution intervals for task A during i^{th} iteration
(4) *Crossing-Combination* // All the possible combinations of sets *Execution-Interval*$_{A_i}$ of distinct tasks that give overlapping execution intervals on a given RZ, during HP
(5) *Comb* // Combination of overlapping execution intervals depicted by [A-min, A-max]
(6) C_D^r // The remaining execution time within task D
(7) P_D^r The remaining period within task D
(8) *Pmax* // The period of conflict for a combination of overlapping execution intervals
(9) **for all** iteration i **do**
(10) **for all** RZ **do**
(11) *Crossing-Combination* = $\{\{\text{Execution-Interval}_{A_j}\}/\bigcap_{j\leq i} \text{Execution-Interval}_{A_j} \neq \oslash\}$
(12) *Comb* = one combination of overlapping execution intervals extracted from sets in *Crossing-Combination*.
(13) **for all** *Comb* in *Crossing-Combination* **do**
(14) In *Comb*, search the latest tasks $\{B\}$ in starting execution
(15) In *Comb*, search all the remaining tasks $\{D\}$ that are ready or start execution before $\{B\}$ and determine their remaining execution times: C_D^r and their remaining periods: P_D^r
(16) $C_D^r = C_D - (B\text{-min} - D\text{-min})$
(17) $P_D^r = P_D - (B\text{-min} - D\text{-min})$
(18) **if** ($\forall D$-min, D-min = $Tready_{D_1} + (k-1) * P_D$, $\forall D$-max, D-max = $Tready_{D_1} + k * P_D$, $\forall B$-min, B-min = $Tready_{B_1} + (m-1) * P_B$ and $\forall B$-max, B-max = $Tready_{B_1} + m * P_B / Tready_{D_1}$ $\in \{Treadymin_{D_1}, \ldots, Treadymax_{D_1}\}$, $Tready_{B_1} \in \{Treadymin_{B_1}, \ldots, Treadymax_{B_1}\}$, and k and m are the iterations from which the execution intervals are taken respectively for tasks $\{D\}$ and $\{B\}$) **then**
(19) *RZ-Load* = $\sum_D C_D^r/P_D^r + \sum_B C_B/P_B$
(20) **else**
(21) *Pmax* = $\max_D(D\text{-max} - B\text{-min})$
(22) *RZ-Load* = $\sum_D C_D^r/Pmax + \sum_B C_B/Pmax$
(23) **end if**
(24) **end for**
(25) **end for**
(26) **end for**

Algorithm 3: Search of overlapping execution intervals and RZ loads.

load of the current RZ for this current *Comb* according to two cases. In the first case (Case 1, line 18-line 19), the remaining tasks with their new execution time values and periods are considered as virtual new tasks, and if their execution intervals in *Comb* corresponds to their total periods, Algorithm 3 intuitively computes the load of the RZ as mentioned in line 19 by considering the occupation rate (C_A/P_A) of each virtual new task A and each latest task $\{B\}$ in *Comb* on the current RZ.

In the second case (Case 2), the longest period of time (*Pmax*) that could be shared by the tasks in *Comb* is determined in line 21 and the load of the RZ is studied during *Pmax* (line 22).

This step deals with all possible cases of execution conflicts on all the RZs. At the end of this step, we obtain at each iteration during the HP, the loads of each RZ produced by each combination of tasks giving overlapping execution intervals. Consequently, we can detect the possible overloads in the RZs (*RZ-load* > 1). Some combinations might be included within other ones. When overloads are detected on some RZs, the next step resolves the problem either by performing migration of tasks respecting their preemption points or by incrementing the number of overloaded RZs until the overloads are covered.

(c) Task Migration or Addition of RZs. Migration of tasks causing an overload on a given RZ during a combination of simultaneous executions might be total or partial. Total migration consists in replacing the entire execution of one or many tasks on another RZs. Partial migration consists in the reallocation of some execution sections within tasks predefined by their preemption points to nonoverloaded RZs. The migration is performed as explained by Algorithm 4. The worst-case temporal complexity of Algorithm 4 is $O(2^{\text{NT}*\text{HP}} * \text{NT} * \text{NZ})$. Algorithm 4 searches the combinations producing overloaded RZs obtained by Algorithm 3 (line 6). Firstly, Algorithm 4 starts by total migration (line 8–line 15) to avoid the preemption of tasks resulting in difficulties related to context switches. During each *Comb* causing overload, the algorithm extracts the execution interval of the task that provides the largest occupation rate in the current *Comb* (line 10). In the case of equality between tasks, the task producing the fewest RZs for total migration during the *Comb* should be selected. The algorithm then determines the execution iteration of the task corresponding to this extracted interval and checks whether the iteration is also studied in another nonoverloaded RZs. Should this be the case, total migration of the task is allowed, and the task is eliminated from the overloaded RZ (line 11). If there are several RZs accepting total migration of the selected

```
(1) Comb // Combination of overlapping execution intervals
(2) Non-overload-comb // Boolean controlling after migration whether RZ is no longer overloaded by Comb
(3) RZ-Load // Load of the RZ corresponding to Comb
(4) RZ-migration // The set of RZs helpful for partial migration
(5) Task-migration // The set of tasks that might perform partial migration
(6) for all Comb giving overloaded RZ do
(7)       Non-overload-comb = False
(8)       while (Non-overload-comb = False) and (Comb ≠ ⊘) do
(9)          // Total migration of tasks
(10)         Select the interval from Comb that gives the most heavy occupation rate and discard it from Comb.
(11)         Check whether the iteration, corresponding to the execution interval of the selected task, is studied on another
              non-overload RZ and update the load of the overloaded RZ after the elimination of the selected task.
(12)         if RZ-Load ≤ 1 then
(13)            Non-overload-comb = True
(14)         end if
(15)      end while
(16)   if Non-overload-comb = False then
(17)      Reinitialize RZ-Load and Comb with its tasks
(18)      Task-migration = ⊘
(19)      RZ-migration = ⊘
(20)      // Partial migration of tasks
(21)      Omit the tasks from the overloaded RZ, corresponding to Comb, that are also acceptable by another RZs (D ≠ ∞) and
          reduce their occupation rates from the loads of these RZs. These latter RZs with the overloaded RZ corresponding to
          Comb are included in the set RZ-migration. The omitted tasks are included in Task-migration
(22)      while Task-migration ≠ ⊘ do
(23)         In Task-migration set, start by the task that gives the best trade-off between least number of RZs in RZ-migration
             where it could migrate and heaviest occupation rate in the overloaded RZ.
(24)         During Comb, within the selected task, choose the biggest execution sections that could be placed in RZs from the
              set RZ-migration without overloading them.
(25)         Update the load of RZs receiving execution sections from the selected task
(26)         if Some execution sections of the selected task are not placed then
(27)            Reinitialize the loads of RZs in RZ-migration to values before processing Comb
(28)            Increment the number of RZ corresponding to Comb up to ⌈RZ-Load⌉, go to 6
(29)         else
(30)            Discard the selected task from Task-migration.
(31)         end if
(32)      end while
(33)      // All the execution sections of tasks are placed
(34)      Non-overload-comb = True
(35)   end if
(36) end for
```

Algorithm 4: Total and partial migration or addition of RZs.

task, the RZ which is least required by tasks in the *Comb* is chosen. In cases of equality, the least loaded RZ is kept. After each total migration of a task, Algorithm 4 updates the load of the RZ corresponding to *Comb* and checks whether it is no longer overloaded (line 12–line 14). When total migration fails to resolve the overload in the RZ, partial migration takes place (line 16–line 35). Partial migration reinitializes the load of the current RZ corresponding to *Comb*. It searches the tasks in *Comb* that give finite D with other RZs and omits their occupation rates from the combinations on these latter RZs and from the current overloaded RZ considering the iteration of each task in *Comb* and includes the omitted tasks in the set *Task-migration*. The RZs accepting the tasks of *Comb* and the RZ corresponding to *Comb* are inserted within the set *RZ-migration* (line 21). Algorithm 4 attributes weights to tasks within *Task-migration* according to their occupation rates in *Comb* and according to the numbers of RZs in *RZ-migration* producing finite D with them. The task yielding the best trade-off between the two parameters is selected (line 23). Within the selected task, partial migration tries to reallocate its predefined execution sections in RZs from *RZ-migration* without causing an overload (line 24). Partial migration promotes the biggest execution sections respecting this rule. It starts by placing the selected execution section on the RZ which is least required by the other tasks in *Task-migration* waiting for partial migration. In cases of equality, it starts with the least loaded RZ in *RZ-migration*. Algorithm 4 pursues the processing of these partial migrations until the set *Task-migration* is empty. If partial migration does not successfully map all the execution sections of a given task in *Task-migration* (line 26), Algorithm 4 reinitializes the loads of the RZs to

their initial values before processing the *Comb* (line 27), stops its processing for the current *Comb*, and increments the number of the corresponding RZ to ⌈*RZ-Load*⌉ (line 28). When a given overloaded *Comb* is resolved by partial migration, Algorithm 4 takes into account the update of loads of all altered RZs (line 25) to be considered for the next *Comb*. Although migration might resolve many overloads for several combinations, it is still very difficult to perform as it is exhaustive and depends on the initial choices of tasks and RZs. One could consider the best case of studied migrations to minimize the number of added RZs. After each increment of RZ, the step must consider the added RZs to deal with the overloads of the remaining *Comb* not yet processed to avoid an excessive number of unusable RZs. As the proposed algorithm of migration is not exact, it might lead to an excessive number of RZs. This problem will be covered during resolution of mapping/scheduling which also optimizes the number of used RZs for the purpose of resource efficiency. However, an excess of RZs is very useful as it guarantees elimination of the infeasibility of analytic resolution and consequently, and it guarantees the feasibility of scheduling of the DAG.

At the end of this step, the number of RZs required to perform the scheduling of the chosen DAG on the FPGA is obtained. The resulting RZs constitute the target multi-reconfigurable-unit system where the scheduling of DAG will be conducted. The next step focuses essentially on determining the optimal valid scheduling that respects the predefined constraints.

3.3.4. Mapping and Scheduling Resolution. In the last step of the current stage, we concentrate on the resolution of mapping and scheduling tasks in the DAG on the resulting multi-reconfigurable-unit system. Mapping and scheduling are highly interlinked. It is well known that static multiprocessor scheduling of DAGs is performed by means of two actions: (i) assignment of an execution order expressed by temporal scheduling and (ii) assignment of processors expressed by mapping, for a set of tasks characterized by precedence and real-time constraints. With our methodology, based on a preemptive model, mapping consists in assigning each task to the most suitable RZs in terms of utilization of costly resources. Mapping is considered as spatial scheduling to a limited number of heterogeneous RZs. The scheduling searches the optimal scenario for task execution on RZs during the HP. At each execution iteration for a given task, it assigns for its execution sections specific times to launch their executions on the corresponding RZs. This scheduling is valid only when it satisfies predefined temporal constraints, and it should optimize the makespan of the graph, parallel efficiency, waiting time, and schedule response time. The proposed resolution leads to global static pipelined scheduling on a heterogeneous multi-reconfigurable-unit system because it is constructed at compile time, and it allows overlapping between execution iterations of distinct tasks on distinct RZs. Moreover, the problem of mapping/scheduling is a combinatorial optimization problem as it uses a discrete solution set and chooses the best combination of feasible assignments by optimizing a multiobjective function.

In this paper, resolution of mapping/scheduling is performed by mixed integer nonlinear programming solver as it is well adapted for this kind of problem. The mapping/scheduling problem is modeled by the quadruplet (constants, variables, constraints, and objective function).

Constants

NT:	Number of tasks constituting the DAG
NZ:	Number of RZs resulting from the task migration or addition of RZs analysis
NP:	Number of RB types existing in the target technology
i, o:	The references of iterations of executions during the HP
j:	The references of RZs
A, B:	The references of tasks
k:	The references of RB types
l, e:	The references of preemption points in tasks
t:	The references of times values, $t \in \{0, \ldots, \infty\}$
$D(A, \mathrm{RZ}_j)$:	The cost D between task A and RZ RZ_j
$RBCost_{\mathrm{k}}$:	The cost of each RB type
HP:	The hyperperiod in the DAG
C_A:	The worst case execution time of task A
P_A:	The period of task A which is equal to its relative deadline
$NbrPreemp_A$:	The number of possible preemption points within task A
$Preemp_{A,l}$:	The set of possible preemption points of task A. The first preemption point for all tasks is equal to 0
$Section_{A,l}$:	The execution section within task A provided by the predefined preemption point l
$NbrIter_A$:	The number of execution iterations of task A during the HP
$TimesValues_t$:	The set of possible times assigned to preemption points during the HP. It is equal to $\{0, \ldots, \infty\}$
$Pred_{A,B}$:	Binary constant takes 1 when task A has a precedence constraint with task B in the DAG
$Depend_A$:	Binary constant takes 1 when task A has data dependence constraints with tasks in the DAG
$Config_j$:	The configuration overhead of RZ_j
Com:	The maximum value of time for transmitting a data of unit length between two dependent tasks
$y_{A,B}$:	The amount of data sent by B for A execution.

Variables

$TUnicity_{j,A,l,t,i}$. Binary variable takes 1 when the preemption point l of task A is mapped to RZ_j at time t at iteration i. This

variable ensures the link between mapping and scheduling. In our resolution, the mapping/scheduling problem is solved when binary values are assigned to all these variables.

$PTimeRZ_{j,A,l,i}$. This variable represents the time value assigned to preemption point l of task A on RZ_j at iteration i during the HP. This variable is not defined when RZ_j gives infinite D with task A. It is obtained as expressed by

$$PTimeRZ_{j,A,l,i} = \sum_{\substack{t \\ 0 \le TimeValues_t \le \mathrm{HP}}} TUnicity_{j,A,l,t,i} \times TimeValues_t,$$
$$\forall 1 \le j \le \mathrm{NZ}, \quad 1 \le A \le \mathrm{NT}, \quad 1 \le l \le NbrPreemp_A,$$
$$1 \le i \le NbrIter_A, \quad D\left(A, \mathrm{RZ}_j\right) \neq \infty. \tag{12}$$

$PTime_{A,l,i}$. This variable provides the time value assigned to preemption point l of task A at iteration i during the HP and is calculated by means of

$$PTime_{A,l,i} = \sum_{\substack{1 \le j \le \mathrm{NZ} \\ D(A,\mathrm{RZ}_j) \neq \infty}} PTimeRZ_{j,A,l,i}, \quad \forall 1 \le A \le \mathrm{NT},$$
$$1 \le l \le NbrPreemp_A, \quad 1 \le i \le NbrIter_A. \tag{13}$$

$Occupation_{j,A}$. The total duration of execution of task A on RZ_j, after achievement of mapping/scheduling; it is computed by summing up all the execution sections of A mapped to RZ_j. This variable is not defined when RZ_j gives infinite D with task A. It is obtained using

$$Occupation_{j,A} = \sum_{\substack{l,i \\ 1 \le l \le NbrPreemp_A \\ 1 \le i \le NbrIter_A}} \sum_{\substack{t \\ 0 \le TimeValues_t \le \mathrm{HP}}} \left(TUnicity_{j,A,l,t,i} \times Section_{A,l}\right)$$
$$\forall 1 \le j \le \mathrm{NZ}, \quad 1 \le A \le \mathrm{NT}, \quad D\left(A, \mathrm{RZ}_j\right) \neq \infty. \tag{14}$$

$Exist_{j,A,l,i}$. This Binary variable tests whether the preemption point l of task A during its execution iteration i is mapped to RZ_j. This variable is not defined when RZ_j gives infinite D with task A. It is obtained by means of

$$Exist_{j,A,l,i} = \sum_{\substack{t \\ 0 \le TimeValues_t \le \mathrm{HP}}} TUnicity_{j,A,l,t,i}, \quad \forall 1 \le j \le \mathrm{NZ},$$
$$1 \le A \le \mathrm{NT}, \quad 1 \le l \le NbrPreemp_A,$$
$$1 \le i \le NbrIter_A, \quad D\left(A, \mathrm{RZ}_j\right) \neq \infty. \tag{15}$$

Constraints

Infeasibility of Mapping for Preemption Points. The constraint expressed by (16) prohibits the mapping of preemption points of task A to RZ_j giving infinite D with the task. Indeed, as explained in the second step of the task clustering stage, infinite D between a task and an RZ means that there is a lack of RBs in the RZ preventing task execution or an absence of RB types which are required by the task in the RZ. In addition, this constraint asserts that the time values chosen during mapping/scheduling for preemption points of tasks must be within the set of integers $\{0, \ldots, \mathrm{HP}\}$,

$$TUnicity_{j,A,l,t,i} = 0,$$
$$\text{when } D\left(A, \mathrm{RZ}_j\right) = \infty \quad \text{or} \quad TimeValues_t > \mathrm{HP},$$
$$\forall 1 \le j \le \mathrm{NZ}, \quad 1 \le A \le \mathrm{NT},$$
$$1 \le l \le NbrPreemp_A, \quad 1 \le i \le NbrIter_A, \quad 0 \le t \le \infty. \tag{16}$$

Uniqueness of Mapping/Scheduling Preemption Points on RZs. As expressed by (17), at each possible execution iteration i, if some tasks $\{A\}$ need to be scheduled on RZ_j at a time referred to as t, one task can be executed on this RZ at this time,

$$\sum_{\substack{A,l,i \\ 1 \le A \le \mathrm{NT} \\ 1 \le l \le NbrPreemp_A \\ 1 \le i \le NbrIter_A \\ D(A,\mathrm{RZ}_j) \neq \infty}} TUnicity_{j,A,l,t,i} \le 1, \quad \forall 1 \le j \le \mathrm{NZ},$$
$$0 \le TimeValues_t \le \mathrm{HP}, \quad 0 \le t \le \infty. \tag{17}$$

Uniqueness of RZs for Preemption Points. This constraint asserts that at each execution iteration i, each preemption point l of A must exist on a unique RZ_j (see (18)) and must be scheduled at a unique time referred to as t. This constraint also guarantees the achievement of task execution at each repetition, as all the preemption points delimiting the execution sections of the task are fitted on RZs at specified time values,

$$\sum_{\substack{j,t \\ 1 \le j \le \mathrm{NZ} \\ 0 \le TimeValues_t \le \mathrm{HP} \\ D(A,\mathrm{RZ}_j) \neq \infty}} TUnicity_{j,A,l,t,i} = 1, \quad \forall 1 \le A \le \mathrm{NT},$$
$$1 \le l \le NbrPreemp_A, \quad 1 \le i \le NbrIter_A. \tag{18}$$

Order of Preemption Points of a Task. At each execution iteration i for a task A, the preemption points must be scheduled in order, that is, the time value assigned to a preemption point $l+1$ must be superior to the completion of execution section specified by the preemption point l. Equation (19) guarantees a consistent order of scheduling

of preemption points for a given task and the completion of their corresponding execution sections,

$$PTime_{A,l+1,i} \geq PTime_{A,l,i} + Section_{A,l}, \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq l \leq NbrPreemp_A - 1, \quad 1 \leq i \leq NbrIter_A. \tag{19}$$

Order between Iterations. Based on (19), the constraint expressed by (20) requires that during each execution iteration i for a task A, the scheduling of tasks must lead to the completion of each execution section within A before the beginning of its next iteration $(i + 1)$. Simultaneous executions of distinct iterations for the same task are not permitted,

$$PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A} \leq PTime_{A,1,i+1}, \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_A - 1. \tag{20}$$

Upper Bound of Task Execution. Based on (19) and (20), (21) indicates that for each scheduled preemption point for a given task A, at each execution iteration i, its execution section must be completed before the HP. Otherwise, the resulting scenario is not considered a feasible scheduling,

$$PTime_{A,NbrPreemp_A,NbrIter_A} + Section_{A,NbrPreemp_A} \leq \mathrm{HP} \quad \forall 1 \leq A \leq \mathrm{NT}. \tag{21}$$

Nonoverlapping Execution. This constraint eliminates task conflicts on a given RZ. For all execution iterations, when several tasks are scheduled on the same RZ, their predefined execution sections must not overlap (see (22)). This constraint is also applicable within the same task, that is, overlapping running between execution sections is not allowed either during the same iteration, which is implicitly expressed by (19) by the imposed order for preemption point scheduling, or between distinct iterations, which is implicitly expressed by the imposed order for execution iterations of a given task in (20). Equation (22) describes this constraint between two tasks A and B existing on the same RZ_j,

$$Exist_{j,A,l,i} \times Exist_{j,B,e,o} \times \left(PTimeRZ_{j,A,l,i} + Section_{A,l}\right) \leq PTimeRZ_{j,B,e,o},$$

or

$$Exist_{j,A,l,i} \times Exist_{j,B,e,o} \times \left(PTimeRZ_{j,B,e,o} + Section_{B,e}\right) \leq PTimeRZ_{j,A,l,i},$$

$$\forall 1 \leq A, B \leq \mathrm{NT}, \quad 1 \leq j \leq \mathrm{NZ}, \quad 1 \leq l \leq NbrPreemp_A, \quad 1 \leq e \leq NbrPreemp_B, \quad 1 \leq i \leq NbrIter_A, \quad 1 \leq o \leq NbrIter_B. \tag{22}$$

Precedence and Dependence Constraints. Equation (23) defines the precedence and dependence constraints. At each possible execution iteration i of task A, if A has dependence links with tasks $\{B\}$, its execution for this iteration can be launched only if the running within ith execution iterations of all its predecessors $\{B\}$ has been completed and the required data for A execution have been received,

$$PTime_{A,1,i} \geq PTime_{B,NbrPreemp_B,i} + Section_{B,NbrPreemp_B} + Com \times y_{A,B} \quad \forall 1 \leq A, B \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_B, \quad Pred_{B,A} = 1. \tag{23}$$

Periodicity Constraint without Dependence Constraints. This constraint focuses on the periodic execution of tasks without predecessors in the DAG. Each task A is repeated periodically according to its specified period P_A in its functional model. Each repetition defines an execution iteration for the task. Equation (24) imposes constraints only on the first preemption point of the task as (19) defines an order for scheduling the preemption points in the same iteration which consequently will take into account the periodicity constraint for the other remaining preemption points,

$$PTime_{A,1,i} \geq (i-1) \times P_A \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_A, \quad Depend_A = 0. \tag{24}$$

Periodicity Constraint with Dependence Constraints. Equation (25) addresses tasks with predecessors in the DAG. These predecessors assert a dependence of execution on their successors at each repetition. Like for tasks without predecessors, tasks having dependence constraints with other tasks in the DAG must be repeated periodically as specified by their functional model and especially taking into consideration the first instants of completion of their predecessors. Indeed, a task with dependence constraints begins its first iteration after the execution achievement of all its predecessors and their data sending. Consequently, the periodicity constraint must consider the beginning time of these tasks with dependence constraints. Similarly, by means of (19), this constraint will be considered for all the preemption points,

$$PTime_{A,1,i} \geq \max_{\substack{B \\ B \neq A \\ Pred_{B,A}=1}} \left(PTime_{B,NbrPreemp_B,1} + Section_{B,NbrPreemp_B} + Com \times y_{A,B}\right) + (i-1) \times P_A, \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_A, \quad Depend_A = 1. \tag{25}$$

Deadline Constraint without Dependence Constraints. The key idea behind the constraint explained by (26) is to respect the strict real-time constraints in the case of the absence of

dependence constraints. At each given execution iteration i for a task A without predecessors, the running of task A must be completed before its absolute deadline,

$$PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A} \leq i \times P_A \quad \forall 1 \leq A \leq \mathrm{NT}, \quad 1 \leq i \leq NbrIter_A, \quad Depend_A = 0. \tag{26}$$

Deadline Constraint with Dependence Constraints. Equation (27) adheres to the deadline constraint in the case of the existence of dependence constraints between tasks in the DAG. Knowing that the beginning of a given task A with predecessors $\{B\}$ in the DAG is considered since the end of running of all its predecessors within their first execution iterations and the receipt of required data, the absolute deadline of A at each execution iteration i must be met as follows:

$$\begin{aligned} &PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A} \\ &\leq \max_{\substack{B \\ B \neq A \\ Pred_{B,A}=1}} \Big(PTime_{B,NbrPreemp_B,1} + Section_{B,NbrPreemp_B} \\ &\qquad + Com \times y_{A,B}\Big) + i \times P_A, \quad \forall 1 \leq A \leq \mathrm{NT}, \\ &\qquad 1 \leq i \leq NbrIter_A, \quad Depend_A = 1. \end{aligned} \tag{27}$$

Objective Function (F). The objective function guides resolution and helps to converge to the optimal solution. The minimization objective function F in (28) defines the optimal solution for the mapping/scheduling sub-problem by means of six parameters. F promotes the solution that provides the best trade-off of lowest values for these parameters. The parameters are all considered important for our methodology and are weighted according to their ranges of values. However, F gives priority to the number of occupied RZs more than other parameters. In fact, F increases exponentially with the minimum growth of the number of used RZs. Preference for this parameter is explained by the fact that our methodology is strongly dependent on physical architecture; hence, the minimum number of RZs enabling scheduling and satisfying the strict predefined constraints must be determined in order to avoid resource wastage and its consequences,

$$\begin{aligned} F &= \ln(Param) + \exp(NumberOfOccupiedRZs), \\ Param &= \delta_1 \times MakeSpan + \delta_2 \times WaitingTime \\ &\quad + \delta_3 \times ResourceOptimization \\ &\quad + \delta_4 \times MigrationNumber \\ &\quad + \delta_5 \times ConfigurationOverhead. \end{aligned} \tag{28}$$

MakeSpan. It is determined by the length of the obtained scheduling. This parameter is considered the most pertinent factor in multiprocessor scheduling of DAGs as it evaluates the efficiency of the performed scheduling in terms of parallelism on processors and execution speed. Equation (29) reduces the *MakeSpan* of the DAG by minimizing the time of finishing execution within the last execution iterations for all tasks as they are linked by precedence and dependence constraints. Consequently, as the execution iterations are also highly dependent, it is necessary to start the execution of a given task for each iteration as soon as possible in order to minimize the makespan of the DAG,

$$MakeSpan = \max_{\substack{A \\ 1 \leq A \leq \mathrm{NT}}} \Big(PTime_{A,NbrPreemp_A,NbrIter_A} + Section_{A,NbrPreemp_A}\Big). \tag{29}$$

WaitingTime. It is determined by the response time of the scheduling for task execution. By means of (30), *WaitingTime* is computed as the sum of differences between the obtained runtimes of the tasks during the scheduling span and their effective execution times. *WaitingTime* includes the waiting time of tasks in the ready queue of the scheduler waiting for execution when their dependence and periodicity constraints are satisfied and the blocking time when the task is preempted,

$$\begin{aligned} &WaitingTime \\ &= \sum_{\substack{A \\ 1 \leq A \leq \mathrm{NT}}} (RunTime_A - NbrIter_A \times C_A), \\ &RunTime_A \\ &= \sum_{\substack{i \\ 1 \leq i \leq NbrIter_A}} \Big(\Big(PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A}\Big) \\ &\qquad - (i-1) \times P_A\Big) \end{aligned}$$

if $Depend_A = 0$,

$$\begin{aligned} &RunTime_A \\ &= \sum_{\substack{i \\ 1 \leq i \leq NbrIter_A}} \Bigg(\Big(PTime_{A,NbrPreemp_A,i} + Section_{A,NbrPreemp_A}\Big) \\ &\qquad - \max\Bigg(\max_{\substack{B \\ B \neq A \\ Pred_{B,A}=1}} \Big(PTime_{B,NbrPreemp_B,1} \\ &\qquad\quad + Section_{B,NbrPreemp_B} \\ &\qquad\quad + Com \times y_{A,B}\Big) + (i-1) \times P_A, \\ &\qquad\quad \max_{\substack{B \\ B \neq A \\ Pred_{B,A}=1}} \Big(PTime_{B,NbrPreemp_B,i} \\ &\qquad\quad + Section_{B,NbrPreemp_B} \\ &\qquad\quad + Com \times y_{A,B}\Big)\Bigg)\Bigg), \end{aligned}$$

else (30)

ResourceOptimization. It is related to the physical features of the chosen technology. The parameter considers resource wastage and the utilization of costly resources. Both issues are involved in cost D computation in the first stage. Thus, resource optimization, using (31), targets mapping tasks with high occupation rates to the RZs providing the lowest cost D with them,

$$\begin{aligned} &ResourceOptimization \\ &= \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}}} \left(\frac{D\left(A,\ \mathrm{RZ}_j\right)^2 \times Occupation_{j,A}{}^2}{4} \right. \\ &\qquad \left. - D\left(A, \mathrm{RZ}_j\right) \times Occupation_{j,A} \right). \end{aligned} \tag{31}$$

MigrationNumber. Although the migration concept optimizes scheduling length, it helps to guarantee real-time constraints and to increase resource efficiency; it also raises configuration overhead. Bearing this in mind, we constructed (32) that aims at minimizing the number of migrations required. For a given task A, the first expression of (32), on the left of the summation, searches the number of migrations within the same execution iteration, and the second part of the summation calculates the performed migrations between iterations,

$$\begin{aligned} MigrationNumber = &\sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 1\le i\le NbrIter_A}} \ \sum_{\substack{1\le l<NbrPreemp_A\\ Exist_{j,A,l,i}\neq 0,\ Exist_{j,A,l+1,i}=0}} 1 \\ &+ \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}}} \ \sum_{\substack{1\le i<NbrIter_A\\ Exist_{j,A,NbrPreemp_A,i}\neq 0,\ Exist_{j,A,1,i+1}=0}} 1. \end{aligned} \tag{32}$$

ConfigurationOverhead. During the scheduling span, we do not consider the configuration overhead before the task execution on the RZ. This configuration overhead is negligible, that is, according to several experimentations performed with Virtex 5 technology, which uses parallel high-speed configuration ports, it represents less than 10% of the computation time of the task and does not impact the real-time functioning. However, after scheduling has been achieved, the configuration overhead, that impacts scheduling performance and power consumption, is evaluated using (33). The configuration overhead parameter should be reduced during resolution of mapping/scheduling. Equation (33) includes four expressions in the summation. The first expression, *Exp*1*,* is the initial configuration when tasks are launched by the scheduler for their first execution iteration as they did not exist on the RZs. The second expression, *Exp*2*,* represents the configuration overhead required to configure a task that migrates from one RZ to another within the same iteration i; the third expression, *Exp*3*,* depicts the configuration overhead required by a task that has finished its execution on a given RZ for an iteration i and started its $(i+1)$th iteration by migrating to another RZ. The fourth expression, *Exp*4, computes the configuration overhead resulting from intermediate tasks preempting a given task running on the same RZ. The first part of *Exp*4 considers the configuration overhead required in cases where a task A, during iteration i, finishes its execution section delimited by preemption point l, then it is preempted by other tasks to perform execution sections, which is then followed by task A performing its $l+1$ execution section at a later stage on the same RZ. This event is detected by the binary variable $CurrentIter_{j,A,l,i}$. This binary variable takes 1 if two successive execution sections for the same task at the same iteration are separated by one or more tasks on the same RZ. By means of variable $InterIter_{j,A,NbrPreempA,i}$, the second part of *Exp*4 deals with situations during which a task finishes its last execution section for a given iteration i on an RZ and starts the first execution section of $(i+1)$th iteration on the same RZ after the execution of other execution sections of other tasks on the same RZ. This variable takes 1 when at least one other task runs on the same RZ between two successive execution iterations of the same task,

$$\begin{aligned} &ConfigurationOverhead = Exp1 + Exp2 + Exp3 + Exp4 \\ &Exp1 = \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 0\le TimeValues_t\le \mathrm{HP}}} TUnicity_{j,A,1,t,1} \times Config_j \\ &Exp2 = \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 1\le i\le NbrIter_A}} \ \sum_{\substack{1\le l<NbrPreemp_A\\ Exist_{j,A,l,i}\neq 0,\ Exist_{j,A,l+1,i}=0}} Config_j \\ &Exp3 = \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}}} \ \sum_{\substack{1\le i<NbrIter_A\\ Exist_{j,A,NbrPreempA,i}\neq 0,\ Exist_{j,A,1,i+1}=0}} Config_j \\ &Exp4 = \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 1\le i\le NbrIter_A\\ 1\le l<NbrPreemp_A}} Config_j \times CurrentIter_{j,A,l,i} \\ &\qquad + \sum_{\substack{1\le A\le \mathrm{NT}\\ 1\le j\le \mathrm{NZ}\\ 1\le i<NbrIter_A}} Config_j \times InterIter_{j,A,NbrPreemp_A,i}. \end{aligned} \tag{33}$$

NumberOfOccupiedRZs. As explained in the previous step (determination of task execution intervals and the number of RZs), the obtained number of RZs ensures the feasibility of scheduling but is not necessarily the optimal number for valid scheduling. Thus, for the purpose of resource efficiency, (34) searches the lowest possible number of RZs required for

feasible real-time scheduling for the chosen DAG on a multi-reconfigurable-unit system,

$$NumberOfOccupiedRZs = \sum_{\substack{1 \le j \le \mathrm{NZ} \\ RZOccupation_j \neq 0}} 1,$$
$$RZOccupation_j = \sum_{\substack{A \\ 1 \le A \le \mathrm{NT}}} Occupation_{j,A}. \quad (34)$$

Other optimization parameters are also integrated in objective function F, but they are not mentioned in the formulation above as they implicitly impact the makespan. These parameters optimize the solutions that launch the execution of tasks as soon as their predecessors terminate their computations. In addition, they aim to bring the beginning of the tasks closer to the start time of each repetition.

After resolution of the first sub-problem of mapping/scheduling tasks on RZs, the resulting RZs must be placed on the target device. The second sub-problem of task cluster placement is a well-known problem in reconfigurable computing systems, and it differentiates our problem of scheduling on reconfigurable devices from that on software multiprocessors, generally not constrained by the physical architecture of the homogeneous processors. The second sub-problem of RZ placement is described in the next and last stage of our proposed methodology and like mapping/scheduling; it is solved by means of mixed integer programming as it is considered to be a combinatorial optimization problem.

3.4. Partitioning of Reconfigurable Device and Fitting of RZs. In general, the problem of hardware task placement includes two primary subfunctions: (i) *partitioning*, which handles free resource space on the reconfigurable device to identify potential sites enabling hardware task execution and (ii) *fitting*, which, according to the chosen criteria, selects a feasible placement solution among sites provided by the partitioning to fit the hardware task on a suitable physical location. Several research works have been proposed for the placement of hardware tasks on reconfigurable devices, and placement can be divided into on-line and off-line placement. For on-line placement, most scenarios propose as a partitioning technique to search for the maximal empty physical rectangles in the free space to avoid the resource wastage. For example, the two techniques proposed by Bazargan et al. in [13], one referred to as "*Keeping All Maximal Empty Rectangles*" searches all overlapping maximal empty rectangles and the other referred to as "*Keeping Nonoverlapping Empty Rectangles*" keeps only the distinct holes in the free space. In [14], Walder et al. describe efficient partitioning algorithms, the main one being the *On-The-Fly* (*OTF*) partitioner which resizes empty rectangles only if the new arrived task overlaps them. Handa and Vemuri introduce staircase partitioning in [15]. Marconi et al. extend in [16] Bazargan's partitioner by means of an *Intelligent Merging* (*IM*) algorithm that dynamically combines three techniques for managing free resources. In [17], Ahmadinia et al. present a new method of on-line partitioning which manages occupied space on devices rather than free space, given the difficulty of managing empty space and the resulting vast increase in empty rectangles. Regarding the fitting subfunction, the works are essentially based on bin-packing rules, such as in [13] which describes the *First Fit*, *Best Fit*, and *Bottom left* bin-packing algorithms. In [14], Walder et al. employ a hash matrix to perform fitting based on *Best Fit*, *Worst Fit*, *Best Fit with Exact Fit*, and *Worst Fit with Exact Fit* algorithms. In [17], Ahmadinia et al. fit tasks on sites, reducing communication costs, by means of the *Nearest Possible Position* algorithm. For off-line placement, many research works deal with metaheuristics such as simulated annealing, greedy research, and *Keeping All Maximal Empty Rectangle-Best Fit* as described in [13]. In [18], ElGindy et al. describe task rearrangement by using a genetic algorithm approach with the *First Fit* strategy. Considering task placement as a 2D packing problem, researchers propose off-line heuristics such as Lodi et al. who describe the *Next-Fit Decreasing Height*, *First-Fit Decreasing Height*, and *Best-Fit Decreasing height* algorithms in [19], as well as the *Floor-Ceiling* and *Knapsack packing* algorithms in [20]. Integer linear programming is also proposed by Panainte et al. in [21] to model the problem of hardware task placement by minimizing the resources area that is reconfigured at runtime.

With our methodology, we focus on off-line placement of RZs on reconfigurable devices. As the reconfigurable devices afford a limited number of reconfigurable resources and DAG includes bounded numbers of tasks, analytic resolution of the placement sub-problem is the recommended method as it guarantees the optimal solution to perform efficient allocation of tasks on FPGA. The placement of RZs is a combinatorial mono-objective problem. It consists in searching for suitable coordinates for each RZ in the FPGA among all the possible combinations of discrete coordinates. In this section, similarly to the mapping/scheduling sub-problem, the resolution of RZ placement on the reconfigurable device is accomplished through mixed integer programming and optimizes resource efficiency. The achievement of RZ placement results in a 2D physical locations for each task cluster. These physical locations are depicted by reconfigurable physical blocs (RPBs) and are represented by their RB model as described in (35). RZs are abstractions of hardware task clusters, and RPBs are the final reconfigurable units where tasks may be executed. With the concept of run-time partial reconfiguration, after the mapping/scheduling results are obtained respecting the strict predefined constraints and optimizing several criteria, task execution sections are reconfigured and executed on RPBs as restricted by the schedule scenario. Thus, the task bitstreams on their associated RPBs are created at compile time. These bitstreams are used during the DAG running as indicated by scheduling which specifies the corresponding state for each task on each RPB,

$$\mathrm{RPB}_j\_\mathrm{RB} = \left\{\gamma_{j,k}\mathrm{RB}_k\right\}, \quad \gamma_{j,k} \in \mathbb{N},$$
$$1 \le j \le \mathrm{NZ}, \quad 1 \le k \le \mathrm{NP}. \quad (35)$$

As shown in Figure 7, the partitioned RPBs on a reconfigurable device for a given RZ must include all its required RB types to ensure the execution of the tasks. The partitioned RPBs for a given RZ might contain some RBs that are not required by the RZ. For instance, RB_4 is inserted within RPB_3 but is not used by the corresponding RZ. This resource inefficiency is explained by the rectangular shape of the RPBs. Thus, the number of RBs included in the RPB could exceed the required RBs in the RZ. Resource inefficiency is also due to the heterogeneity of the device; partitioning could book some RBs which are not used by the RZ. RB utilization efficiency is an important metric parameter for evaluating placement quality. During RZ placement resolution, we focus on fitting RZs as closely as possible to RPBs in terms of number and types of RBs.

By means of mixed integer linear programming solver, both the partitioning and fitting subfunctions are solved simultaneously. The RZ placement sub-problem is modeled by the quadruplets (Constants, Variables, Constraints, and Objective Function).

Constants

NZ:	Number of RZs resulting from mapping/scheduling resolution
NP:	Number of RB types existing in the target technology RZ features, RB features
Device features	
Device_Width:	The width of the device
Device_Height:	The height of the device
Device_RB:	The RB model of the device.

Variables

Placement resolution consists in assigning discrete values for the four coordinates specifying the RPB_j for each RZ_j. Each RPB_j is constructed by four coordinates:

X_j:	The abscissa of the upper left vertex of RPB_j
Y_j:	The ordinate of the upper left vertex of RPB_j
$WRPB_j$:	The abscissa of the upper right vertex of RPB_j
$HRPB_j$:	The ordinate of the bottom left vertex of RPB_j.

Constraints

Heterogeneity Constraint. As RZs are fitted on RPBs, during RPB partitioning, the number of each RB type within the RPBs (X_j, $WRPB_j$, Y_j, $HRPB_j$) must be greater than or equal to those in the RZs ($\beta_{j,k}$) as formulated by (36) in order to satisfy the RB requirements of the RZs. Because of the heterogeneity of RBs in the device and the rectangular shape of RPBs, the partitioned RPBs could include some RB types not required by the RZs. Moreover, the number of RB types in the RPBs and included in the RZs might exceed that required by the RZs. This resource inefficiency is minimized by means of the objective function,

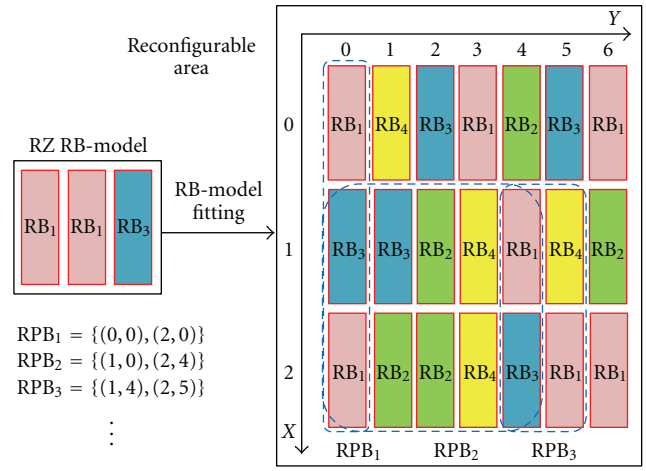


Figure 7: Example of RPBs for RZ.

$$\beta_{j,k} \leq \sum_{\substack{X_j \leq m \leq WRPB_j \\ Y_j \leq n \leq HRPB_j}} \sum_{device\_RB[m][n]=\mathrm{RB}_k} 1,$$

$$\forall 1 \leq j \leq \mathrm{NZ}, \quad 1 \leq k \leq \mathrm{NP},$$

$$\mathrm{RZ}_j\_\mathrm{RB} = \left\{\beta_{j,k}\ \mathrm{RB}_k\right\}, \quad \mathrm{RPB}_j\left(X_j, WRPB_j, Y_j, HRPB_j\right). \tag{36}$$

Nonoverlapping between RPBs. As expressed by (37), this constraint restricts the fitting of RZs on nonoverlapping RPBs

$$\begin{aligned} &X_q > WRPB_j \quad \text{or} \quad X_j > WRPB_q \\ &\text{or} \quad Y_q > HRPB_j \quad \text{or} \quad Y_j > HRPB_q, \\ &\forall j \neq q, \quad 1 \leq j, \quad q \leq \mathrm{NZ}. \end{aligned} \tag{37}$$

Objective Function (F). Objective function F comprises one parameter which is *ResourceEfficiency*. This primordial parameter focuses on finding the closest RPB partitioned on the FPGA to fit each RZ in terms of number and types of RBs. By respecting both previous constraints, (38) assesses placement quality after RZ insertion on the selected RPBs in order to achieve the optimal solution. Increasing resource efficiency reduces configuration overhead,

$$\begin{aligned} &ResourceEfficiency = \sum_{\substack{1 \leq j \leq \mathrm{NZ} \\ 1 \leq k \leq \mathrm{NP}}} RBCost_k \times \left(\gamma_{j,k} - \beta_{j,k}\right), \\ &\mathrm{RPB}_{j_{\mathrm{RB}}} = \left\{\gamma_{j,k}\ \mathrm{RB}_k\right\}, \quad \mathrm{RZ}_j\_\mathrm{RB} = \left\{\beta_{j,k}\ \mathrm{RB}_k\right\}, \\ &1 \leq j \leq \mathrm{NZ}, \quad 1 \leq k \leq \mathrm{NP}. \end{aligned} \tag{38}$$

Both subproblems, that is, (i) mapping/scheduling tasks on RZs and (ii) partitioning/fitting RZs on the FPGA, are successively solved by means of powerful solvers dedicated

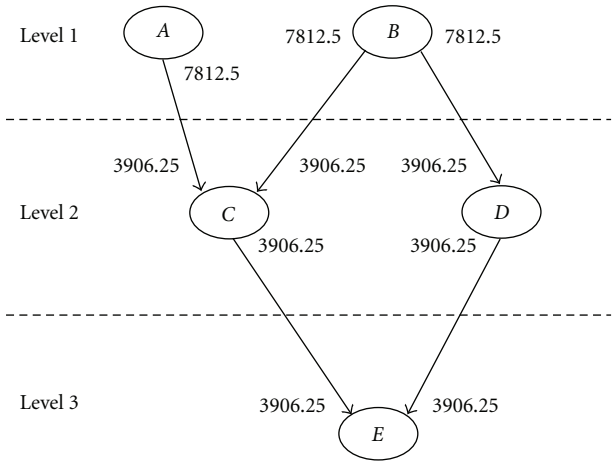


Figure 8: Tasks in the DAG.

by the AIMMS [22] environment. The chosen solvers for mixed integer programming satisfy predefined constraints and provide an optimal solution in an acceptable time frame. The following section shows how our proposed methodology may be applied to an example of DAG on Virtex 5 technology; the section also evaluates scheduling and placement quality.

4. Application and Results

The experiments were separated into two parts. The first part focuses on constructing the DAG as restricted by the rules of the mapping/scheduling stage, detailing the performed analysis and showing the results of task mapping/scheduling on the resulting RZs with a performance evaluation. The second part demonstrates placement resolution of the RZs on the reconfigurable device, Virtex 5 FX70T (ML507 board), and describes the metrics used to measure placement quality.

To illustrate the methodology proposed for static scheduling of DAGs on multi-reconfigurable-unit systems with precedence and deadline constraints, we designed the fully connected DAG shown in Figure 8. Tasks were selected from opencores (http://www.opencores.org/).

Each task is characterized by its worst-case execution time (WCET) which also integrates the communication latency for data sending to task successors, its period (relative deadline), and the set of preemption points. The number of needed slices is also synthesized for each task. In the slices, either only LUTs are used or only registers are used or both of them are used. The number of slices used by the task for LUTs and those for registers are mentioned in the third column of Table 2.

The configuration overheads were computed by (3) and by using our own IP based on the ICAP reconfiguration port having a width of 32 bits and a frequency of 100 MHz. Values on the DAG edges represent the average number of packets of 16 words of 32 bits for communication between tasks.

With the Virtex 5 technology [23], there are four main resource types, that is, CLBL, CLBM, BRAM, and DSP. Considering reconfiguration granularity, the RBs in Virtex 5 are vertical stacks composed of the same type of resources: RB_1 (20 CLBMs), RB_2 (20 CLBLs), RB_3 (4 BRAMs), and RB_4 (8 DSPs). In our application, for Virtex 5 FX70T, we considered that the lower the number of the RB types on the device and the higher its functioning speed, the more its cost would increase. We, respectively, assigned 16, 10, 168, and 194 as the *RBCost* for RB_1, RB_2, RB_3, and RB_4. We modeled the Virtex 5 FPGA and the hardware tasks with their RB-models. The task features are shown in Table 2.

The overview functioning of selected tasks is described below.

Reed-Solomon (*A*). It is an error correcting code that works by oversampling the Galouee's field polynomial constructed from the data to be coded. It is widely used to recover data from possible errors that occur during disk reading.

AES (*B*). Advanced encryption standard can decrypt input data using 256-bit key.

IIR (*C*). It performs infinite impulse response low-pass filter which cut off frequency in the range of 0.1 to 0.4 of the sampling frequency.

FFT (*D*). It performs 64 points Fast Fourier Transform where the data and coefficients are adjustable in the range 8 to 16 bits.

Basic DES (*E*). It performs single Data Encryption Standard by processing 16 identical rounds. Each round encrypts 32-bit block using 64-bit key to provide 32-bit output block.

We applied the proposed methodology for placement and scheduling of DAGs on reconfigurable devices and obtained the following results.

4.1. Task Clustering Results. This stage executes Algorithm 1 which results in RZ types (RZ_1, RZ_2, RZ_3) represented by their RB models in the first column of Table 3. RZ_1 is inserted by *A* and *B*, RZ_2 is provided by *C* and *D*, and task *E* creates RZ_3.

When the maximal numbers of RBs within a constructed RZ are provided by several tasks, the configuration overhead of the RZ must be recomputed as described by (3). In our application, the maximum numbers of RBs within RZs are created by the same task. Thus, the RZ configuration overheads are provided by the predefined task features shown in Table 2. The second step of this stage computes the costs *D* between the DAG tasks and the resulting RZs. In Table 3, the bold numbers are the lowest costs *D* for tasks with the most suitable RZs.

The resolution of scheduling of the chosen DAG on Virtex 5 FX70T is detailed in Sections 4.2 and 4.3.

4.2. Mapping/Scheduling Results and Scheduling Quality Evaluation. DAG behavior is studied within the time interval (HP) of a period equal to the least common multiple of the

Table 2: Task features.

Reference	Name	Slices (LUTs/registers)	WCET (μs)	Period (ms)	Configuration overhead (μs)	Preemption points (μs)	RB model
A	Reed-Solomon	2234/1224	26576	500	1116	14770, 20200	$\{8RB_1, 7RB_2, 1RB_3, 1RB_4\}$
B	AES	1150/507	42733	500	675	5000, 11000, 23000, 30000	$\{5RB_1, 5RB_2, 1RB_3, 1RB_4\}$
C	IIR	678/565	11805	250	524	3334, 10000	$\{4RB_1, 1RB_2, 0RB_3, 1RB_4\}$
D	FFT	2333/2010	11806	250	1199	5000, 11667	$\{9RB_1, 7RB_2, 0RB_3, 1RB_4\}$
E	Basic DES	387/192	22280	250	188	7500, 15000, 19000	$\{2RB_1, 2RB_2, 0RB_3, 0RB_4\}$

Table 3: RZ types and D costs.

	A	B	C	D	E
$RZ_1\{8RB_1, 7RB_2, 1RB_3, 1RB_4\}$	**0**	**68**	292	∞	448
$RZ_2\{9RB_1, 7RB_2, 0RB_3, 1RB_4\}$	∞	∞	**140**	**0**	356
$RZ_3\{2RB_1, 2RB_2, 0RB_3, 0RB_4\}$	∞	∞	∞	∞	**0**

periods of its tasks which is equal to 500 ms. Consequently, the execution iterations of tasks during the HP are deducted as follows:

$$\begin{aligned} NbrIter_A &= \left\lfloor \frac{\mathrm{HP}}{P_A} \right\rfloor = \left\lfloor \frac{500}{500} \right\rfloor = 1, \\ NbrIter_B &= 1, \qquad NbrIter_C = 2, \\ NbrIter_D &= 2, \qquad NbrIter_E = 2. \end{aligned} \tag{39}$$

The three steps preceding mapping/scheduling resolution are detailed hereunder. In the remaining sections of the paper, the values are expressed in μs.

4.2.1. Checking of Precedence, Dependence, and Real-Time Rightness in the DAG. The constraints for checking precedence, dependence, and real-time rightness in the proposed DAG are detailed below.

Dependence Checking. In Figure 8, it is obvious that the periods of tasks between the levels are guarded or dubbed. Thus, the successors are more repetitive than the predecessors.

In this case, all the data produced on an edge linking a source task and its successor must be sufficient and consumed by the latter task. As can be seen in Figure 8, the constraint expressed by (7) is satisfied for all the edges. For example, task B produces at its unique execution iteration a data of size $x_{B,C}$ equal to 7812.5 packets on its outgoing edge. These data are consumed totally by its first successor C during its two iterations where it consumes $y_{C,B}$ (3906.25 packets) data at each iteration. A similar remark can be made for data interchanged on the edge linking B and D.

Precedence Checking. This constraint is satisfied by all the interdependent tasks in the DAG and is checked by means of (8) and (9). $Tready_A = 0$, $Tready_B = 0$, $Tready_C = \max(C_A, C_B) = 42733$, $Tready_D = C_B = 42733$, $Tready_E = \max(Tready_C + C_C, Tready_D + C_D) = \max(42733 + 11805, 42733 + 11806) = 54539$.

Task C and Task B Precedence:

$$K = 1\text{: } 42733 + 250000 = 292733 > 0. \tag{40}$$

Task C and Task A Precedence:

$$K = 1\text{: } 42733 + 250000 = 292733 > 0. \tag{41}$$

Task D and Task B Precedence:

$$K = 1\text{: } 42733 + 250000 = 292733 > 0. \tag{42}$$

Task C and Task E Precedence:

$$\begin{aligned} &K = 1\text{: } 54539 + 250000 = 304539 > 42733, \\ &K = 2\text{: } 54539 + 2 * 250000 \\ &\qquad = 554539 > 42733 + 250000 \\ &\qquad = 292733. \end{aligned} \tag{43}$$

Task D and Task E Precedence:

$$\begin{aligned} &K = 1\text{: } 54539 + 250000 = 304539 > 42733, \\ &K = 2\text{: } 54539 + 2 * 250000 \\ &\qquad = 554539 > 42733 + 250000 \\ &\qquad = 292733. \end{aligned} \tag{44}$$

The above tests prove that each execution iteration A_i of a given task A includes only execution iteration B_i or iterations preceding B_i of each successor B. Consequently, the precedences between tasks in the chosen DAG are correct.

Real-Time Checking. For the purpose of real-time functioning, (10) considers dependence constraints between tasks and makes it possible to verify, in the best case of spatial conditions and by providing an RZ for each task, the rightness of real-time functioning according to computation times and periods assigned to tasks in the DAG.

An example of testing real-time constraints for task C is shown hereunder.

Task C

$$\Pi_C = \{A, B\},$$
$$K = 0:\ \max(0 + 26576, 42733 + 0) + 11805 = 54538 \le \min(42733 + 250000, 500000) = 292733\ (A),$$
$$\max(0 + 42733, 42733 + 0) + 11805 = 54538 \le \min(42733 + 250000, 500000) = 292733(B)$$
$$K = 1:\ (42733 + 250000) + 11805 = 304538 \le \min(42733 + 2 * 250000, 500000) = 500000. \quad (45)$$

By providing an RZ for each task, the deadlines of the tasks are met taking into consideration the dependence and precedence constraints.

The above three tests validate the chosen graph in terms of dependence, precedence, and real-time constraints and in the following section; spatial/temporal analyses are performed in order to determine the required number of RZs allowing valid scheduling for the DAG.

4.2.2. Determination of Lists of Ready Times. The first temporal analysis uses (11) to search the ready times of tasks which are helpful to limit execution intervals. Without considering the number of available reconfigurable units and taking into account precedence, dependence, periodicity, and real-time constraints, the ready times are determined as shown in Figure 9.

4.2.3. Determination of Task Execution Intervals and the Number of RZs. The first step of this spatial/temporal analysis uses Algorithm 2 to determine the possible execution intervals of tasks based on lists of ready times and respecting specified constraints. The possible execution intervals for tasks in the DAG are shown below

$$\textit{Execution-Interval}_{A_1} = \{[0, 215915], [1, 215915], \ldots, [i, 215915], \ldots, [189339, 215915]\},$$
$$\textit{Execution-Interval}_{B_1} = \{[0, 215914], [1, 215914], \ldots, [i, 215914], \ldots [173181, 215914]\},$$
$$\textit{Execution-Interval}_{C_1} = \{[42733, 227720], [42734, 227720], \ldots, [215915, 227720]\},$$
$$\textit{Execution-Interval}_{C_2} = \{[292733, 477720], [292734, 477720], \ldots, [465915, 477720]\},$$
$$\textit{Execution-Interval}_{D_1} = \{[42733, 227720], [42734, 227720], \ldots, [215914, 227720]\},$$
$$\textit{Execution-Interval}_{D_2} = \{[292733, 477720], [292734, 477720], \ldots, [465914, 477720]\},$$
$$\textit{Execution-Interval}_{E_1} = \{[54539, 304539], [54540, 304540], \ldots, [i, i + P_E], \ldots, [227720, 477720]\},$$
$$\textit{Execution-Interval}_{E_2} = \{[304539, \mathrm{HP}], [304540, \mathrm{HP}], \ldots, [477720, \mathrm{HP}]\}. \quad (46)$$

Using Algorithm 3, the analysis therefore integrates the spatial aspect and studies possible conflicts between tasks in shared RZs to detect possible overloads. Algorithm 3 must respect the rules explained in Section 3.3.3(b) to search the possible overlapping execution intervals. At each iteration, for each RZ, Algorithm 3 searches all the combinations of tasks causing overlapping execution intervals and inserts them in *Crossing-Combination* and computes the loads of RZs according to two cases as detailed as follows.

Iteration 1

RZ_1. *Crossing-Combination* = {{*Execution-Interval*$_{A_1}$, *Execution-Interval*$_{B_1}$}}. This *Crossing-Combination* contains several combinations of overlapping execution intervals between A and B. In the worst case of overlapping, such as the overlapping between $[0, 215915]$ from *Execution-Interval*$_{A_1}$ and $[0, 215914]$ from *Execution-Interval*$_{B_1}$, the load of RZ_1 obtained by Case 2 is 32% satisfying the predefined constraints.

RZ_2. *Crossing-Combination* = {{*Execution-Interval*$_{C_1}$, *Execution-Interval*$_{D_1}$}}. Similarly, this *Crossing-Combination* provides several combinations of overlapping execution intervals between C and D. In the worst case of overlapping, such as when both tasks have confused execution intervals equal to $[42733, 227720]$, the load of RZ_2 obtained in Case 2 is 12%. Another possible solution consists in total migration, as expressed by Algorithm 4, of task C to the RZ RZ_1 to improve execution parallelism and to minimize

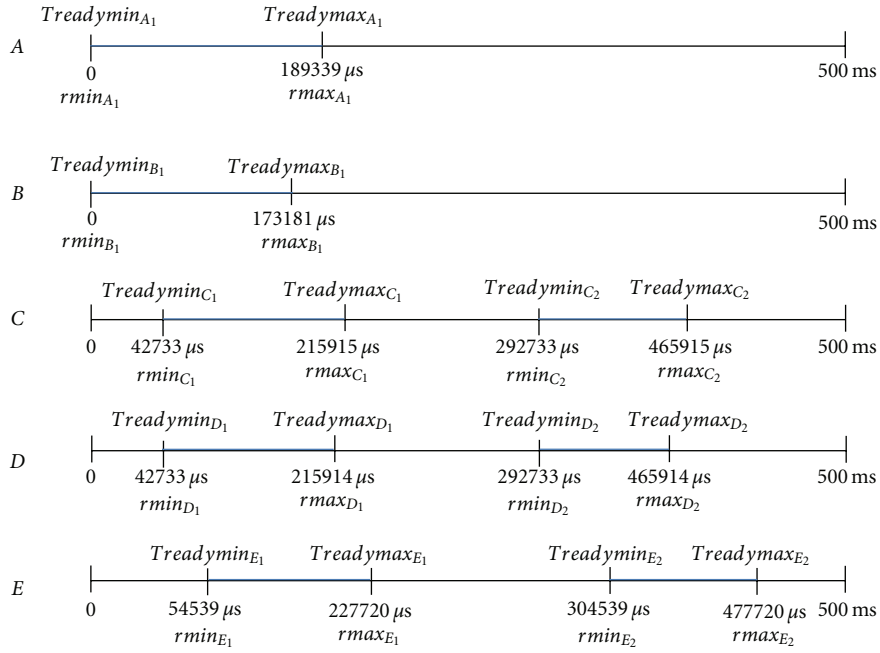


Figure 9: *Treadymin* and *Treadymax* of tasks in the DAG.

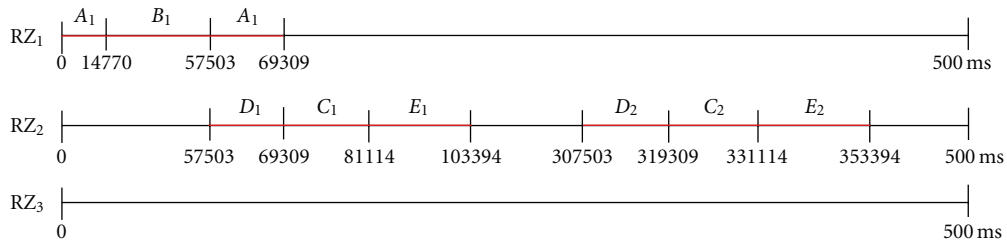


Figure 10: Mapping/scheduling resolution.

the makespan of the DAG. Thus, as soon as tasks A, B computations are achieved, tasks C and D may be launched, respectively, on RZ_1 and RZ_2.

During the first iteration, no combination of detected overlapping execution intervals causes an overload on RZs.

Iteration 2

RZ_2. *Crossing-Combination* = {{*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$}, {*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$, *Execution-Interval*$_{E_1}$}, {*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{D_2}$}, {*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{C_2}$}} {*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$}: As with the first iteration, all the combinations of overlapping execution intervals between tasks C and D do not cause an overload on RZ_2. The load of RZ_2, in the worst case of confused execution intervals, is equal to 12%. Hence, in the worst case, tasks C and D could be executed consequently on RZ_2 respecting all the predefined constraints.

As expressed by the rules set out in Section 3.3.3(b), detection of conflicting tasks must consider current and preceding iterations.

{*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$, *Execution-Interval*$_{E_1}$}: This set provides many combinations of overlapping execution intervals between tasks C, D, and E. However, following Algorithm 3 and the optimization parameters for reducing the makespan as explained at the end of mapping/scheduling resolution, task E starts its first iteration immediately after the achievement of its predecessors C and D which is equal, in the worst case, to 227720 μs, and RZ_2 is idle at this time. Consequent-ly, according to this start time of the first repetition, task E finishes its execution at 250 ms which is not attainable by the least start time of the second iterations of C and D, equal to 292733 μs. Thus, this set leads to no overlapping of execution between C, D, and E and becomes equivalent to the *Crossing-Combination* set: {*Execution-Interval*$_{C_2}$, *Execution-Interval*$_{D_2}$}. Another possible solution for this set is total migration of C to RZ_1 and E to RZ_3. This solution guarantees efficient parallelism between task computations.

{*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{D_2}$}, {*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{C_2}$}: As explained for the previous set, these sets of overlapping execution intervals do not cause an overload on RZ_2.

RZ_1

Crossing-Combination = {{*Execution-Interval*$_{E_1}$, *Execution-Interval*$_{C_2}$}}. As described for RZ_2 with the last three sets, tasks *C* and *E* do not cause overloads on RZ_1 during this *Crossing-Combination*. In fact, application of Algorithm 3 and the optimization parameters results in no remaining execution of the first repetition of *E* when *C* starts its second execution iteration.

After these spatial and temporal analyses, one can conclude there is no need to perform migration or to add other RZs. The three RZs resulting from the clustering stage are sufficient to perform valid scheduling of the chosen DAG. Thus, RZ_1, RZ_2, and RZ_3 represent the multi-reconfigurable-unit system for DAG scheduling. Based on powerful solvers dedicated by the AIMMS environment, we obtained the following mapping/scheduling which is evaluated in the next section.

4.2.4. Mapping and Scheduling Resolution. Figure 10 describes the mapping/scheduling obtained for five tasks in the DAG on three RZs. As the number of RZs is the highest priority parameter in the objective function, the resolution concludes that for scheduling this DAG, only two RZs are required. RZ_3 is not used, and hence it will not be placed on Virtex 5 FX70T in the third stage. The elimination of RZ_3 enhances resource efficiency and enables the DAG to be extended and performed on the FPGA for future needs. RZ_1 and RZ_2 are selected by the solver to remain in the multi-reconfigurable-unit system since *A* and *B* can execute only on RZ_1, task *D* can execute only on RZ_2, and tasks *C* and *E* can run on both RZs.

The obtained mapping/scheduling takes into account all the optimization parameters described in the sub-problem formulation and satisfies the specified constraints. Tasks *A* and *B* launch their computations first as they have no predecessors, and both can only execute on RZ_1. To reduce the makespan of the DAG, the scheduler decides the preemption of *A* at its second preemption point (14770 μs) to enable the execution of task *B*, the termination of which permits the execution of task *D* on RZ_2. The scheduler decides the execution of *A* before *B* in order to reduce the span between the executions of dependent tasks expressed by the dependence between *C*, *D*, and *E*. Hence, the scheduler promotes the solution that starts *C* immediately after completion of *A* and begins the execution of *E* once its predecessors *C* and *D* complete execution as reinforced by the optimization parameters explained at the end of mapping/scheduling resolution. These parameters aim at bringing closer the execution start of each task to the completion of its predecessors. *C* and *E* can execute on RZ_1 and RZ_2. The mapping assigns their executions to RZ_2 as this RZ provides the best utilization of costly resources guided by their costs *D*. For the purpose of fully exploiting the multi-reconfigurable-unit system and to increase parallel efficiency, task *A* resumes its execution simultaneously with the start time of the first repetition of task *D* at 57503 μs. In their second execution iterations, respecting the periodicity and precedence constraints tasks *C*, *D*, and *E* are performed on their optimal RZ, that is, RZ_2, to optimize resource utilization.

The resulting scheduling respects all the predefined constraints and during task running generates a small waiting time, equal to 42733 μs for task *A* when it is preempted and 14770 μs for task *B* in the ready queue. This waiting time represents only 16% of the overall running time. For the other tasks, the scheduler response is immediate, and the tasks are performed without preemption or migrations that substantially decrease the configuration overhead estimated in the following stage of resolution. The short response time of the scheduler is also reinforced by parallel computation.

Thanks to parallel computation and the optimization parameters, especially the launching of tasks whenever the start times of their repetitions are valid, the makespan is optimal and equal to 353394 μs. Compared to sequential execution of the DAG, the achieved speedup is 1.03.

This speedup refers to the amount by which the pipelined scheduling speeds up the sequential execution of the DAG. Consequently, the parallel efficiency of this scheduling indicates how efficiently the RZs are exploited to perform DAG execution and is obtained by dividing the achieved speedup by the number of used RZs in the multi-reconfigurable-unit system. In our resolution, the parallel efficiency is about 0.5 which is considered acceptable as tasks are heterogeneous and are not allowed to be executed in all the RZs. The resolution of this first sub-problem is conducted on a CPU of 2 GHz with 2 GB of RAM and lasts 6280 seconds.

4.3. Partitioning/Fitting RZs Results and Placement Quality Evaluation. Based on powerful solvers, RZ_1 and RZ_2 are fitted on their most suitable RPBs defined by the following coordinates after 80.63 seconds on Virtex 5 FX70T (8 × 47 RBs):

(i) RPB_1 for RZ_1: $X_1 = 14$, $Y_1 = 6$, $WRPB_1 = 32$, $HRPB_1 = 6$,

(ii) RPB_2 for RZ_2: $X_2 = 14$, $Y_2 = 3$, $WRPB_2 = 33$, $HRPB_2 = 3$.

Table 4 shows the comparison between the RBs of the obtained RPBs and their RZs expressed in the last column by cost Δ. The not null differences in RBs (Δ) are due to the rectangular shape of the RPBs and the heterogeneity of the device. Δ gives RB_3 in excess. In fact, both RPBs include RB_4, and in Virtex 5 FX70T there is no way to book RBs in a given RPB requiring DSP resources (RB_4) with CLBL and CLBM resources (RB_1 and RB_2) without crossing BRAM (RB_3) columns. Costs Δ also demonstrate how much the resulting placement of required RZs ensures resource efficiency. The low values found in Δ (1 RB_3 and 2 RB_3) show that the obtained RPBs are very close to their corresponding RZs, and consequently, resource efficiency is maintained during the conducted resolution.

Figure 11 depicts the floorplanning of RPB_1 and RPB_2 on Virtex 5 FX70T as obtained by their coordinates. Figure 11 also represents the placing/routing of tasks *A* and *D* named,

Table 4: Resource efficiency.

	RB_1	RB_2	RB_3	RB_4	Δ
RPB_1	8	7	2	1	1 RB_3
RPB_2	9	7	2	1	2 RB_3

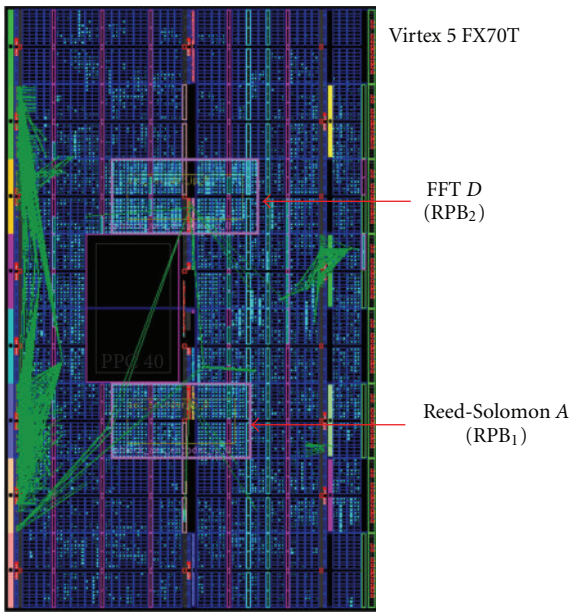


Figure 11: Floorplanning of RPBs and placing/routing of tasks *A* and *D* on Virtex 5 FX70T.

respectively, Reed-Solomon and FFT, which are running, respectively, on RZ_1 and RZ_2 during DAG execution in the time interval [57503, 69309]. The obtained results show an average resource utilization of 12.46% of the available resources on the reconfigurable device. This average is computed according to the number and the cost of each RB type. Optimization in the utilization of resources minimizes the area of the FPGA which is reconfigured at runtime. Due to dynamic partial reconfiguration, resource efficiency is improved by 17.3% compared to the static design of the chosen DAG. The static design is created by floorplanning each task in the DAG on its unique corresponding RPB without sharing any RPBs between different tasks. Once the RPBs are allocated on the reconfigurable device, their bitstreams are created, and their real configuration overheads are computed. The incurred reconfiguration overhead obtained after mapping/scheduling is 6729 μs and represents only 2% of the total running time of the DAG.

Although the experimental conditions are not the same in terms of DAG size and used architecture, we compared our results to attainable improvements in previous works of multiprocessor scheduling. Speedup in our resolution is modest and is about 1.03, compared to [6] which achieves 5.01 for FIR implementation and [3], which reaches a speedup of up to 2. Similarly, regarding parallel efficiency on processors, the results of [6] show a parallel efficiency of 1.3, [3] improves this parameter to 1, and the highest parallelism system is obtained in the work described in [10] which produces 9.3 degrees of parallelism. Nevertheless, our methodology ensures a parallel efficiency of 0.5. The modest improvement in speedup and parallelism results obtained with our methodology can be explained by the heterogeneity of the tasks, the non suitability of all the provided RZs to execute the tasks, and the small number of provided RZs in order to increase resource efficiency in reconfigurable devices. The highest improvement in speedup and parallel efficiency reached in previous works of multiprocessor scheduling does not take into account the resource efficiency and the processor heterogeneity. Effectively, the processors are considered homogeneous as they have identical features, and tasks are allowed to be executed on whatever processor whenever is idle.

However, compared to [18] where 80% of available resources are utilized, our methodology increases resource efficiency in heterogeneous devices by up to 17% compared to a static design and optimizes reconfigurable resource utilization by up to 12.46%. In contrast to [2] that targets an application of two or three tasks and attains a configuration overhead of 8% of total execution time, we immensely reduced the configuration overhead to 2% of the running time for a given DAG of five tasks. Similarly, in [24], Resano et al. attain 18% of configuration overhead to schedule only a JPEG decoder.

Concerning computational complexity, compared to the proposed heuristics to perform multiprocessor scheduling, the worst case temporal complexity is $O(n * (n + e))$, where n is the number of nodes, and e is the number of edges in the graph. We are aware that the efficiency and the optimality of our proposed methodology may be impaired by its temporal complexity in finding the optimal solution and which grows exponentially with the number of tasks in the DAG. It is estimated by: $O(2^{\mathrm{NT}*\mathrm{NZ}*NbrPreemp*\mathrm{HP}*NbrIter}) + O((Device_{width^2} \times Device_{height^2})^{\mathrm{NZ}})$, where *NbrPreemp* and *NbrIter* are, respectively, the maximum number of preemption points and the maximum number of execution iterations for a given task.

Thanks to the pertinent results obtained by the studied DAG, the efficiency of our proposed methodology in performing the placement and scheduling of small DAGs with few tasks is reinforced. However, due to the large size of search space containing the candidate solutions which will be checked for admission by constraints and evaluated for optimality by the objective function, our proposed approach is not capable to deal efficiently with intensive paralleled applications with hundreds of repetitive tasks. Effectively, using our approach in this class of application burdens the resolution time and the memory space, required for the computing of the several constraints and the different objective functions, and for searching the high number of assignments of possible values to the variables, parameters, and indexes. Currently, in our research project FOSFOR (Flexible Operating System FOr Reconfigurable platforms), the proposed methodology is efficiently employed as the target application is of type dataflow and contains small number of tasks.

5. Conclusion and Future Work

Under strict real-time constraints and from a parallel processing perspective, our paper deals with the problem of static scheduling of DAGs on multi-reconfigurable-unit system. In our opinion, most of the works proposed in this field do not consider resource efficiency or configuration overhead, and they are not applied to new heterogeneous technology. The approaches focus only on improving computation speedup and parallel efficiency for DAGs on homogeneous execution units. By means of a new methodology comprising three main stages and by selecting a preemptive model, we take into account periodicity, precedence, dependence, and real-time constraints, and we employ rigorous efficient analytic resolution in order to enhance quality of placement and scheduling in the most recent heterogeneous reconfigurable devices. Our methodology is illustrated in a realistic application, and the results obtained are encouraging. The resolution tries to find the trade-off between all the cited criteria since the performance of the DAG on reconfigurable devices is mainly defined by the degree of parallelism, the resource efficiency, and the amount of incurred reconfiguration. Our proposed approach is largely dependent on the physical features of tasks and technology as well as on temporal characteristics. During our tests, we concluded that our proposed methodology is efficient for small DAGs rather than for larger ones. In fact, solver processing is immensely delayed when the number of tasks in DAG exceeds five.

Static multiprocessor scheduling is a well-understood problem, and many efficient heuristics have been proposed to create compile-time scheduler scenarios. However, the approaches face difficulties in dealing with nondeterministic systems with run-time characteristics that are not well known before the DAG running. Thus, our future challenge is to define dynamic scheduling in heterogeneous reconfigurable devices to be applied for several DAGs of different sizes with nondeterministic behavior. We aim to consider intertask communication, all the specified constraints detailed throughout this paper, as well as to optimize all the cited criteria, especially degree of parallelism, resource efficiency, and configuration overheads.

Acknowledgments

The authors would like to thank the National Research Agency in France and the world-ranking "Secured Communicating Solutions" (SCS) cluster that sponsors our research project FOSFOR (flexible operating system for reconfigurable platforms). This work was also supported by AIMMS technical support and Xilinx tools.

References

[1] G. L. J. Djordjević and M. B. Tošić, "A heuristic for scheduling task graphs with communication delays onto multiprocessors," *Elsevier Parallel Computing*, vol. 22, no. 9, pp. 1197–1214, 1996.

[2] J. A. Clemente, C. Gonzalez, J. Resano, and D. Mozos, "A hardware task-graph scheduler for reconfigurable multitasking systems," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, pp. 79–84, Cancun, Mexico, December 2008.

[3] F. E. Sandnes and G. M. Megson, "Improved static multiprocessor scheduling using cyclic task graphs: a genetic approach," in *Parallel Computing: Fundamentals, Applications and New Directions*, vol. 12, pp. 703–710, North-Holland, 1998.

[4] Y. Abdeddaim, A. Kerbaa, and O. Maler, "Task graph scheduling using timed automata," in *Proceedings of the 17th IEEE International Symposium on Parallel and Distributed Processing (IPDPS '03)*, p. 237, Nice, France, April 2003.

[5] F. Redaelli, M. D. Santambrogio, and S. O. Memik, "An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," *International Journal for Reconfigurable Computing*, pp. 97–102, 2008.

[6] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architectures," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE '09)*, pp. 33–38, Nice, France, April 2009.

[7] F. E. Sandnes and O. Sinnen, "A new strategy for multiprocessor scheduling of cyclic task graphs," *International Journal High Performance Computing and Networking*, vol. 3, no. 1, pp. 62–71, 2005.

[8] M. Huang, H. Simmler, P. Saha, and T. El-Ghazawi, "Hardware task scheduling optimizations for reconfigurable computing," in *Proceedings of the 2nd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '08)*, pp. 1–10, Austin, Tex, USA, November 2008.

[9] S. Fekete, E. Kohler, P. Saha, and J. Teich, "Optimal FPGA module placement with temporal precedence constraints," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE '01)*, pp. 658–665, Munich, Germany, March 2001.

[10] Z. Pan and B. E. Wells, "Hardware supported task scheduling on dynamically reconfigurable SoC architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, Article ID 4633696, pp. 1465–1474, 2008.

[11] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1235–1238, 1975.

[12] I. Belaid, F. Muller, and M. Benjemaa, "Off-line placement of hardware tasks on FPGA," in *Proceedings of the 19th International Conference on Field Programmable Logic and Application (FPL '09)*, pp. 591–595, Prague, Czech Republic, September 2009.

[13] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[14] H. Walder, C. Steiger, and M. Platzner, "Fast online task placement on FPGAs: free space partitioning and 2D-hashing," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*, p. 178, Nice, France, April 2003.

[15] M. Handa and R. Vemuri, "An efficient algorithm for finding empty space for online FPGA placement," in *Proceedings of the Design Automation Conference (DAC '04)*, pp. 960–965, San Diego, Calif, USA, June 2004.

[16] T. Marconi, Y. Lu, K. Bertels, and G. Gaydadjiev, "Intelligent merging online task placement algorithm for partial reconfigurable systems," in *Proceedings of the Design Automation Test Europe Conference (DATE '08)*, pp. 1346–1351, Munich, Germany, March 2008.

[17] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A new approach for on-line placement on reconfigurable devices," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '04)*, p. 134, Santa Fe, New Mexico, April 2004.

[18] H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Task rearrangement on partially reconfigurable FPGAs with restricted buffer," in *Proceedings of the International Conference on Field Programmable Logic and Application*, pp. 379–388, Villach, Austria, August 2000.

[19] A. Lodi, S. Martello, and M. Monaci, "Two-dimensional packing problems: a survey," *European Journal of Operational Research*, vol. 141, pp. 241–252, 2001.

[20] A. Lodi, S. Martello, and D. Vigo, "Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem," in *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pp. 125–139, Kluwer Academic Publishers, 1997.

[21] M. Panainte, K. Bertels, and S. Vassiliadis, "FPGA-area allocation for partial run-time reconfiguration," in *Proceedings of the Design Automation Test Europe Conference (DATE '05)*, pp. 100–105, Munich, Germany, March 2005.

[22] http://www.aimms.com/.

[23] "Virtex-5 FPGA Configuration User Guide," Xilinx white paper, August 2009.

[24] J. Resano, D. Mozos, D. Verkest, S. Vernalde, and F. Catthoor, "Run-time minimization of reconfiguration overhead in dynamically reconfigurable systems," in *Proceedings of the International Conference on Field Programmable Logic and Application*, pp. 585–594, Lisbon, Portugal, September 2003.