EURASIP Journal on Embedded Systems

Reconfigurable Computing and Hardware/Software Codesign

Guest Editors: Toomas P. Plaks, Marco D. Santambrogio, and Donatella Sciuto

Reconfigurable Computing and Hardware/Software Codesign

EURASIP Journal on Embedded Systems

Reconfigurable Computing and Hardware/Software Codesign

Guest Editors: Toomas P. Plaks,
Marco D. Santambrogio, and Donatella Sciuto

Contents

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 731830, 2 pages
doi:10.1155/2008/731830

Editorial
Reconfigurable Computing and Hardware/Software Codesign

Toomas P. Plaks,[1] Marco D. Santambrogio,[2] and Donatella Sciuto[2]

[1] *University of Reading, Berkshire RG6 6AH, UK*
[2] *Politecnico di Milano, 20133 Milano, Italy*

Correspondence should be addressed to Toomas P. Plaks, plakstp@aol.com

Received 23 December 2007; Accepted 23 December 2007

Modern consumer appliances as wireless communication and multimedia systems present very strong requirements for the digital parts of these systems: digital design process must provide solutions which possess high performance, flexibility for multifunctional use, and energy efficiency. Over the past decade, the reconfigurable computing platform has been an emerging approach in scientific research and in practical implementations to meet these requirements.

The special issue on "Reconfigurable Computing and Hardware/Software Codesign" addresses the advances in reconfigurable computing architectures, in algorithm implementation methods, and in automatic mapping methods of algorithms onto hardware and processor spaces, indicating the changes in codesign flow due to the introduction of new, reconfigurable hardware platform. Using this platform, the designer faces a new paradigm of computing and programming: the computing system is capable of run-time and autonomous modification of its functionalities following the changing needs of applications.

This new scenario of hardware/software codesign provides a great improvement in the embedded system design and implementation. To cope effectively and timely with the new challenges, the new and more sophisticated dynamic reconfiguration strategies together with codesign methods have to be developed.

In the first paper, "Design flow instantiation for run-time reconfigurable systems: a case study," Y. Qu et al. present a design flow instantiation for run-time reconfigurable systems using a real-life application—part of a WCDMA decoder. The design flow is roughly divided into two parts: system level and implementation. At system level, hardware resource estimation and performance evaluation are applied. At implementation level, technology-dependent tools are used to realize the run-time reconfiguration. The results show that run-time reconfiguration can save 50% of the area when compared to a functionally equivalent fixed system and achieves 30 times speedup in processing time when compared to a functionally equivalent pure software design.

In "A flexible system level design methodology targeting run-time reconfigurable FPGAs," F. Berthelot et al. present an automatic design generation methodology for heterogeneous architectures. This method automatically generates designs for fixed and partially reconfigurable parts of an FPGA and enables a reconfiguration prefetching technique to minimize reconfiguration latency and buffer-merging techniques to minimize memory requirements of the generated design. This concept has been applied to different wireless access schemes, based on a combination of OFDM and CDMA techniques.

The next paper, "RRES: a novel approach to the partitioning problem for a typical subset of system," by G. B. Knerr et al., integrates some of the most powerful approaches for system partitioning into a consistent design framework for wireless embedded systems, which has led to the development of an entirely new approach for the system partitioning problem. The paper introduces the restricted range exhaustive search algorithm and compares this to popular and well-reputed heuristic techniques based on tabu search, genetic algorithm, and the global criticality/local phase algorithm. This search algorithm proves superior performance for a set of system graphs featuring specific properties found in human-made task graphs, since it exploits their typical characteristics such as locality, sparsity, and their degree of parallelism.

The paper "Software-controlled dynamically swappable hardware design in partially reconfigurable systems," by C. Huang and H. Pao-Ann, considers different wrapper designs for hardware designs such that they can be enhanced

with the capability for dynamic swapping controlled by software. A hardware design with proposed wrappers can be swapped out of the partially reconfigurable logic at run-time in some intermediate state of computation and then swapped in when required to continue from that state. With the capability for dynamic swapping, high-priority hardware tasks can interrupt low-priority tasks in real-time embedded systems so that the utilization of hardware space per unit time is increased.

In "DART: a functional-level reconfigurable architecture for high energy efficiency," S. Pillement et al. deal with functional-level reconfiguration to improve energy efficiency. The paper presents the DART architecture, which supports two modes of reconfiguration: fine-grained and functional level, to achieve the optimized solutions. The compilation framework is built using compilation and high-level synthesis techniques. As a proof of the concept, a 3G mobile communication application has been implemented and the VLSI design of a 0.13 μm CMOS SoC implementing a specialized DART cluster is presented.

The last paper of this issue "Exploiting process locality of reference in RTL simulation acceleration," by A. D. Blumer and C. D. Patterson, addresses the simulation acceleration of digital designs. An analysis of six register transfer level (RTL) code bases shows that only a subset of the simulation processes is executing at any given time. Run-time adaptations are made to ensure that acceleration resources are not wasted on idle processes, and these adaptations may be effected through process migration between software and hardware. Finally, the paper describes an implementation of an embedded, FPGA-based migration system; the empirical data are obtained for use in mathematical and algorithmic modelling of more complex acceleration systems.

Toomas P. Plaks
Marco D. Santambrogio
Donatella Sciuto

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 856756, 9 pages
doi:10.1155/2008/856756

Research Article

Design Flow Instantiation for Run-Time Reconfigurable Systems: A Case Study

Yang Qu,[1] Kari Tiensyrjä,[1] Juha-Pekka Soininen,[1] and Jari Nurmi[2]

[1] *Communication Platforms, Technical Research Centre of Finland (VTT), Kaitoväylä 1, 90571 Oulu, Finland*
[2] *Institute of Digital and Computer Systems, Tempere University of Technology, KorkeaKoulunkatu 1, 33720 Tampere, Finland*

Correspondence should be addressed to Yang Qu, yang.qu@vtt.fi

Received 25 May 2007; Revised 28 September 2007; Accepted 12 November 2007

Recommended by Donatella Sciuto

Reconfigurable system is a promising alternative to deliver both flexibility and performance at the same time. New reconfigurable technologies and technology-dependent tools have been developed, but a complete overview of the whole design flow for run-time reconfigurable systems is missing. In this work, we present a design flow instantiation for such systems using a real-life application. The design flow is roughly divided into two parts: system level and implementation. At system level, our supports for hardware resource estimation and performance evaluation are applied. At implementation level, technology-dependent tools are used to realize the run-time reconfiguration. The design case is part of a WCDMA decoder on a commercially available reconfigurable platform. The results show that using run-time reconfiguration can save over 40% area when compared to a functionally equivalent fixed system and achieve 30 times speedup in processing time when compared to a functionally equivalent pure software design.

1. INTRODUCTION

Reconfigurability is an important issue in the design of system-on-chip (SoC) because of the increasing demands on silicon reuse, product upgrade after shipment, and bug-fixing ability. The reconfigurability is usually achieved by embedding reconfigurable hardware into the system. The result is a heterogeneous SoC that has the advantages of both reconfigurable hardware and traditional types of computing elements such as general-purpose processors (GPP) and application-specific integrated circuit (ASIC). Such combination allows parts of the system to be reconfigured at run time while the rest is still running. This feature is referred to as run-time reconfiguration (RTR), which can significantly increase silicon reusability.

As today's applications become more and more complex, the implementation needs more hardware resources. It means that either larger chips or more chips are needed, which might not be suitable for many products such as portable devices that require to have a limited dimension. With RTR, tasks that are nonoverlapping either in time domain or space domain can be mapped onto the same reconfigurable logic. Tasks that are required initially can be configured in the beginning. When another task is required, the configuration to load it can then be triggered. For example, in a typical smartphone environment, different wireless technologies, such as GSM, WCDMA, WLAN, and WiMax in the future, have to be supported. However, it is not likely that these wireless technologies will be used at the same time. Therefore, it is possible to put them into reconfigurable logic and dynamically load the one that is needed.

A number of reconfigurable platforms are commercially available. Xilinx [1] and Altera [2] provide fine-grain FPGA platforms. They contain embedded processor cores, which make it possible to design rather complex systems in such FPGA platforms. PACT XPP [3] and QuickSilver [4] provide coarse-grain reconfigurable computing platforms, which are suitable for DSP-type tasks. The Triscend A7S [5] and the Motorola MRC6011 [6] are configurable SoCs, which bring both high flexibility and high performance.

The drawbacks of the RTR are configuration latency and power consumption related to the configuration process, which can largely degrade the system performance. How to address these problems and evaluate the effects of reconfiguration at an early phase of the design are not supported in existing system-level design methodologies and

tools. In addition, at system level, how to support and make system partitioning for not only software and hardware, but also reconfigurable logic, needs to be studied.

The ultimate goal of our work is to develop a complete design methodology and highly automatic tools for design of reconfigurable SoC (RSoC). In this paper, we present a design flow instantiation for implementing part of a WCDMA decoder in an RTR system. At system level, our supports for system partitioning and performance evaluation are applied [7, 8]. At implementation level, commercial and technology-dependent tools are applied. The structure of the paper is as follows. Related work is presented in Section 2. Brief explanations of the case study and the target platform are given in Section 3. The system-level design flow instantiation is presented in Section 4, and low-level implementation work is presented in Section 5. Finally, the conclusions are given in Section 6.

2. RELATED WORK

System-level design covers various issues, such as partitioning, task scheduling, and synthesis. In [9], an SW/HW partitioning and online task scheduling approach are presented. In [10], a survey of various SW/HW partitioning algorithms is presented, and a new approach to map loops into reconfigurable hardware is proposed. In [11], a codesign environment for DSPs/FPGAs-based reconfigurable platforms is presented. Both applications and architectures are modeled as graphs, and an academic system-level CAD tool is used. In [12], a macro-based compilation framework to replace logic synthesis and technology mapping is presented. In [13], a synthesis approach based on list-scheduling is presented. The target system is a single application that can be divided into a number of dependent tasks. The approach considers HW/SW partitioning, temporal partitioning, as well as context scheduling. In [14, 15], an HW/SW cosynthesis framework for real-time embedded reconfigurable system is presented. Each application consists of a number of dependent tasks and has its own period. A task can be mapped either onto a host processor or dynamically reconfigurable hardware (DRHW). Synthesis is performed by statically scheduling the applications over a hyperperiod, which is the least common multiple of the different application periods. In [16], a SystemC simulation environment for RTR systems is presented. The idea is to remove the unloaded module from the activation list of the SystemC kernel. In [17], various system-level approaches to reduce the effect of configuration latency are studied.

Different from these approaches, our ultimate goal is not to develop a fully automatic system partitioning approach, which we believe will not succeed. This is because nowadays' applications and platforms are becoming so complex that it is not possible to quantitatively characterize them precisely in the early design phase so that complex mathematical formulas can be applied to fully partition the design in such a way that optimal solutions can be guaranteed. However, providing supports to designers at this phase can help to prune the design space and possibly avoid re-designs. In our work, approaches to support partitioning and modeling for fast design space exploration are provided. To reduce coding effort, a tool to automatically generate reconfigurable components is developed.

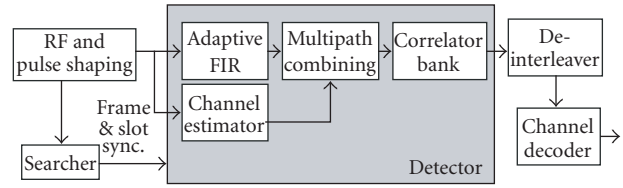


Figure 1: The WCDMA base-band receiver system.

3. APPLICATION AND TARGET PLATFORM

Reconfigurable system is a promising alternative to deliver both flexibility and performance at the same time. Technology-dependent tools and high-level abstract supporting tools have been proposed to solve the various design problems at different abstraction levels. However, a complete overview of how to integrate them into a single design flow is missing. In this work, we use a real case study to demonstrate our design flow of RTR systems. The design case is a set of wireless communication functions [18], and the target is a RTR-type implementation on VP20FF1152 development board from Memec Design group [19], which contains one Virtex2P XC2VP20 FPGA [1] that supports partial RTR.

The whole WCDMA base-band receiver system is depicted in Figure 1. The case study focuses on the detector portion (shaded area in Figure 1) of the receiver and a limited set of the full features were taken into account. It uses 384 kbits/s user data rate without handover. The functions are an adaptive filter, a channel estimator, a multipath combiner, and a correlator bank. The adaptive filter is performing the signal whitening and part of the matched filtering implemented traditionally with the RAKE receiver. The channel estimator module calculates the phase references. In the combiner part, the different multipath chip samples are phase rotated according to the reference samples and combined. Finally, the received signal is despread in the correlator bank.

4. SYSTEM-LEVEL DESIGN FLOW AND INSTANTIATION STEPS

Our design flow is divided into system-level design and implementation-level design. The task at system-level design is to make various partitioning decisions and evaluate system performance. At implementation level, executable code and HW netlist are generated using technology-dependent tools. A generic view of the system-level design flow is depicted in Figure 2. The following new features are identified in each phase when reconfigurability is taken into account.

(i) *System requirements/specification capture* needs to identify requirements and goals of reconfigurability (e.g., flexibility for specification changes and performance scalability).

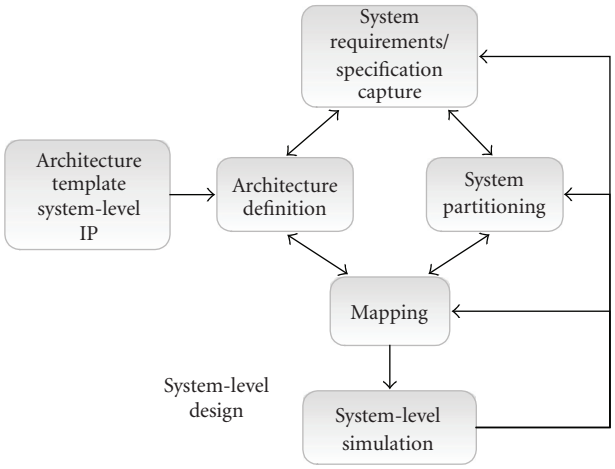
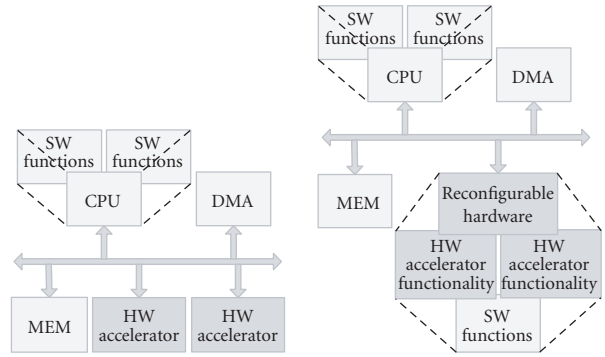


Figure 2: A generic system-level design flow.



(a) An initial fixed SoC architecture

(b) A modified architecture using reconfigurable hardware

Figure 3: Creating RSoC from fixed platform.

(ii) *Architecture definition* needs to model the reconfigurable technologies of different types and vendors at abstract level and include them in the architecture models.

(iii) *System partitioning* needs to analyze and estimate the functions of the application for SW, fixed HW, and reconfigurable HW. The parts of the targeted system that will be realized on reconfigurable HW must be identified. There are some rules of thumb that can be applied. If an application has roughly several same-sized hardware accelerators that are not used at the same time, these accelerators can be implemented onto DRHW. If an application has some parts in which specification changes are foreseeable or there are foreseeable plans for new generations of the applications, it may be beneficial to implement it onto DRHW.

(iv) *Mapping needs* to map functions allocated to *reconfigurable* hardware onto the respective architecture model. The special concerns at this step are the temporal allocation and the scheduling problem. Allocation and scheduling algorithms need to be made either online or offline.

(v) *System-level simulation* needs to observe the performance impacts using reconfigurable resources for a particular system function. The effect of configuration overhead should be highlighted in order to support designers to perform system analysis or design space exploration.

It should be noted that reconfigurability does not appear as an isolated phenomenon, but as a tightly connected part of the overall SoC design flow. Our approach is therefore not intended to be a universal solution to support the design of any reconfigurability. Instead, we focus on a case, where the reconfigurable components are mainly used as coprocessors in SoCs. In addition, we assume that RTR system design does not start from scratch, but it is a more advanced version of an existing device. The new architecture is defined partly based on the existing architecture and partly using the system specification as input. The initial architecture is often dependent on many things not directly resulting from the requirements of the application. The company may have experience and tools for certain processor core or semiconductor technology, which restricts the design space and may produce an initial hardware/software (HW/SW) partition. Therefore, the initial architecture and the HW/SW partition are often given at the beginning of system-level design.

SystemC 2.0 is selected as the backbone of the approach since it provides a universal environment to model HW/SW and seamlessly cosimulate them at different abstract levels. The way that the SystemC-based approach incorporates dynamically reconfigurable parts into architecture is to replace SystemC models of some hardware accelerators, as shown in Figure 3(a), with a single SystemC model of reconfigurable block, as shown in Figure 3(b). The objective of the SystemC-based extensions is to provide a mechanism that allows designers to easily test the effects of implementing some components in DRHW. Referring to the system-level design flow, as shown in Figure 2, we provide estimation support for system partitioning and modeling support for system-level simulation.

4.1. Estimation approach to support system analysis

The estimation approach [20] is used to support system analysis to identify functions that are beneficial to be implemented in DRHW. It focuses on VirtexII-like FPGA [1] DRHW, in which the main resources are lookup-tables (LUTs) and multipliers. The estimation approach starts from function blocks represented using C language, and it can produce the following estimates for each function block: execution time in terms of running the function on DRHW and resource utilization of DRHW. The framework of the estimation approach is shown in Figure 4. The designer decides the granularity of partitioning by decomposing the algorithm down to function blocks. The estimation tool produces the estimates for each of the functions.

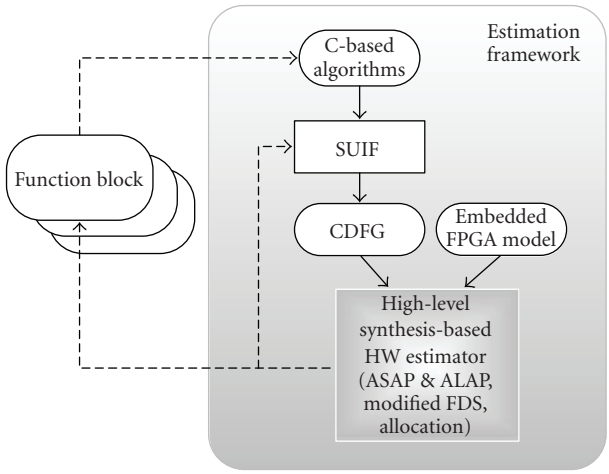


Figure 4: The estimation framework.

4.1.1. High-level synthesis-based hardware estimation

A graphical view of the hardware estimation is shown in Figure 4. Taking control-data flow graph (CDFG) and a model of embedded FPGA as inputs, the hardware estimator carries out a high-level synthesis-based approach to produce the estimates. Main tasks performed in the hardware estimator as well as in a real high-level synthesis tool are scheduling and allocation. Scheduling is the process in which each operator is scheduled in a certain control step, which is usually a single clock cycle, or crossing several control steps if it is a multi-cycle operator. Allocation is the process in which each representative in the CDFG is mapped to a physical unit, for example, variables to registers, and the interconnection of physical units is established.

In the estimator, a function block is represented as a CDFG, which is a combined representation of data flow graph (DFG) that exposes the data dependence and parallelism of algorithms, and control flow graph (CFG) that captures the control relation of a group of DFGs. A SUIF-based [21] front-end pre-processor is used to extract CDFG from the C code. First it dismantles all high-level loops (e.g., while loop and for loop) into low-level jump statements and restrict the produced code to minimize the number of jumps. Then, basic blocks are extracted. A basic block contains only sequential statements without any jump in between. Data dependence inside each basic block is analyzed, and a DFG is generated for each basic block. After the creation of all DFGs, the control dependence between these DFGs is extracted from the jump statements to construct the CDFG. Finally, profiling results, which are derived using gcov [22], are inserted into the CDFG as attributes.

The basic construction units of the embedded FPGA are static random access memory (SRAM)-based lookup tables (LUT) and certain types of specialized function units, for example, custom-designed multiplier. Routing resources and their capacity are not taken into account. The model of the embedded FPGA is in a form of mapping-table. The index of the table is the type of the function unit, for example, adder. The value mapped to each index is hardware resources in terms of the number of LUTs and the number of specialized units for this type of operation.

As-soon-as-possible (ASAP) scheduling and as-late-as-possible (ALAP) scheduling [23] determine the critical paths of the DFGs, which together with the control relation of the CFGs are used to produce the estimate of hardware execution time. For each operator, the ASAP and ALAP scheduling processes also set the range of clock cycles within which it could be legally scheduled without delaying the critical path. These results are required in the next scheduling process, a modified version of force-directed-scheduling (FDS) [24], which intends to reduce the number of function units, registers, and buses required by balancing the concurrency of the operations assigned to them without lengthening the total execution time. The modified FDS is used to estimate the hardware resources required for function units.

Finally, allocation is used to estimate the hardware resources required for interconnection of function units. The work of allocation is divided into 3 parts: register allocation, operation assignment, and interconnection binding. In register allocation, each variable is assigned to a certain register. In operation assignment, each operator is assigned to a certain function unit. Both are solved using the weighted-bipartite algorithm, and the common objective is that each assignment should introduce the least number of interconnection units that will be determined in the last phase, the interconnection binding. In this approach, multiplexer is the only type of interconnection unit, which eases the work of interconnection binding. The number and type of multiplexersis are determined by simply counting the number of different inputs to each register and each function unit. After allocation, the clock frequency is determined by searching for the longest path between two registers. Because routing resources are not modeled, the delay takes into account only the function units and the multiplexers.

We assume that all variables have the same size, since our goal is to quickly produce estimates with pure ANSI-C code instead of generating optimal synthesizable RTL code, which often uses some kinds of subset C code and applies special meanings to variables. Our estimation framework also supports to explore parallelism for loops. This is done at the SUIF-level, where we provide a module that allows designers to perform loop unrolling (loops must have fixed number of iterations) and loop merging (loops must have the same number of iterations).

4.1.2. Instantiation for the case study

For the case study, we started with C-representation of the system. It contained a main control function and the four computational tasks, which lead to a simple system partition that the control function was mapped onto SW and the rest onto RTR hardware. The estimation tool was used first to produce the resource estimates. The results are listed in Table 1, where LUT stands for lookup table and register refers to word-wide storages. The multiplexer refers to the hard-wired 18×18 bits multipliers embedded in the target FPGA.

Based on the resource estimates, the dynamic context partitioning was done as follows. The channel estimator was

Table 1: Estimates of FPGA resources required by the function blocks.

Functions	LUT	Multiplier	Register
Adaptive filter	1078	8	91
Channel estimator	1387	0	84
Combiner	463	4	32
Correlator	287	0	17
Total	3215	12	224

assigned to one context (1387 LUTs), and the other three processing blocks were assigned to a second context (1078 + 463 + 287 = 1828 LUTs). This partition resulted in both balanced resource utilization and less interface complexity compared to other alternatives. In implementation phase, both contexts are mapped onto the same region of the target FPGA, and the system dynamically loads the one that is needed.

4.2. Modeling of DRHW and the supporting transformation tool

The modeling of reconfiguration overhead is divided into two steps. In the first step, different technology-dependent features are mapped onto a set of parameters, which are the size of the configuration data, the clock speed of configuration process, and the extra delays apart from the loading of the configuration data. Thus, by tuning the parameters, designers can easily evaluate the tradeoffs between different technologies without going into implementation details. In the second step, a parameterized SystemC module that models the behavior of run-time reconfiguration process is created. It has the ability to automatically capture the reconfiguration request and present the reconfiguration overhead during performance simulation. Thus, designers can easily evaluate the tradeoffs between different technologies by tuning the parameters.

4.2.1. DRHW modeling approach

We model DRHW in such a way that the component can automatically capture reconfiguration requests during simulation and trigger reconfigurations when needed. In addition, a tool to automate the process that replaces some existing SystemC models by a DRHW SystemC model is developed, so system designers can easily perform the test-and-try and thus the design space exploration process is easier. In order to let the DRHW component be able to capture and understand incoming messages, SystemC modules must implement predefined but interface methods, such as *read()*, *write()*, *get_low_addr()*, and *get_high_addr()*. With the forthcoming SystemC TLM 2.0 standard [25], new interface methods could be defined to comply with the TLM 2.0. Equivalently, OCP standard transaction-level interfaces [26] can be used.

A general model of RSoC is shown in Figure 5. The left-hand side depicts the architecture of the RSoC. The right-hand side shows the internal structure of the DRHW component. It is a single hierarchical SystemC module, which

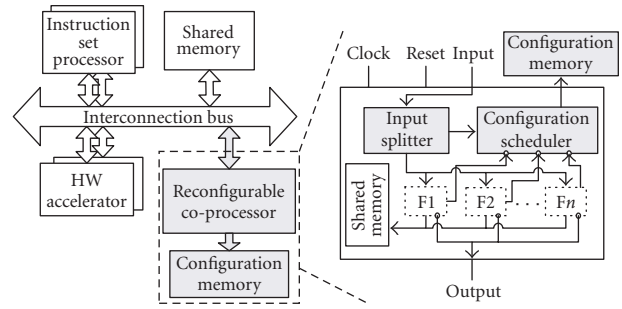
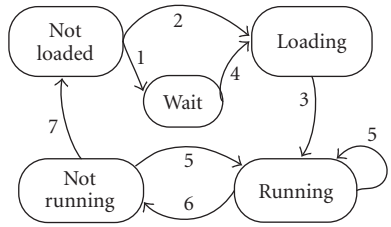


Figure 5: A generic model of RSoC.

implements the same bus interfaces as other HW/SW modules do. A configuration memory is modeled, which could be an on-chip or off-chip memory that holds the configuration data. Functions mapped onto DRHW (*F1* to *Fn*) are individual SystemC modules that implement the predefined bus interfaces with separate system address space. The input splitter (IS) is an address decoder and it manages all incoming interface-method-calls (IMCs). The configuration scheduler (CS) monitors the status of the mapped function and controls reconfiguration processes.

The DRHW component works as follows. When the IS captures an IMC, it will hold the IMC and pass the control to the CS, which decides if reconfiguration is needed. If so and the CS detects the DRHW is free to use, it issues a reconfiguration that uses the technology-dependent parameters to generate the memory traffic and the associated delays to mimic the reconfiguration latency. If the CS detects the DRHW is loaded with another module, a request to reconfigure the target module will be put into a FIFO queue and the reconfiguration will be started after the DRHW has no running module. After finishing the reconfiguration, the IS will dispatch the IMC to the target module. This is a generic description of the context switching process, and designers can develop different CS models when different types of RTR hardware are used such as partial reconfiguration or multicontext device. In our approach, context switching with preemption is not supported because of its high implementation cost in DRHW.

There is a state diagram common to each of the mapped function modules. Based on the state information, the CS makes reconfiguration decisions for all incoming IMCs and DONE signals. A state diagram of partial reconfiguration is presented in Figure 6. For single context and multicontext reconfigurable resources, similar state diagrams can be used in the model. In fact, different techniques for reducing the effect of the configuration latency can be applied, for example, configuration prefetching [27]. The idea is to load a module before it is needed. In the state diagram, this can be achieved by enabling the branch 2 when the module is known to be executed soon, so the module can be loaded before an IMC to it is issued. However, prefetching conditions should be decided at design time and stored in a table, which can be accessed by the CS at run-time.



State definitions:

Not loaded: module is only in the configuration memory
Loading: module is being loaded
Wait: module is waiting in a FIFO queue to be loaded
Running: module is running
Not running: module is loaded, but not running

State transition conditions ()* for configuration prefetching

1. IMC to the module occurs & not enough resources
2. (IMC to the module occurs & enough resources)|
   (The module is to be used soon & enough resources)*
3. CS finishes the loading
4. Other modules finish & enough resources
5. IMC to the module occurs
6. Module finishes
7. CS flushes the module

Figure 6: State diagram of functions.

4.2.2. An automatic code transformer for DRHW component

In order to reduce the coding effort, we have developed a tool that can automatically transform SystemC modules of the function blocks into a DRHW component. The inputs are SystemC files of a static architecture and a script file, which specifies the names of the mapped functions and the associated design parameters such as configuration latency. The tool contains a hard-coded DRHW template. It first parses the input SystemC code to locate the declarations of the candidate components (the C++ parser is based on opencxx [28]). Then the tool creates a DRHW component by filling the DRHW template with the declarations and establishing appropriate connections between the CS, the IS, and the functions. Finally, in the top-level structure, original SystemC modules of the mapped functions are replaced with the generated DRHW component. During simulation, reconfigurations will be automatically monitored and saved in a text file for analysis. A value change dump (VCD) file will also be produced to visualize the reconfiguration effects.

4.2.3. Instantiation for the case study

For the case study, we first created a SystemC model of a fixed system, which had two purposes in the design. The first was to use its simulation results as reference data, so the data collected from the reconfigurable system could be evaluated. The second purpose was to automatically generate the reconfigurable system model from it via the transformation tool.

In the fixed system, each of the four WCDMA decoding functions was mapped to an individual hardware accelerator, and pipelined processing was used to increase the performance. A small system bus was modeled to connect all of the processing units and storage elements. The channel data used in the simulation was recorded in text files, and the processor drove a slave I/O module to read the data from the file. The SystemC models were described at transaction level, in which the workload was derived based on the estimation results but with manual adjustment. The results showed that 1.12 milliseconds were required for decoding all 2560 chips of a slot when the system was running at 100 MHz.

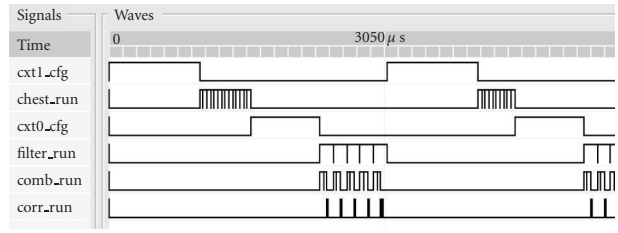


Figure 7: Simulation waveform shows the reconfiguration latency.

The transformation tool was then used to automatically generate the reconfigurable system model from the fixed model. The reconfiguration latency of the two dynamic contexts was derived based on the assumption that the size of the configuration data was proportional to the resource utilization, the number of LUTs required. The total available LUTs and size of full bitstream were taken from the Xilinx XC2VP20 datasheet. In the future, accurate approaches to derive the reconfiguration latency will be studied.

The performance simulation showed that the system required two reconfigurations per processing one slot of data. This is presented by the cxt0_cfg and cxt1_cfg in Figure 7. When the configuration clock was running at 33 MHz and the configuration bit width was 16, the reconfiguration latency was 2.73 milliseconds and the solution was capable of processing 3 slots of data in a frame.

5. LOW-LEVEL IMPLEMENTATION

The task at low-level implementation is to generate C code for SW, RTL-VHDL code for HW, and further generate executable binary code and netlist. For the HW part, there are commercial high-level synthesis tools that could be used to reduce the design time. However, considering the cost of such tools and the fact that the four WCDMA decoding functions can be implemented straightforward, we manually generated synthesizable RTL code for HW implementation. Simulation of the reconfigurable system was also performed at the RTL level by using the dynamic circuit switching (DCS)-based technique [29]. Multiplexers and selectors are inserted after the outputs of the modules and before the inputs of the modules. They are automatically switched on or off according to the configuration status. In the cycle-accurate simulation model, the reconfiguration is modeled as pure delay. For implementing the RTR, technology-dependent tools were used. Reconfigurations are triggered and managed by the main controlling SW task. The reconfiguration is implemented using the SystemACE compact flash (CF) solution

and the configuration data is stored in a CF card. A simple device driver that controls the SystemACE module is developed and the reconfiguration request is implemented as function calls to the SystemACE device driver.

5.1. Detailed design and implementation

In the low-level design phase, the main controlling SW task is mapped onto the embedded PowerPC core in the target FPGA, and the data memories are mapped onto the embedded block memories. Other components are mapped onto Xilinx IP cores, if corresponding matches can be found, for example,the bus model to the Xilinx processor local bus (PLB) IP core. In addition to the basic functionality, we added a few peripherals for debugging and visualization. Vendor-specific tools were used in the system refinement and implementation phases. Other than the traditional design steps for HW/SW implementation, additional steps for interface refinement, configuration design,and partially reconfigurable module (PRM) design were needed.

5.2. Interface refinement

The number of signals crossing the dynamic region and the static region must be fixed, since the dynamic region cannot adapt itself for changing the number of wires. In our work, the step to define the common set of boundary signals shared between the PRMs is referred to as interface refinement. In Xilinx FPGAs, the boundary signals are implemented as bus macros [30], which are prerouted hard macros used to specify the exact routing channels and will not change when modules are reconfigured. Because each bus macro is defined to hold 4 signals and there are only a limited number of bus macros, the boundary signals cannot be oversized. Therefore, it is more beneficial to minimize the number of signals crossing the dynamic region and the static region, which can also relax the constraint during placement and routing. In this case study, the number of boundary signals is reduced to 82, which correspond to the signals connected to the two 16-bit dual-port data memories and the PLB bus adapter. 21 bus macros are needed.

5.3. Partial reconfigurable module design

Synthesis results of the four functions are listed in Table 2. When considering the estimation, the results are overestimated at about 55% in average. The main reasons arethat: (1) the estimator assumes fixed-length computation for all variables, (2) the estimator maps all multiplexers directly to LUTs but real synthesis tools usually utilize the internal multiplexers in individual logic elements. For the PRM, the Xilinx module-based partial reconfiguration design flow [30] was used. First, each of the four detector functions was implemented as a single block. Then a context wrapper that matches the boundary signals was generated to wrap the channel estimator as one module and the other three as another module. The static part was assigned to the right side of the FPGA, because most used IO pads were in the right side. The dynamic region was in the left side of the FPGA.

Table 2: HW synthesis results.

Functions	LUT	Multiplier	Register (bits)
Adaptive filter	553	8	1457
Channel estimator	920	0	2078
Combiner	364	4	346
Correlator	239	0	92

The size of the configuration data was 279 KB for the context 1 and 280 KB for the context 2.

Routed PRMs on the dynamic region are shown in Figure 8. The context 1 that contains the channel estimator is shown in Figure 8(a), and the context 2 that contains the other three modules is shown in Figure 8(b). In addition, a routed design after module assembly is shown in Figure 8(c), which is the integration of the context 2 and the static part. The bus macros that are used for providing reliable connections for the boundary signals are marked by the block in the middle.

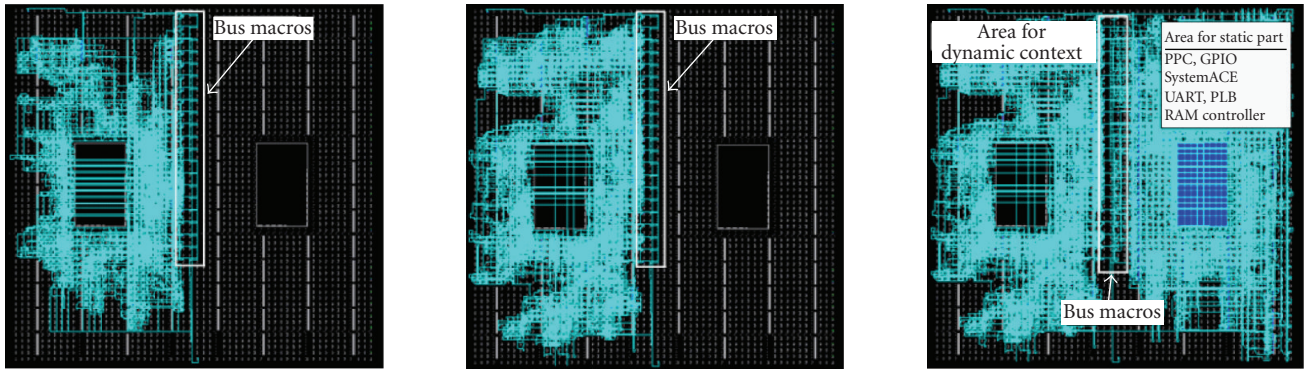
5.4. Comparison with other approaches

In addition to the RTR implementation, a fixed hardware implementation and a pure software implementation were made as reference designs. In the fixed-hardware implementation, the processing blocks were mapped onto static accelerators and the scheduling task was mapped onto SW that ran on the PowerPC core. The resource requirements were 4632 LUTs (24% of available resources), 55 Block RAMs (62%) and 12 Block Multipliers (13%). The system was running at 100 MHz. The execution time for processing one slot of data was 1.06 ms. Compared to the fixed reference system, the dynamic approach achieved over 40% resource reduction in terms of the number of LUTs, but at the cost of 8 times longer processing time.

For the full software implementation, the design was done as a standalone approach and no operating system was involved. Everything was running in a single PPC core and data were entirely stored in internal Block RAMs. For the same clock frequency, the processing time of one slot of data was 294.6 milliseconds, which was over 30 times of the processing time in run-time reconfiguration case. This did not fulfill the real-time requirements.

6. CONCLUSIONS

The main advantage of RTR systems is the combined flexibility and performance. However, implementing RTR does require extra efforts in the various design stages, from the abstract system-level down to the timing-accurate circuit-level. In this work, we present a design flow instantiation for RTR systems. It combines design supports at system level for design partitioning and system modeling to evaluate the effect of reconfiguration overhead. In implementation level, commercial and technology-dependent tools are applied. A set of wireless communication functions is used in the case study. Compared to a completely fixed implementation, the reduction of LUTs is over 40%. Compared to a full software



(a) Context: channel estimator (b) Context: adaptive filter, combiner, correlerator (c) An integrated design

Figure 8: Routed design of PRM on the dynamic region.

implementation, the run-time reconfiguration approach is over 30 times faster. The commercial off-the-shelf FPGA platform caused limitations on the implementation of run-time reconfiguration. Although the selected approach used partial reconfiguration, the required configuration time affected the performance a lot in the data-flow type WCDMA design case. The ratio of computing to configuration time was about 1/8 in this design case. The results clearly show that the configuration overhead is nowadays the main bottleneck of RTR systems. In the future, techniques to reduce its effect will be studied. In addition, improvements of our system-level design supporting tools are needed, such as power analysis and more accurate HW resource estimation approach, which will be studied.

ACKNOWLEDGMENTS

This work was previously supported by the European Commission under the Contract IST-2000-30049 ADRIATIC, and later by Tekes (National Technology Agency of Finland) and VTT under EUREKA/ITEA Contract 04006 MARTES.

REFERENCES

[1] Xilinx, "Virtex platform datasheet," May 2007, http://www.xilinx.com.
[2] Altera, "Stratix platform datasheet," May 2007, http://www.altera.com.
[3] PACT XPP technologies, "XPP performance media processor datasheet," May 2007, http://www.pactxpp.com.
[4] QuickSilver Technologies, "Adapt2000 ACM system platform overview," May 2007, http://www.qstech.com.
[5] Triscend, "A7 field configurable system-on-chip datasheets," 2004, www.triscend.com.
[6] Motorola, "Press release of MRC6011 RCF device," 2003, http://www.motorola.com.
[7] Y. Qu, K. Tiensyrjä, and K. Masselos, "System-level modeling of dynamically reconfigurable co-processors," in *Proceedings of the 14th International Conference on FPL*, vol. 3203 of *Lecture Notes in Computer Science*, pp. 881–885, Tampere, Finland, August 2004.
[8] Y. Qu, K. Tiensyrjä, and J.-P. Soininen, "SystemC-based design methodology for reconfigurable system-on-chip," in *Proceedings of the 8th Euromicro Conference on Digital System Design (DSD '05)*, vol. 2005, pp. 364–371, Porto, Portugal, August 2005.
[9] J. Noguera and R. M. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 399–415, 2002.
[10] J. Harkin, T. M. McGinnity, and L. P. Maguire, "Partitioning methodology for dynamically reconfigurable embedded systems," *IEE Proceedings: Computers and Digital Techniques*, vol. 147, no. 6, pp. 391–396, 2000.
[11] F. Berthelot, F. Nouvel, and D. Houzet, "Design methodology for runtime reconfigurable FPGA: from high level specification down to implementation," in *Proceedings of IEEE Workshop on Signal Processing Systems (SiPS '05)*, vol. 2005, pp. 497–502, Athens, Greece, November 2005.
[12] M. Handa, R. Radhakrishnan, M. Mukherjee, and R. Vemuri, "A fast macro based compilation methodology for partially reconfigurable FPGA designs," in *Proceedings of the 16th International Conference on VLSI Design*, pp. 91–96, New Delhi, India, January 2003.
[13] K. S. Chatta and R. Vemuri, "Hardware-software codesign for dynamically reconfigurable architectures," in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL '99)*, pp. 175–184, Glasgow, UK, August-Septembe 1999.
[14] L. Shang and N. K. Jha, "Hardware-software cosynthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs," in *Proceedings of the 7th Asia and South Pacific and the 15th International Conference on VLSI Design Automation Conference (ASP-DAC '02)*, pp. 345–352, Bangalore, India, January 2002.
[15] L. Shang, R. P. Dick, and N. K. Jha, "SLOPES: Hardware-software cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 3, pp. 508–525, 2007.
[16] A. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. U. K. Melcher, "Modeling and simulation of dynamical and partially reconfigurable systems using SystemC," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, pp. 35–40, Porto Alegre, Brazil, March 2007.

[17] Z. Li, *Configuration management techniques for reconfigurable computing*, Ph.D. thesis, Department of Electrical and Computer Engineering, Northwestern University, Evanston, Ill, USA, 2002.

[18] M. J. Heikkilä, "A novel blind adaptive algorithm for channel equalization in WCDMA downlink," in *Proceedings of the 12th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '01)*, vol. 1, pp. A41–A45, Diego, Calif, USA, September 2001.

[19] Memec, "VirtexIIpro demonstration board datasheet," 2003, http://www.memec.com.

[20] Y. Qu and J.-P. Soininen, "Estimating the utilization of embedded FPGA co-processor," in *Proceedings of the Euromicro Symposium on Digital Systems Design (DSD '03)*, pp. 214–221, Belek-Antalya, Turkey, September 2003.

[21] R. P. Wilson, R. S. French, C. S. Wilson, et al., "SUIF: an infrastructure for research on parallelizing and optimizing compilers," in *Proceedings of the 7th ACM SIGPLAN symposium on Principles and practice of parallel pro programming*, pp. 37–48, San Diego, Calif, USA, December 1994.

[22] Redhat, "gcov: the test coverage tool," May 2007, http://www.redhat.com.

[23] D. D. Gajski, et al., *High-level synthesis: Introduction to chip and system design*, Kluwer Academic Publishers, Boston, Mass, USA, 1997.

[24] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.

[25] The Open SystemC Initiative (OSCI), "The SystemC TLM 2.0 documentation," May 2007, http://www.systemc.org/home.

[26] OCP-IP, "OCP 2.2 Specification," February 2007, http://www.ocpip.org/home.

[27] S. Hauck, "Configuration prefetch for single context reconfigurable coprocessors," in *Proceedings of the 1998 ACM/SIGDA 6th International Symposium on Field Programmable Gate Arrays (FPGA '98)*, pp. 65–74, Monterey, Calif, USA, February 1998.

[28] S. Chiba, "OpenC++ reference manual," May 2007 http://opencxx.sourceforge.net.

[29] P. Lysaght and J. Stockwood, "A simulation tool for dynamically reconfigurable field programmable gate arrays," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 3, pp. 381–390, 1996.

[30] Xilinx, "Xilinx application note: XPP290 Two Flows for Partial Reconfiguration: Module-Based or Difference-Based," May 2007.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 793919, 18 pages
doi:10.1155/2008/793919

Research Article

A Flexible System Level Design Methodology Targeting Run-Time Reconfigurable FPGAs

Florent Berthelot,[1] Fabienne Nouvel,[1] and Dominique Houzet[2]

[1] *CNRS UMR 6164, IETR/INSA Rennes, 20 avenue des Buttes de Coesmes, 35043 Rennes, France*
[2] *GIPSA-Lab, INPG, 46 avenue Felix Viallet, 38031 Grenoble Cedex, France*

Correspondence should be addressed to Florent Berthelot, florent.berthelot@irisa.fr

Received 31 May 2007; Revised 8 October 2007; Accepted 20 November 2007

Recommended by Donatella Sciuto

Reconfigurable computing is certainly one of the most important emerging research topics on digital processing architectures over the last few years. The introduction of run-time reconfiguration (RTR) on FPGAs requires appropriate design flows and methodologies to fully exploit this new functionality. For that purpose, we present an automatic design generation methodology for heterogeneous architectures based on DSPs and FPGAs that ease and speed RTR implementation. We focus on how to take into account specificities of partially reconfigurable components from a high-level specification during the design generation steps. This method automatically generates designs for both fixed and partially reconfigurable parts of an FPGA with automatic management of the reconfiguration process. Furthermore, this automatic design generation enables a reconfiguration prefetching technique to minimize reconfiguration latency and buffer-merging techniques to minimize memory requirements of the generated design. This concept has been applied to different wireless access schemes, based on a combination of OFDM and CDMA techniques. This implementation example illustrates the benefits of the proposed design methodology.

1. INTRODUCTION

Applications such as multimedia, encryption, or wireless communication require highly repetitive parallel computations and lead to incorporate hardware components into the designs to meet stringent performance and power requirements. On the other hand, the architecture flexibility is the key point for developing new multistandard and multiapplication systems.

Application-specific integrated circuits (ASICs) are a partial solution for these needs. They can reach high performance, computation density, and power efficiency but to the detriment of architecture flexibility as the computation structure is fixed. Furthermore, the nonrecurring engineering costs (NREs) of ASICs have been increased dramatically and made them not feasible or desirable for all the applications especially for bugfixes, updates, and functionality evolutions.

Flexibility is the processor's architecture paradigm. The algorithm pattern is computed temporally and sequentially in the time by few execution units from a program instruction stream. This programmability potentially occurs at each clock cycle and is applied to a general computation structure with a limited computation parallelism capacity. The datapath can be programmed to store data towards fixed memories or register elements, but cannot be truly reconfigured. This kind of architecture suffers from the memory controller bottleneck and their power inefficiency. These architectures have a very coarse-grained reconfiguration, reported as *system level*.

Reconfigurable architectures can provide an efficient platform that satisfy the performance, flexibility, and power requirements of many embedded systems. These kinds of architectures are characterized by some specific features. They are based on a spatial computation scheme with high parallelism capacity distributed over the chip. Control of the operator behavior is distributed instead of being centralized by a global program memory. Multiple reconfigurable architectures have been developed [1, 2]. They can be classified by their reconfiguration granularity.

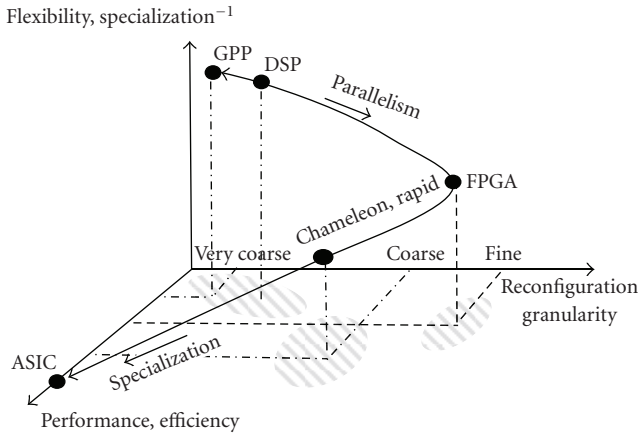


Figure 1: Architecture flexibility in regard with granularity and performance.

FPGAs have a logic level of reconfiguration. Communication networks and functional units are bit-level configurable.Their highly flexible structure allows to implement almost any application. A volatile configuration memory allows to configure the datapath and the array of configurable logic blocks.

Some reconfigurable architectures are based on coarse-grained arithmetic units of larger sizes such as 32 bits for algorithms needing word-width data paths, such as DART [3], Chameleon [4], Rapid [5], or KressArray [6]. They use a functional level of reconfiguration allowing to reach a greater power efficiency and computational density. Moreover, the amount of configuration data needed is limited. But coarse-grained architectures target domain-specific applications.

Figure 1 represents the architecture flexibility in regard with the granularity and performance of the above mentioned architectures. The architecture granularity elevation allows either specialization and performance or the improvement of architecture flexibility as for the processors.

The introduction of dynamically reconfigurable systems (DRSs) has opened up a new dimension of using chip area. Recently, run-time reconfiguration (RTR) of FPGA parts has led to the concept of virtual hardware [7].

RTR allows more sections of an application to be mapped into hardware through a hardware updating process. A larger part of an application can be accelerated in hardware in contrast to a software computation. Partial reconfiguration ability of recent FPGAs allows updating only a specified part of the chip while the other areas remain operational and unaffected by the reconfiguration. These systems have the potential to provide hardware with a flexibility similar to that of software, while leading to better performance. In [8] an extensive review of benefits of hardware reconfiguration in embedded systems is addressed. Some applications like software defined radios (SDR) [9] are from them. This emerging radio technology allows the development of multiband, multifunctional wireless devices [10].

However, cutting-edge applications require heterogeneous resources to efficiently deal with the large variety of signal and multimedia processing. This is achieved by mixing various processing granularities offered by general purpose processors (GPPs), digital signal processors (DSPs), and reconfigurable architectures. Methodologies based on a system-level design flow can handle these heterogeneous architectures. Our proposed design flow uses graphs to represent both the application and the architecture and aims at obtaining a near optimal scheduling and mapping of the application tasks on an heterogeneous architecture [11].

This paper presents our methodology allowing a fast integration and use of run-time reconfigurable components, especially FPGAs. This implies a convenient methodology and conception flow which allows a fast and easy integration of run-time reconfiguration onto applications and an automatic management of the reconfiguration process. The design space exploration will benefit from this improvement as operations of the application algorithm graph could be mapped either on DSP/GPP devices or statically/dynamically on FPGA devices thanks to run-time reconfiguration.

The rest of the paper is organized as follows. Following the introduction, Section 2 gives an overview of related approaches, especially those which deal with run-time reconfiguration from a high-level design specification. Section 3 introduces the system-level design methodology used in this work based on the CAD tool named SynDEx [12], which is used for the application partitioning stage of our methodology. Section 4 deals with the impact of run-time reconfiguration on this methodology and the way of modeling partially reconfigurable FPGAs from a system-level point of view. Section 5 then addresses the automatic design generation targeting run-time reconfigurable architectures. The overall design flow and the generated architecture layers are described. We detail the way to generate an automatic management of run-time reconfiguration, along with the steps to achieve design generation, memory minimization, and dynamic operator creation necessary for partial reconfiguration. These design flow and methodology have been applied in Section 6 for the implementation of a transmitter system for future wireless networks for 4G air interface. Two implementation examples, based on a network on chip (NoC) and a point to point communication scheme, are presented. Finally Section 7 provides a conclusion of this work and gives some indication of future works.

2. RELATED WORK

Numerous researches are focused on reconfigurable architectures and on the way to exploit efficiently their potentials. Higher level of abstraction, design space exploration, hardware/software partitioning, codesign, rapid prototyping, virtual component design and integration for heterogeneous architectures; all these topics are strong trends in architectures for digital signal processing.

This section reviews several relevant propositions in these fields with a special emphasis on frameworks which enable run-time reconfigurations of applications from a high-level design specification along with considerations of hardware-dependent steps allowing partial reconfiguration.

Silva and Ferreira [13] present a hardware framework for run-time reconfiguration. The proposed architecture is

based on a general-purpose CPU which is tightly connected to a dynamically reconfigurable fabric (DRF). A tool named BitLinker allows the relocation and assembly of bitstreams of individual components and is used to automate these steps which are very dependent on the underlying hardware's organization. This approach is architecture-centered and the application mapping steps on this specific platform are not addressed.

This issue requires a higher level of abstraction for the design specification. PaDReH [14] is a framework for the design and implementation of run-time reconfigurable systems and deals only with the DRS hardware design flow. The use of SystemC language enables a higher level abstraction for the design and validation. After a translation to the RTL level a space-time scheduling and hardware partitioning is performed allowing the generation of a run-time reconfiguration controller. Bitstreams generation is based on Xilinx [15] modular design flow. In [16], a technique based on some modifications of the SystemC kernel is presented. It allows to model and simulate partial and dynamic reconfiguration. The acquired results can assist in the choice of the best cost/benefit tradeoff regarding FPGA chip area.

Craven and Athanas [17] present a high-level synthesis (HLS) framework to create HDL. The hardware/software partitioning is performed by the designer. Reconfiguration simulations and reconfiguration controller generation is allowed from a modified version of the Impulse C ANSI C-based design language supporting HLS from C to RTL HDL. Nevertheless, the final implementation steps are not addressed.

The EPICURE project [18] is a more comprehensive methodology framework based on an abstraction tool that estimates the implementation characteristics from a C-level description of a task and a partitioning refinement tool that realizes the dynamic allocation and the scheduling of tasks according to the available resources in the dynamically reconfigurable processing unit (DRPU). An intelligent interface (ICURE) between the software unit and the DRPU acts as a hardware abstraction layer and manages the reconfiguration process. Overlapping between computations and reconfigurations is not supported.

In [19], Rammig and Al present an interesting tool-assisted design space exploration approach for systems containing dynamically hardware reconfigurable resources. A SystemC model describes the application, while an architecture graph models the architecture template. Next a multi-objective evolutionary algorithm is used to perform automatic system-level design space exploration. A hierarchical architecture graph is used to model the mutual exclusion of different configurations.

Most of these above methodologie frameworks assume a model of external configuration control, mandating the use of a host processor or are tightly coupled to a specific architecture [13, 18]. Hence custom heterogeneous architectures are not supported. Few of them address methods for specifying run-time reconfiguration from a high-level down to the consideration of specific features of partially reconfigurable components for the implementation.

Our proposed methodology deals with heterogeneous architectures composed of processors, FPGA or any specific circuits. The implementation of run-time reconfiguration on hardware components, especially FPGAs, is automated and eased by the use of a high-level application and architecture specification. This step is handled by the SynDEx tool which allows the definition of both the application and the hardware from a high level and realizes an automated Hardware/software mapping and scheduling. Run-time reconfiguration and overall design control scheme are independent of any specific architecture.

3. GENERAL METHODOLOGY FRAMEWORK

Some partitioning methodologies based on various approaches are reported in the literature [11, 20, 21].

They are characterized by the granularity level of the partitioning, targeted hardware, run-time-reconfiguration support, on-line/off-line scheduling policies, HW-SW relocation [22], reconfiguration prefetching technique [23] and flow automation.

As discussed in the previous section, few tools based on these partitioning methodologies provide a seamless flow from the specification down to the implementation.

To address these issues, this work uses the SynDEx CAD tool for the high-level steps of our methodology, which include automatic design partitioning/scheduling and RTL code generation for FPGA.

3.1. AAA/SynDEx presentation

Our methodology aims at finding the best matching between an algorithm and an architecture while satisfying constraints. AAA is an acronym for adequation algorithm architecture, adequation is a French word meaning efficient matching. AAA methodology is supported by SynDEx, an academic system-level CAD tool and is used in many research projects [24, 25]. AAA methodology aims at considering simultaneously architecture and application algorithm, both are described by graphs, to result in an optimized implementation.

The matching step consists in performing a mapping and a scheduling of the algorithm operations and data transfers onto the architecture processing components and the communication media. It is carried out by a heuristic which takes into account durations of computations and inter-component communications to optimize the global application latency. Operation durations and inter-component communications are taken into account for that purpose.

3.1.1. Application algorithm graph

Application algorithm is represented by a dataflow graph (DFG) to exhibit the potential parallelism between operations. Figure 2 presents a simple example. The algorithm model is a direct data dependence graph. An operation is executed as soon as its inputs are available, and this DFG is infinitely repeated. SynDEx includes a hierarchical algorithm representation, conditional statements and iterations of algorithm parts. The application can be described in

a hierarchical way by the algorithm graph. The lowest hierarchical level is always composed of indivisible operations (C2, F1, and F2). Operations are composed of several input and output ports. Special inputs are used to create conditional statements. Hence an alternative subgraph is selected for execution according to the conditional entry value. As illustrated in Figure 2, the conditional entry "X" of "C1" operation represents such a possibility. Data dependencies between operations are represented by valued arcs. Each input and output port has to be defined with its length and data type. These lengths are used to express either the total required data amount needed by the operation before starting its computation or the total amount of data generated by the operation on each output port.

3.1.2. Architecture graph

The architecture is also modeled by a graph, which is a directed graph where the vertices are computation operators (e.g., processors, DSP, FPGA) or media (e.g., OCB busses, ethernet) and the edges are connections between them. So the architecture structure exhibits the actual parallelism between operators. Computation vertices have no internal computation parallelism available. An example is shown in Figure 3. In order to perform the graph matching process, computation vertices have to be characterized with algorithm operation execution times. Execution times are determined during the profiling process of the operation. The media are also characterized with the time needed to transmit a given data type.

3.1.3. AAA results

The result of the graph matching process is a mapping and scheduling of the application algorithm over the architecture. These informations are detailed by the automatically generated files called "macrocodes." Macrocodes are generated for each computation vertex of the architecture graph. Figure 4 shows a possible macrocode file which describes the behavior of the vertex named "FPGA1" when only the "C2" operation is mapped on. The macrocode is a high-level language made of hardware independent primitives. Macrocodes are always composed of several threads. There is one thread for the computation controls where operations are called, like the C2 operation with input and output buffers.

There is one thread for each communication port of the operator as modeled in the architecture graph. Communication threads can receive and send data thanks to the "Recv_" and "Send_" primitives. The primitives "Suc1_," "Suc0_," "Pre0_," "Pre1_" manipulate the semaphores to ensure mutual exclusions and synchronizations between threads.

There are also macros for memory allocations. They define depth and data type of buffers used to store the computation results of the operator (number of generated events on the arc defined in the application algorithm graph). The set of generated macrocode files represents a distributed synchronized executive. This executive is dedicated to the application and can be compared to an offline static operating system.

Figure 5 depicts the overall methodology flow. Each macrocode is translated toward a high-level language (HDL or C/C++) for each HW and SW component. This translation produces an automatic dead-lock free code. The macrocode directives are replaced by their corresponding code from libraries (C/C++ for software components, VHDL for hardware components). Many libraries have been developed for heterogeneous platforms and we present how we extend SynDEx capacities to handle runtime reconfigurable components.

4. RUN-TIME RECONFIGURATION CONSIDERATIONS

4.1. Architecture graph modeling of run-time reconfigurable components

In our methodology, the algorithm graph application description is realized at the functional level representing coarse-grained operations. This description is handled during the design flow through the use of libraries containing IP definitions for code generation. These coarse-grained IPs are supposed to be developed to fully exploit parallelism of their final implementation targets. This step is basically achieved by the compiler for software components or the synthesizer for hardware components. Hence, the logical-level architecture granularity of FPGA is exploited by the RTL-synthesizer to generate IP netlists.

With this high-level modeling, static components (which do not provide reconfiguration support) and run-time reconfigurable components are modeled by vertices in the architecture graph. We have defined two kinds of vertices to match reconfiguration support of hardware components: the *static vertices* and the *run-time reconfigurable vertices*. Run-time reconfigurable vertices can be considered as an extension of static vertices in term of functional flexibility as their operations can be changed dynamically.

As the last FPGA generations have reached high densities, it can be advantageous to divide them in several independent parts for architecture modeling. Example (b) in the left side of Figure 6 presents the modeling of an FPGA divided in five parts. One is considered as static (F1), two represent embedded microprocessors (P1, P2), and the last two represent two partially reconfigurable areas (D1, D2). We can also see the architecture graph modeling of this FPGA made with several vertices under SynDEx. The internal links named CC1, IL1, IL2 and IL3 represent data communications between these sub-FPGA parts. The modeling of a prototyping platform (case (a), Figure 6) composed of two FPGAs, one CPLD and a processor, can be represented with the same architecture graph. Only the characterization of the communication media differs.

The top of Figure 6 shows a set of six operations composing an algorithm graph. Once the mapping process is done, these operations are affected to the architecture graph vertices. All operations mapped to a same vertex are sequentially executed at run-time. Hence we have defined three kinds of operators for the operation executions on the hardware. The first is a *static operator* which can not be reconfigurated, so dedicated to one operation. The two others, named

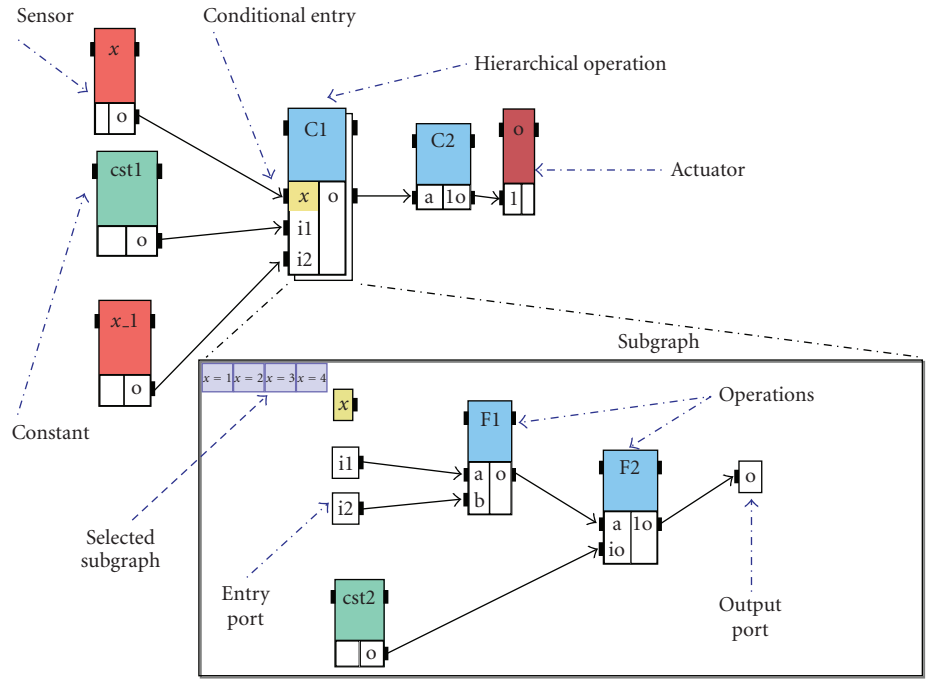


Figure 2: Algorithm graph.

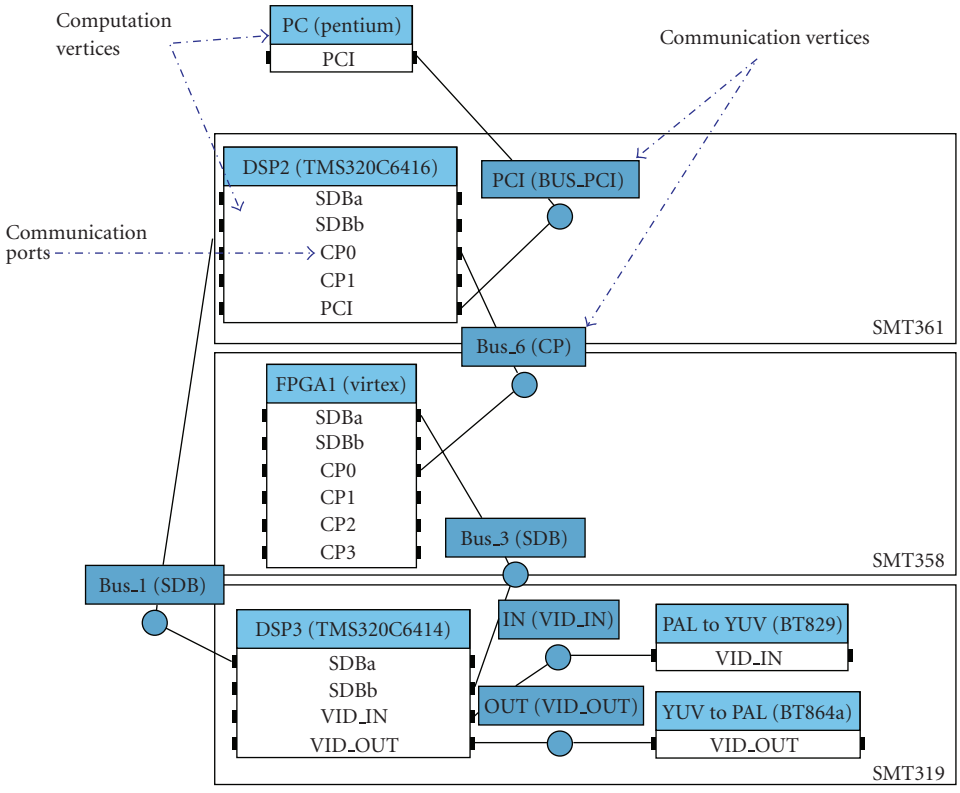


Figure 3: Architecture graph of a prototyping platform.

```
Include (syndex.m4x)
Processor_(Virtex, FPGA1)
 Semaphores_(                          }
 mem1_empty, mem1_full,                } Semaphore
 mem2_empty, mem2_full,                } allocation
            )                          }
Alloc_(char, mem1, 92)                 }   Memory allocation
Alloc_(char, mem2, 92)                 }  (data type, name, depth)
Thread_(CP, Bus6)
      Loop_
          Suc1_(mem1_empty)            }
          Recv_(mem1 , bus6)           }
          Pre0_(mem1_full)             }  Communication
          Suc1_(mem2_full)             }       thread
          Send_(mem2 , bus6)           }
          Pre0_(mem2_empty)            }
Endloop_
Endthread_
Main_
Loop_
          Suc0_(mem1_full)             }
          C2(mem1, mem2)               }  Computation
          Pre1_(mem1_empty)            }      thread
          Pre1_(mem2_full)             }
Endloop_
Endmain_

Endprocessor_
```

Figure 4: Simple SynDEx macrocode example generated for a computation vertex.

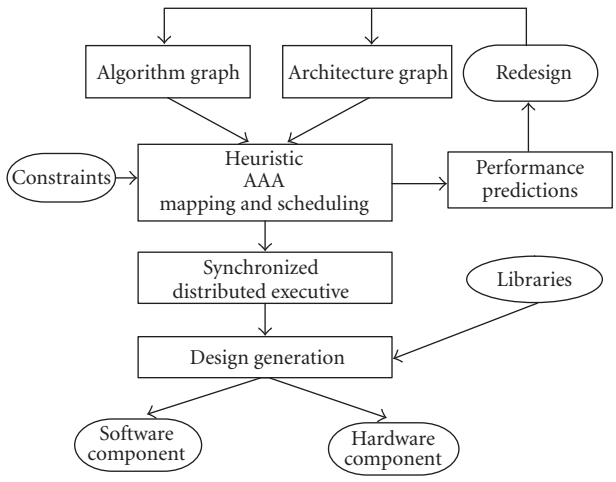


Figure 5: SynDEx methodology flow.

parameterizable operator and *dynamic operator*, are able to dynamically change their functionalities to match a given algorithm graph operation. They differ by their ways to implement a new operation during the runtime.

We define "*dynamic operator*" as a special operator which is managed to implement sequentially new operations through run-time reconfiguration. On current FPGAs it can be achieved thanks to partial reconfiguration technology and is mainly dependent on the specific architecture features. The *Parameterizable operator* is appropriate to enable fast operation switching through a parameter selection step.

Unlike to static vertex, a reconfigurable vertex can implement some static operators, some parameterizable operators and some dynamic operators. As a result, additional functionalities to control run-time reconfigurations in order to implement dynamically new functionalities, are necessary.

By assigning parts of the design from a high level, we can make run-time reconfiguration specification more flexible and independent of the application coding style. Moreover that allows to keep this methodology flow independent of the final implementation of the run-time reconfiguration process which is device dependent and tightly coupled to backend tools. This modeling can be applied on various components of different granularities.

4.2. Operation selection criterions for a dynamic implementation

Selection of operations for a dynamic implementation is a multicriteria problem. Reconfiguration can be used as a mean to increase functionality density of the architecture or for flexibility considerations. However some general rules can be brought out. The boundary between a fixed or a dynamic implementation is depending on the architecture ability to provide a fast reconfiguration process. This point is generally linked to the reconfiguration granularity of the architecture. When a set of common coarse grain operators are used for the execution of several operations, the use of a parameterized structure could be a more efficient solution and could allow fast inter-operation switchings [26]. That is achieved by increasing the operator granularity. As a result the amount of data configurations needed is decreased.

Parametrization and partial reconfiguration concepts are closely linked. In both cases the aim is to change the functionality, only the amount of configuration data and their targets differ. From the manager point of view both operators share a same behavior.

The choice of implementation style (static, partial reconfiguration, parameterized structure) of operations can be based on the ratio between reconfiguration time of operator, computation time and activity rate. A good mapping and scheduling is a tradeoff to avoid that reconfigurable surfaces remain unused or under exploited.

To illustrate this, Figure 7 represents a set of eight operations composing an application. These operations can be ordered in a diagram. Operations which are placed closely have a high degree of commonality and nearly the same complexity (logical resources). Hence a parameterized operator can be created, as for the case of operations B, C, and D. This parameterized operator does not have any configuration latency. Operations A, E, F, and H have nearly the same degree of complexity but they doesn't share a great amount of commonality and have not a high activity rate. In this case they can be implemented by a dynamic operator with run-time reconfiguration. Operation G has a high activity rate, it is desirable to implement it as a static operator.

The number of operations implemented on a same dynamic operator has a direct impact on the implementation

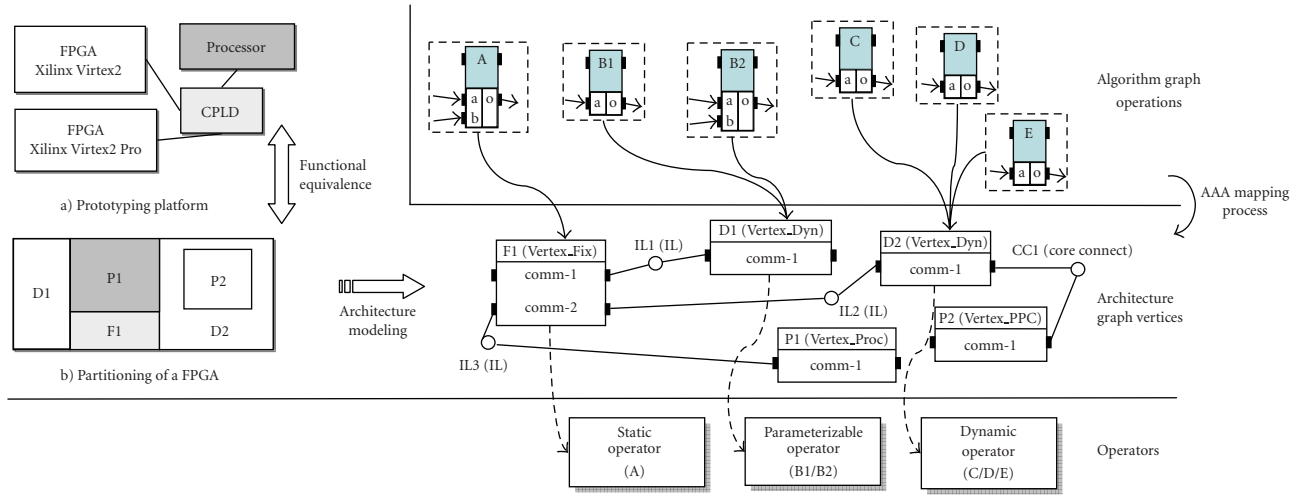


Figure 6: Architecture graph example and vertex operations implementation style.

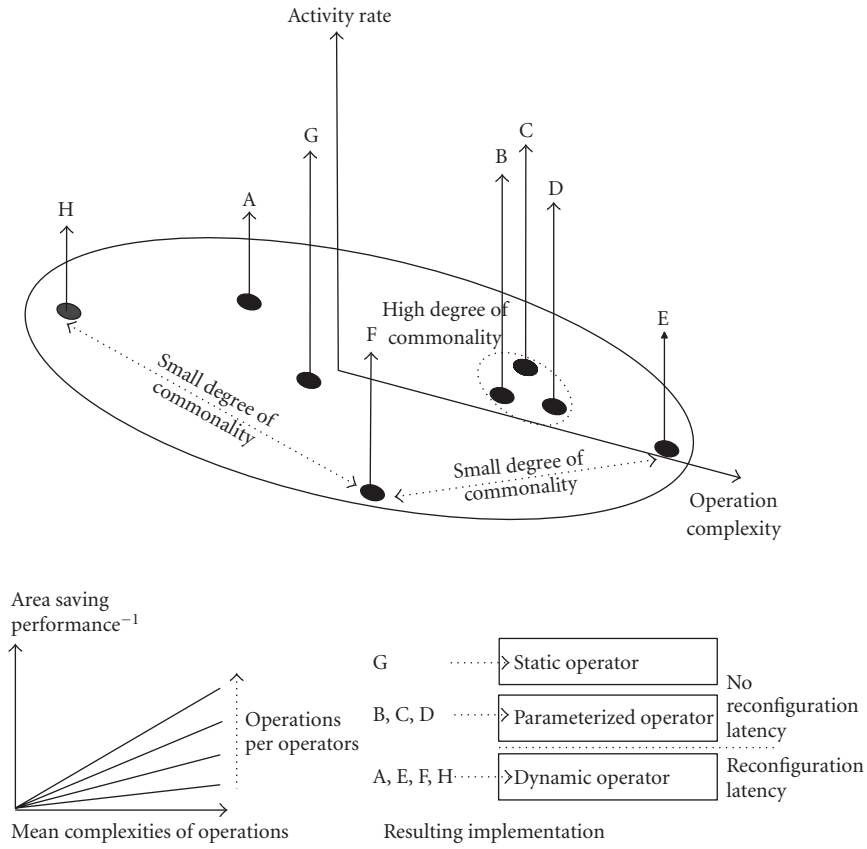


Figure 7: Choice and influence of the implementation style of operations.

area and the performance. The extreme cases are those when all the operations are implemented through only one dynamic operator or on the contrary when each operations are static and consume more resources. The influence of the number of operations per operator over the saving space is decreasing when the mean complexity of the operations are decreasing. So, we can consider that dynamic reconfiguration is useful when applied to medium and complex operations which do not have high activity rates, thus avoiding to waste too much time in reconfiguration. On the other hand, operations of small complexities can remain static or parameterized.

As the heuristic does not yet support run-time reconfigurable vertices (reconfiguration latencies are not taken into

account), the operations of the algorithm graph have to be characterized to define their way of implementation (static or dynamic). It is achieved by a manual mapping constraint from the designer through SynDEx tool.

We plan to automate this step thanks to the extraction of some metrics from the algorithm graph such as the operation complexity, activity rate, and relative commonality. These metrics could be used to guide the designer in the static-dynamic functionality implementation tradeoff.

4.3. Minimizing the reconfiguration cost

Run-time partially reconfigurable components must be handled with a special processing during the graph matching process. A major drawback of using run-time reconfiguration is the significant delay of hardware configuration. The total runtime of an application includes the execution delay of each task on the hardware combined with the total time spent for hardware reconfiguration between computations. The length of the reconfiguration is proportional to the reprogrammed area on the chip. Partial reconfiguration allows to change only a specified part of the design hence decreasing the reconfiguration time. An efficient way to minimize reconfiguration overhead is to overlap it as much as possible with the execution of other operations on other components. It is known as configuration prefetching techniques [27].

Therefore the heuristic of SynDEx has to be improved to take into account reconfiguration overhead with configuration prefetching techniques to minimize it. A solution based on a genetic heuristic is under development to integrate more metrics during the graph matching process. This study applied on run-time reconfiguration is presented in [28]. However the architectural model considered is based on a processor and a single-context partially reconfigurable device which acts as a coprocessor. The ideal graph matching process is a completely automated selection of operation implementation styles (fixed, dynamic, parameterized) while satisfying multiple constraints.

5. AUTOMATIC DESIGN GENERATION

5.1. General FPGA synthesis scheme

The overall design generation flow and hierarchical design architecture are shown in Figure 8. For each vertex of the architecture graph,after the mapping and scheduling processes, a synchronized executive represented by a macrocode is generated.

A list defines the implementation way of each algorithm graph function. The following cases are possible.

(i) For static vertices, only static operators are allowed.
(ii) For reconfigurable vertices, static operators, parameterizable operators, and dynamic operators can be implemented.

Macrocodes generated for these vertices must be handled by specific libraries during the design generation to include dynamic reconfiguration management. The design generation is based on the translation of macrocodes to the VHDL code. This translation uses the GNU macro processor M4 [29] and macros defined in libraries in addition to communication media and IP operators. Each macrocode is composed of one computation thread to control the sequencing of computations, and communication threads for each independent communication port. These threads are synchronized through semaphores. This high-level description has a direct equivalence in the generated design and corresponds to the highest design level which globally controls datapaths and communications. A lower level is in charge of resources management, typically in case of simultaneous accesses on buffers for computations or communications. The *configuration manager* process belongs to this design level. The next level brings together resource controls. Resources are operators, media and buffers. The *protocol builder* functionality, which is detailed later, has access to the underlying configuration memory.

A same operator can be reused many times over different data streams but only one instantiation of each kind of operator appears in VHDL. We have defined a standard interface for each operator through encapsulation, as described in the next section.

These points lead to build complex data paths automatically. Our libraries can perform these constructions by using conditional VHDL signal assignments. In next sub-sections, we deal with some important issues: optimization of memory for exchanged data through buffer merging, operators merging for dynamic instantiation and design generation for run-time reconfiguration management.

5.2. Macrocode preprocessing for memory minimization

SynDEx is based on a synchronous dataflow model of computation, processes communicating through buffers and reading/writing a fixed number of tokens each time they are activated. Hence this kind of representation allows to determine completely the scheduling at compile-time. Applications often have many possible schedules. This flexibility can be used to find solutions with reduced memory requirements or to optimize the overall computation time. Memory minimization is essential for an implementation on reconfigurable hardware devices which have limited on-chip memory capabilities.

Currently only the latency optimization is taken into account during the matching step of SynDEx. Macrocodes are generated without memory minimization considerations. The latest SynDEx version operates a macrocode memory minimization analysis on the macrocodes. A graph coloring based heuristic is used to find which buffers have to be merged. Buffers allocated by SynDEx can be heterogeneous as they store computation results of operations working on various data depths and widths. The integrated memory minimization of SynDEx allows to merge buffers of various depth and same data widths. In the other cases a manual analysis must be made by the designer to merge buffers of different widths. Table 1 sums up the cases.

The result is a list of buffers which must be merged into a global one. Hence, global buffers are automatically generated

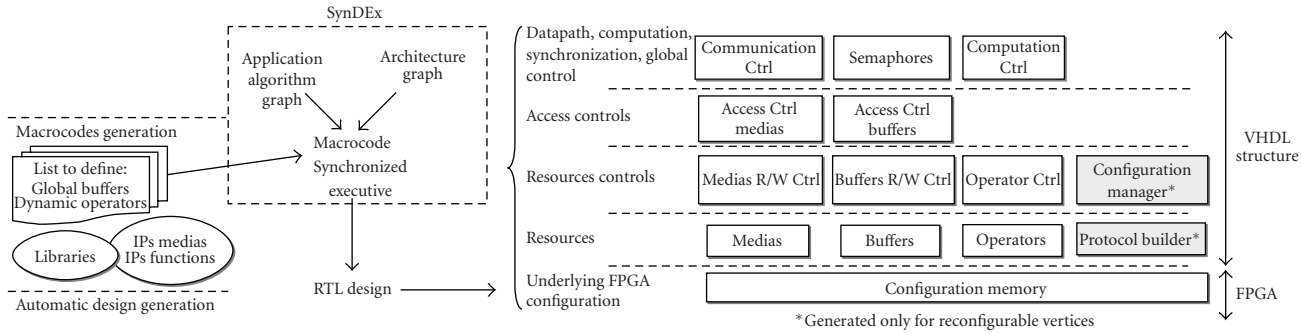


Figure 8: Hierarchical architecture design generation for hardware components.

Table 1: Buffer minimization cases.

Depth	Width	
	Same	Different*
Same	SynDEx	Manual
Different*	SynDEx	Manual

*Buffers widths must be multiple.

according to the operator data types. We have the following denotations:

(i) $L : \{b_1, b_2, \ldots, b_n\}$: *A list of n buffers which can be merged, where* $L_{(k)} = b_k$,

(ii) D_k: *the depth of buffer k.*

(iii) W_k: *the data width of buffer k.*

Hence the total amount of memory needed is

(i) without buffer merging: $M_{\overline{\text{bm}}} = \sum_{i=1}^{n} D_{L(i)^*} W_{L(i)}$,

(ii) with buffer merging: $M_{\text{bm}} = \max(D_{L(i)^*} W_{L(i)})$ for $1 \leq i \leq n$.

So, the amount of memory saved is

$$S_{\text{mem}} = M_{\overline{\text{bm}}} - M_{\text{bm}}. \quad (1)$$

Global buffers are automatically generated in VHDL. We have developed a parameterizable memory IP able to work on various depths and data widths and providing fast accesses. This IP is implemented either on dedicated hardware (i.e., Xilinx blockram) or on distributed on-chip FPGA memory (look-up tables).

5.3. Generic computation structure

After selection of candidates among all the operators of the algorithm graph for partial reconfiguration or parameterization, we obtain a set of operators that must be implemented into a run-time reconfigurable device. To achieve design generation, a generic computation structure is employed. This structure is based on buffer merging technique and functionality abstraction of operators. The aim is to obtain a single computation structure able to perform through run-time reconfiguration or parameterization the same functionalities as a static solution composed of several operators.

Encapsulation of operators through a standard interface allows us to obtain a generic interface access. This encapsulation eases the IP integration process with this design methodology and provides functionality abstraction of operators. This last point is helpful to easily manage reconfigurable operators with run-time reconfiguration or configuration for parameterized operators.

This encapsulation is suitable for coarse-grained operators with dataflow computation. It is a conventional interface composed of an input data stream, an output data stream along with "enable" and "ready" signals to control the computation. In the case of parameterized operators, a special input is used to select the configuration of the operator.

As operators can work on various data widths, the resulting operator interface has to be scaled for the most demanding case. Next, we have to apply our buffer merging technique, which allows us to create global buffers able to work on various depths and data widths, as presented in the previous section.

Figure 9 presents such a transformation operated on two operators (Op_A and Op_B), working on different data widths and depths. A generic reconfigurable operator named *Op_Dyn* is created and its interface is adapted. Its functionality will be defined through run-time reconfiguration by the *configuration manager* process. Only the *Config_Select* input is added in the case of a parameterizable operator (*Op_Param*).

5.4. Design generation for run-time reconfiguration management

In order to perform reconfiguration of the dynamic part we have chosen to divide this process in two subparts: a *configuration manager* and a *protocol builder*. The *"configuration manager"* is automatically generated from our libraries according to the sequencing of operations expressed in the macrocode. A *"configuration manager"* is attached to each parameterizable or dynamic operator.

The configuration manager is in charge of operation selection which must be executed by the configurable operator by sending configuration requests. These requests are sent only when an operation has completed its computation and

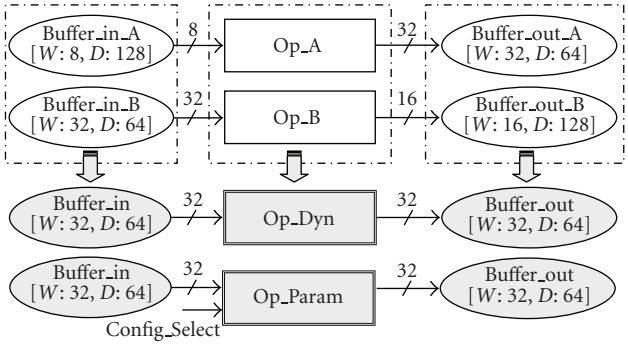


Figure 9: Buffers and operators merging example.

if a different operation has to be executed after. So reconfigurations are performed as soon as the current operation is completed in order to enable configuration prefetching as described before. This functionality provides also information on the current state of the operator. This is useful to start operator computations (with *"enable"* signal) only when the configuration process is ended.

Figure 10 shows a simple example based on two operations (j and k) which are executed successively. Labels **M** and **P** show where the functionalities *"configuration manager"* and *"protocol builder"* are, respectively, implemented.

Case (a) shows the design generated for a nonreconfigurable component. The two operators are physically implemented and are static. Case (b) is based on a parameterizable operator; the selection of configurations is managed by the *configuration manager*. There is no configuration latency; the operator is immediately available for computation. The signal *Config_Select* is basically a request of configuration, it results in the selection of a set of parameters among all internally stored. The third case (case (c)) is based on a dynamically reconfigurable operator which implements successively the two operations thanks to run-time reconfiguration. The reconfigurable part provides a virtual hardware, so at a time only one operator is physically implemented on this dynamic part. The configuration requests are sent to the *protocol builder* which is in charge to construct a valid reconfiguration stream in agreement with the used protocol mode (e.g., selectmap).

The configuration manager can perform reconfiguration (parametrization or partial reconfiguration) during other communications or computations scheduled on the same vertex. If the next configuration is conditional, this one is performed as soon as its value is known, allowing reconfiguration prefetching.

Encapsulation of operators with a standard interface allows to reconfigure only the area containing the operator without altering the design around. Buffers and functionalities involved in the overall control of the dynamic area remain on a static part of the circuit. This partitioning allows to reduce the size of the bitstream which must be loaded and decreases the time needed to reconfigure.

5.5. Reconfiguration control implementation cases

This way of proceeding must be adapted according to architecture considerations. There are many ways to reconfigure partially an FPGA. Figure 11 shows five solutions of architectures for this purpose.

Case (a) shows a standalone self-reconfiguration where the fixed part of the FPGA reconfigures the dynamic part. This case can be adapted for small amounts of bitstream data which can be stored in the on-chip FPGA memory. However, bitstreams require often a large amount of memory and cannot fit within the limited embedded memory provided by FPGAs, so bitstreams are stored in an external memory as depicted in case (b).

Case (c) shows the use of a microprocessor to achieve the reconfiguration. In this case, the FPGA sends reconfiguration requests to the processor through hardware interrupts, for instance. This microprocessor can be considered as a slave for the FPGA. The CPLD is used to provide some "glue logic" to link these two components. In case (c), the FPGA is still the initiator of the reconfiguration requests but they are performed through a microprocessor. This case is probably the most time consuming as the "protocol builder" functionality is a task on the microprocessor which can be activated through an hardware interrupt. Hence a noticeable latency is added, which is due to the program loading and fetching and hardware/software handshake.

In case (d), the FPGA can be seen as a slave as the reconfiguration control is fully managed by the microprocessor. In such a case the FPGA is used as a coprocessor for hardware acceleration. Last FPGA generations have embedded microprocessors and allow the last case of reconfiguration control. Implementation of these functionalities have a direct impact on the reconfiguration latency.

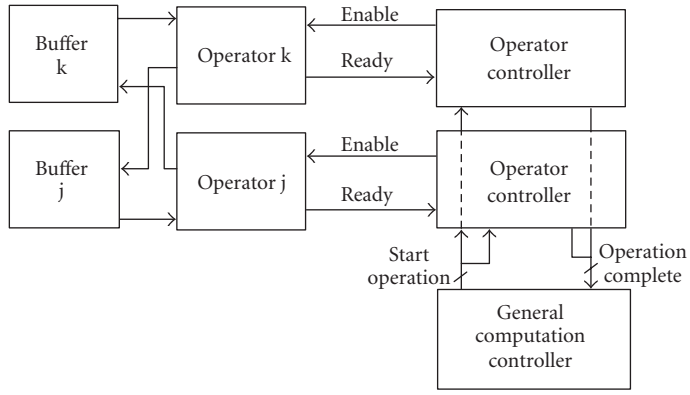
6. IMPLEMENTATION EXAMPLE

Our design flow and methodology have been applied for the implementation of a transmitter for future wireless networks in a 4G-based air interface [30]. In an advanced wireless application, SDR does not just transmit. It receives informations from the channel networks, probes the propagation channel and configures the system to achieve best performance and respond to various constraints such as bit-error rate (BER) or power consumption. We have considered here a configurable transmitter which can switch between three transmission schemes.

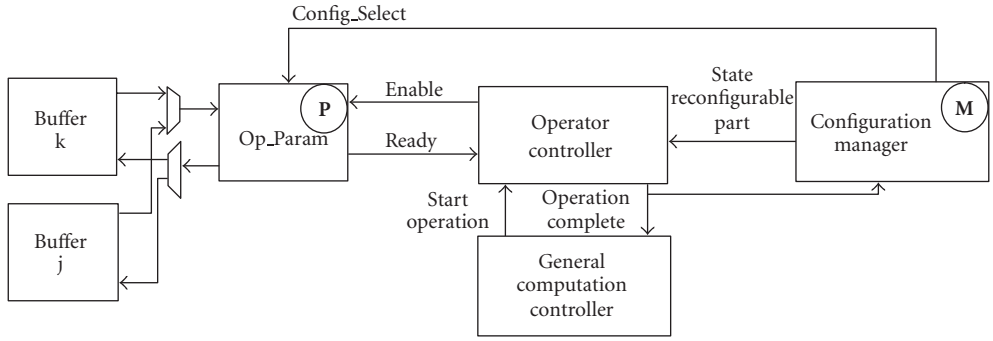
The basic transmission scheme is a multicarrier modulation based on orthogonal frequency division multiplexing (OFDM). OFDM is used in many communications systems such as: ADSL, Wireless LAN 802.11, DAB/DVB, or PLC. The first scheme corresponds to the most simple transmitter configuration named OFDM.

The second configuration uses a multicarrier code division multiple access (MC-CDMA) technique [30]. This multiple access scheme combines OFDM with spreading allowing the support of multiple users at the same time.

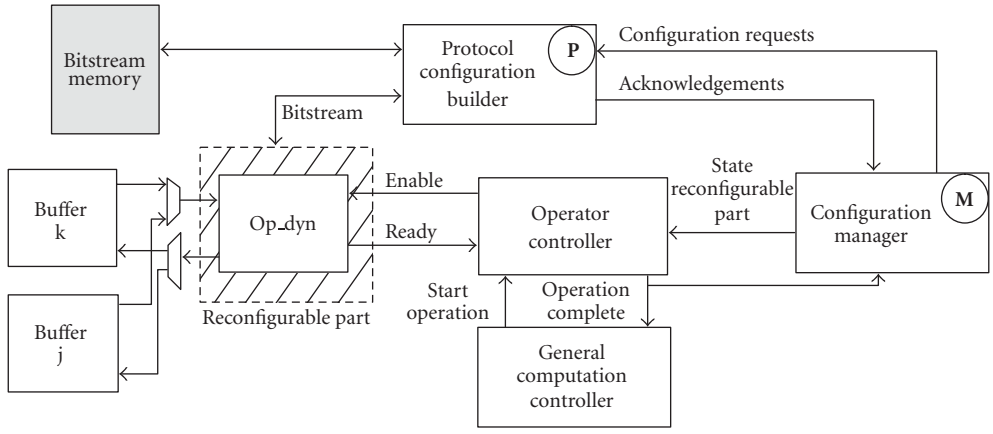
The third configuration uses a spread-spectrum multicarrier multiple access with frequency hopping pattern



(a) Static operator based structure



(b) Parameterizable operator-based structure



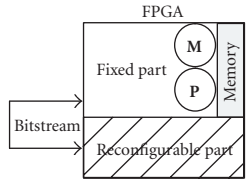
(c) Dynamic operator-based structure

Figure 10: Architecture comparison between a fixed/parameterized or dynamic computation-based structure.

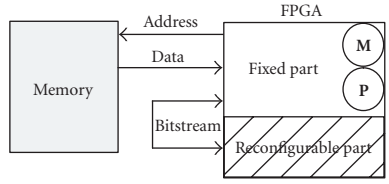
(FH-SS-MC-MA), as used in 802.11b standard (WiFi). It is a spread spectrum modulation technique where the signal is repeatedly switching frequencies during radio communication, minimizing probability of jamming/detection.

SynDEx algorithm graph, depicted by Figure 12(a), shows the numeric computation blocks of this configurable multimode transmitter.

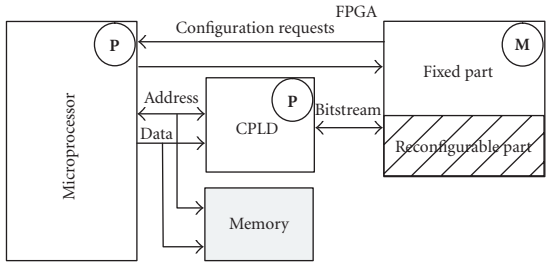
These three transmission schemes use channel coding and perform a forward correction error (FEC), corresponding to the *Channel_coding* block. The DSP can change the FEC to select a Reed-Solomon encoder or a convolutional encoder. Next, a modulation block performs an adaptive modulation between QPSK, QAM-16, or QAM-64 modulations. For MC-CDMA and FH-SS-MC-MA schemes, a *spreading* block implements a fast Hadamard transform (FHT). This block is inactive for OFDM scheme. A chip mapping (*Chip_mapping* block) is used in order to take into account the frequency diversity offered by OFDM modulation. This block performs either an interleaving on OFMD symbols for MC-CDMA, whereas the interleaving in FH-SS-MC-MA



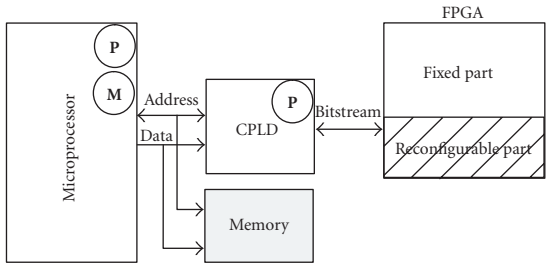
(a) Standalone self reconfiguration with on-chip bitstream memory



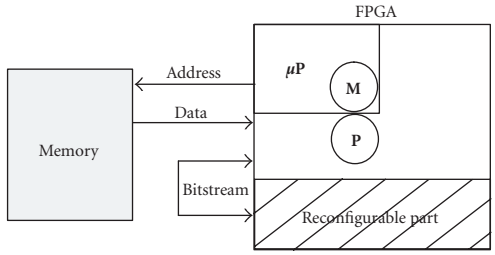
(b) Standalone self reconfiguration with off-chip bitstream memory



(c) Reconfiguration through a microprocessor



(d) Microprocessor reconfiguration control (FPGA as a slave)



(e) Standalone self-reconfiguration with embedded microprocessor

Figure 11: Different ways to reconfigure dynamic parts of an FPGA.

scheme is a frequency hopping (FH) pattern to allow the user data to take advantage of the diversity in time and frequency. The OFDM modulation is performed by an inverse fast Fourier transform thanks to *IFFT* block which also implements zero-padding process. For complexity and power consumption this IFFT can be implemented in radix-2 (*IFFT-R2*) or radix-4 (*IFFT-R4*) mode. Configuration selection (conditional entry *Config*) and data generation are handle by the *Data_Gen* block, whereas *DAC_If* block represents the interface to the digital-to-analog converter (DAC) device of the platform.

6.1. Implementation on a prototyping platform

We have implemented this reconfigurable transmitter on a prototyping board from Sundance Technology (Matignon, France) [31]. This board is composed of one DSP (C6701 from Texas Instrument) and one partially reconfigurable FPGA (Xc2v2000 from Xilinx with 10752 slices). Its SynDEx representation is shown in Figure 12(b). Communications between DSP and FPGA are ensured by SHB (Sundance high-speed bus) and CP (communication Port) communication medium from Sundance Technology. We have chosen to divide the FPGA in four vertices. One is static (*Interface*) and represents pre-developed logic for FPGA interfacing. The three remaining vertices (*FPGA_Dyn*, *FPGA_Dyn_1*, and *FPGA_Dyn_2*) are run-time reconfigurable vertices. Internal communications between these parts are ensured by the LI media.

Table 2 details the configurations and complexities of the reconfigurable transmitter computational blocks (depending on the transmission schemes). These complexities are obtained on a Xilinx VirtexII FPGA, where each slice includes two 4-input function generators (LUT). Some of these functions can be implemented thanks to the available Xilinx IPs [32].

6.1.1. Functions mapping

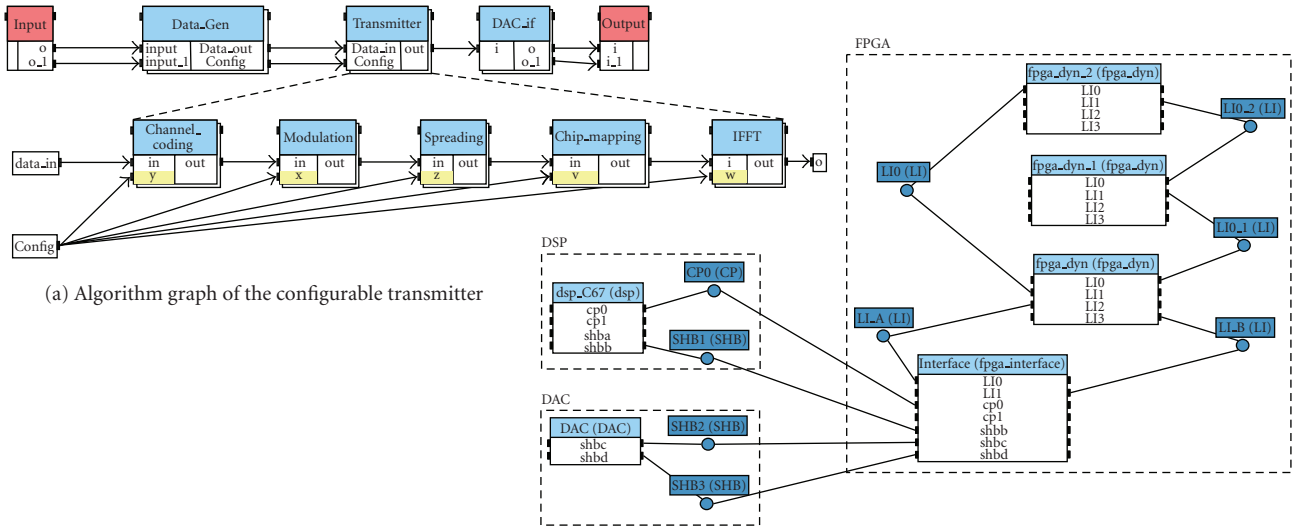
From the characterization of the computational blocks we can determine a possible implementation of the transmitter on the prototyping board. Table 3 summarizes this mapping.

IFFT block will be implemented thanks to a parameterizable IP from Xilinx, and mapped on the *FPGA_Dyn* vertex. Two static operators used for the Reed-Solomon encoder and convolutional encoder are also mapped on *FPGA_Dyn* vertex.

Functionalities *Interleaving* and *FH* of the *Chip mapping* block will be sequentially implemented by a dynamic operator with run-time reconfiguration on the *FPGA_Dyn_1* vertex.

Spreading block (*FHT*) will be performed through a static operator implemented on the *FPGA_dyn_2* vertex. On the same vertex the modulation is a parameterizable operator.

The *Data_Gen* and *DAC_If* blocks are mapped on the *Interface* vertex to ensure DSP's data and configurations transmission. They will not be detailed as we focus on the generation for the run-time reconfigurable vertices.



(a) Algorithm graph of the configurable transmitter

(b) Prototyping board architecture graph

Figure 12: Algorithm and architecture graphs of the configurable transmitter.

Table 2: Configurations and complexities.

Transmitter configuration	Computational blocks	Channel coding		Modulation		Spreading		Chip mapping		IFFT	
(1) Sampling frequency = 20 Mhz (2) Number of users = 32 (3) FFT = 256 points (4) OFDM symbol duration = 12.8 us (5) Frame duration = 1.32 ms	Configurations and complexities	A B C	Reed-Solomon encoder: 120 slices.	A B C	QPSK, 16QAM, 64QAM	B C	FHT (spreading factor: 32): 873 slices.	A B	Interleaving: 186 slices, 2 BlockRam.	A B C	IFFT-R2 (256-points, 16 bits): 752 slices, 3 BlockRam, 3 Mult18*18 – IP Xilinx COREGen xFFT v3.1.
		A B C	Convolutional encoder: 43 slices Xilinx Convolution Encoder v5.0.	A B C	Parameterizable operator: 60 slices			C	Frequency Hopping Pattern (FH): 246 slices, 2 Block-Ram.	A B C	IFFT-R4 (256-points, 16 bits): 1600 slices, 7 BlockRAM, 9 Mult18*18 – IP Xilinx COREGen xFFT v3.1. Parameterizable operator: 1600 slices - IP Xilinx COREGen xFFT v3.1.

Transmission schemes: OFDM (A), MC-CDMA (B), FH-SS-MC-MA (C).

6.1.2. *Resulting FPGA architecture*

The code, both for the fixed and dynamic parts, has been automatically generated with SynDEx thanks to the libraries. However, the generation of bitstreams needs a specific flow from Xilinx called modular design [15]. Modular design is based on a design partitioning in functional modules which are separately implemented and allocated to a specific FPGA area. Each module is synthesized to produce a netlist and then placed and routed separately. Reconfigurable modules communicate with the other ones, both fixed and reconfigurable, through a special bus macro (3-state buffers or slice-based bus macros) which is static. They guarantee that each time the partial the reconfiguration is performed the routing channels between modules remain unchanged. Partial bitstreams are created from the individual module designs.

Case (b) of Figure 11 represents our architecture. Virtex II integrates the ICAP FPGA primitive. ICAP is an acronym for internal configuration access port providing a direct access to the FPGA configuration memory and so enables a self-partial reconfiguration.

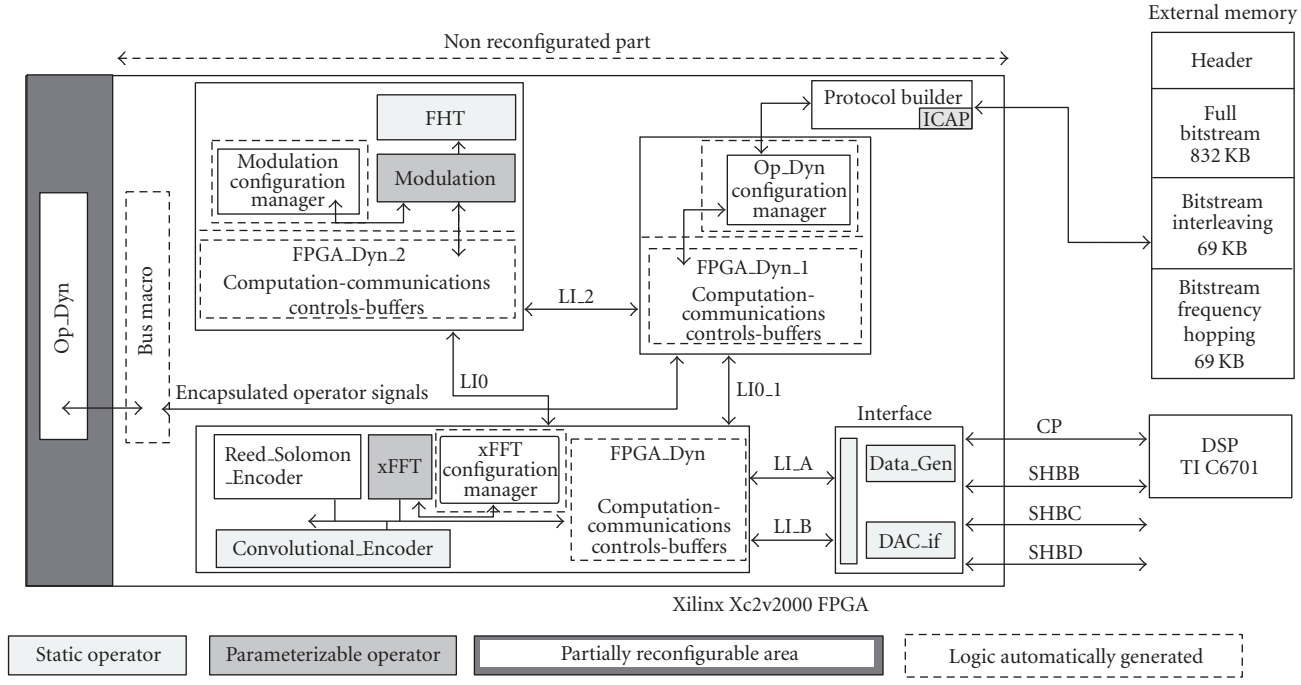
Figure 13 shows the resulting design of the reconfigurable transmitter which is compliant with the modular design flow for partial reconfiguration.

The nonreconfigurable part of the FPGA is composed of four areas resulting from the design generation of the four architecture graph vertices. Reconfigurable vertices architectures are detailed. Each one is composed of a general computation/communication controller with buffers.

Table 3: Functional blocks mapping (design automatically generated).

Computation vertice	FPGA_Interface	FPGA_Dyn	FPGA_Dyn_1	FPGA_Dyn_2	DSP C67	DAC
Functional Block	Data_Gen (1) DAC_if (1)	IFFT (2) Reed Solomon Encoder(1) Convolutional Encoder (1)	Chip Mapping (3)	FHT (1) Modulation (2)	Input	Output

(1): Static operator, (2): parameterizable operator, (3): dynamic operator.



CP : low speed digital communication bus for transmitter configuration
SHB : high speed digital communication bus for data transmission
ICAP : internal configuration access port

Figure 13: Reconfigurable transmitter architecture.

Parameterizable and run-time reconfigurable operators have their own configuration manager. The Dynamic operator configuration manager can address reconfiguration requests to the protocol builder. The protocol builder performs partial reconfigurations thanks to the ICAP primitive and bitstreams stored in an external memory (*interleaving* and *frequency hopping* bitstreams).

Only the left FPGA side is a run-time reconfigurable area and implements the dynamic operator. Encapsulated signals of the dynamic operator are accessed through bus macros as circumscribed by the modular Design flow. The size of the reconfigurable area has to be scaled to the most demanding function in logical resources, here FH function (246 slices, 2 BlockRam). Besides the shape of the reconfigurable area is constrained by the modular design and leads to allocate a greater area size than really necessary. In this case, the area takes the full FPGA height and 12 slices width (1300 slices). This area is the only run-time reconfigurated, other areas remain unchanged and are defined once during the full FPGA configuration.

Recently, this area constraint has been removed by the early-access partial reconfiguration design environment (EAPR) [32], which now allows partial reconfiguration modules of any rectangular size. This new modular design flow for run-time reconfiguration is supported by the PlanAhead floorplanner tool [33]. The VHDL files automatically generated by SynDEx are input source files for such floorplanning tools. Hence placement of bus macros, modules floorplanning, and partial bitstreams generation are performed with this target specific design environment. Nevertheless, our high-level methodology is independent of these backend steps concerning the final physical implementation. Any module-based flow for partial reconfiguration is compatible with our methodology.

6.1.3. Numerical results of implementation

The reconfiguration operates at 50 Mhz. The first and full configuration of the device takes 16 milliseconds while the partial reconfiguration process of chip mapping functionality (operator *Op_Dyn*) is about 2 milliseconds. That is of the order of several data frames, thus partial reconfiguration is suitable for a transmission scheme switching, as in the case of chip mapping functionality which is changed for MC-CDMA and FH-SS-MC-MA schemes. On the other hand, partial reconfiguration is too time consuming to be used for intratransmission scheme reconfiguration. It is the case if the channel coding and IFFT are implemented on the same dynamic operator.

6.1.4. Implementation comparison of chip mapping

As shown in Table 4, chip mapping operation is implemented either using "Interleaving" or "Frequency hopping" algorithms. Both have different complexities when implemented separately and statically (resp, 186 and 246 slices). With a dynamic implementation a same FPGA area is allocated for both (12% of the device) and each version requires 69 KB of external memory for the partial bitstreams (EA-PR flow [32] could reduce these needs).

FPGA resources needed to implement logic controls of the chip mapping functionality are more important with a dynamic reconfiguration implementation scheme (107 + 550 = 657 slices) as for a static and hand-made implementation (200 slices). The overhead is about 450 slices to allow run-time reconfiguration of the chip mapping functionality. This overhead is due to the generic VHDL structure generation, based on the macrocode description. However, this gap is decreasing with a greater number of configurations implemented on the same dynamic operator. The aim is to take advantage of the virtual hardware ability of the architecture. However, the flexibility and implementation speed up, through the automatic VHDL generation given by this methodology, can overcome this hardware resource overhead. For instance, we can add a Turbo convolutional encoder for the channel coding block (1133 slices, 6 BlockRam—IP Xilinx3GPP Compliant Turbo Convolutional Codec v1.0). As the size of the reconfigurable part is fixed by the designer, any design able to be satisfied with this area constraint can be implemented.

6.2. Implementation based on a network on chip

We have implemented the previous application on a specific architecture topology based on a Network on Chip. The goal is to combine the functional flexibility given by the run-time reconfiguration with the regular and adaptable communication scheme of an NoC. Hence allowing us to skip the need to redesign the architecture graph when the application is modified.

Network on Chip (NoC) is a new concept developed since several years and intended to improve the flexibility of IP communications and the reuse of intellectual property (IP) blocks. NoCs provide a systematic, flexible and scalable framework to manage communications between a large set of IP blocks. It can also reduce IP connection wires and optimize their usage. The dynamic reconfigurability of the communication paths responds to the fluctuating processing needs of embedded systems.

Dataflow IPs can be connected either through point to point links or through a NoC. Tools have been proposed in order to design and customize NoCs according to their application needs. We have developed both a NoC and its corresponding tool. This NoC is one possible target of the presented methodology. This NoC is adapted and optimized to allow the plug and the management of dynamically reconfigurable IPs. Reconfigurability is one important source of flexibility when combined with a flexible communication mechanism.

6.2.1. MicroSpider NoC presentation

Our NoC [34] is built with routing nodes using a wormhole packet switching technique to carry messages. Operators are linked to the routing nodes through network interfaces (NIs) with an FIFO-like protocol. Our NoC is customizable through an associated CAD tool [34]. Our CAD Tool is a decision and synthesis tool to help designers to obtain the hadhoc NoC depending on the application and implementation constraints. It is able to configure the various functionalities of our NoC. Finally, this tool generates an optimized dedicated NoC VHDL code at RTL level.

Network interfaces

Network interfaces are flexible and configurable to be adapted with the connected processing elements. They implement the network protocol. NIs connect IP blocks to the NoC. For NoC standardization reasons, we made the choice of the OCP interface [35] for the communication between NI and IPs. NI uses a table to transform OCP addresses map in packet header routing information according to the NoC configuration. The network interface architecture is strongly related to SynDEx scheduling technique that requires a virtual channel controller and buffering capabilities.

6.2.2. Implementation

The operations from the previous application example (Figure 12(a)) are dataflow operations. Dataflow operators have FIFO like protocols. We have added specific features to our NoC in order to optimize the dataflow traffic and the plug of IPs. We have also standardized the interfaces of the NoC. A subset of OCP interface standard has been selected and implemented.

We have implemented the previous application on a six-node NoC (Figure 14) integrated in the same FPGA, with one reconfigurable area per NoC routing node, and one node dedicated to a bridge to the external C6701 DSP. Each of the five remaining nodes can be the target of any application task. Several tasks can be grouped on the same node either to be dynamically reconfigured, parameterized or fixed and simply scheduled in time. We have evaluated latency and

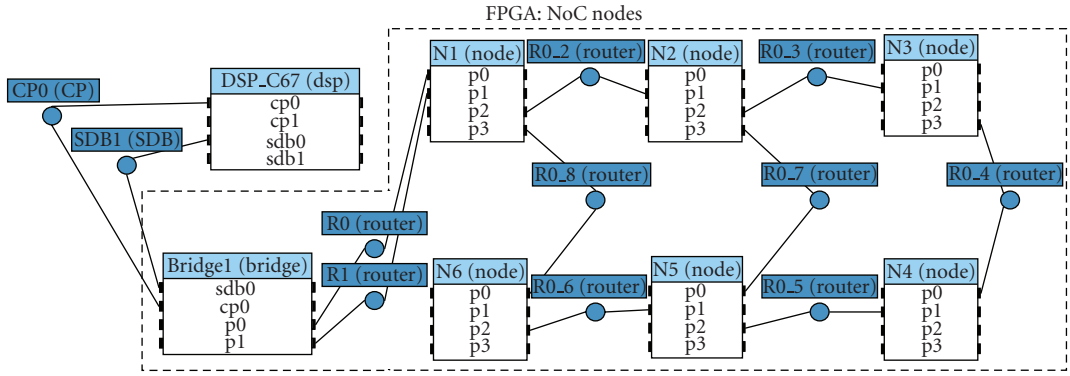


Figure 14: Architecture graph of the NoC nodes and the DSP.

Table 4: Static-dynamic implementation comparison of chip mapping.

Chip mapping functionalities coding and implementation	Designer (hand made) implementation: all static		Designed and generated automatically with SynDEx implementation: run-time reconfiguration		
	Controls	IPs (Interleaving+FH)	Controls		IPs
			Protocol Builder	Overall dynamic part controls	Reconfigurable area capacity (used)
Slices:	200	186 + 246 = 432	107	550	1300 (246)
Block RAM(18 Kbits):	2	—	—	2	14 (2)
FPGA area:	1.8%	4%	1%	5.1%	12% (2.5%)
Switching latency:	—	—	—	—	2 ms
External memory:	832 KB (Full bitstream)		832 KB (Full bitstream) + 2*69 KB (Partial bitstreams)		

Table 5: (1): Static operator, (2): parameterizable operator, (3): dynamic operator application function mapping on the NoC.

N1	N2	N3	N4	N5	N6
Turbo encoder (1)	Reed Solomon encoder(3) Convolutional encoder(3)	Modulation(2)	Spreading(1)	Chip mapping (3)	IFFT (2)

throughput of unloaded NoC links. These figures are introduced in SynDEx heuristic. Table 5 shows the functions mapping on this NoC As SynDEx schedules transfers in time, we use virtual channels in order to guarantee priority of first scheduled transfers. Thus the latency is deterministic and accurate. The M4 code generated by our methodology provides all the scheduling of treatments and communications as well as the source and target of each communication. These informations are extracted and translated to a C code [36] for a Xilinx Picoblaze micro-controller in charge of the dynamic operators and parameterizable operators.

Figure 15 details a NoC node structure. There is one Picoblaze for each NoC interface linked to dynamically reconfigurable operators. The Picoblaze controls the scheduling of operators, the size, the target and the starting of data transfers from the running operator. A NoC IP is the grouping of the operators, the picoblaze processor and the control operators and OCP adaptation logic. The scheduling of communications is managed with virtual channels in the NoC interfaces. They are configured by the Picoblaze. Implementation results are presented in Table 6.

Table 6: Noc node resources usage.

IP	Nb Slices	Freq (MHz)	Nb BRAM
PicoBlaze	110	200	1
NoC Node	430	200	2

The NoC cost is similar to the point to point solution with all the advantages of flexibility and scalability. With this solution there is no need to design a dedicated architecture graph for each new application. One general purpose 2D mesh can be selected for the architecture graph. Also, the coupling of an NoC with dynamically reconfigurable operators allow a new level of flexibility and optimization.

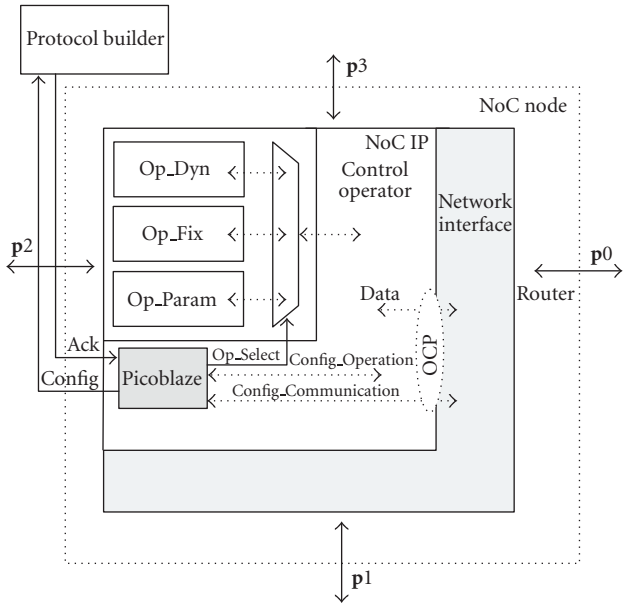


Figure 15: NoC Node detailed view.

7. CONCLUSION

We have described a methodology flow to manage automatically partially reconfigurable parts of an FPGA. It allows to map applications over heterogeneous architectures and fully exploit the advantages given by partially reconfigurable components. This design flow has the main advantage to target software components as well as hardware components to implement complex applications from a high-level functional description.

This methodology is independent of the final implementation of the run-time reconfiguration which is device dependent and achieved with back-end tools. This modeling can be applied on various components of different granularities. The AAA methodology and associated tool SynDEx have been used to perform the mapping and code generation for fixed and dynamic parts of FPGA. However, SynDEx's heuristic needs additional developments to fully take into account the reconfiguration time during the graph matching process.

That will allow the user to find a mapping and a scheduling of the application in order to improve the reconfiguration time or functional density of the resulting architecture. This methodology can easily be used to introduce dynamic reconfiguration on predeveloped fixed designs as well as for fast IP block integration on fixed or reconfigurable architectures. Thanks to the automatic code generation the development cycle is alleviated and accelerated. This top-down design approach makes it possible to accurately evaluate system implementation, according to functions complexity and architecture properties. Besides, the benefits of this approach fit into the SoftWare radio requirements for efficient design methods, and adds more flexibility and adaptation capacities through partial run-time reconfiguration on FPGA-based systems.

REFERENCES

[1] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 2, pp. 193–207, 2005.

[2] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '01)*, pp. 642–649, Munich, Germany, March 2001.

[3] R. David, D. Chillet, S. Pillement, and O. Sentieys, "Mapping future generation mobile telecommunication applications on a dynamically reconfigurable architecture," in *Proceedings of IEEE International Conference on Acoustic, Speech, and Signal Processing (ICASSP '02)*, vol. 4, p. 4194, Orlando, Fla, USA, May 2002.

[4] B. Salefski and L. Caglar, "Re-configurable computing in wireless," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 178–183, Las Vegas, Nev, USA, June 2001.

[5] C. Ebeling, C. Fisher, G. Xing, M. Shen, and H. Liu, "Implementing an OFDM receiver on the RaPiD reconfigurable architecture," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1436–1448, 2004.

[6] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Mapping applications onto reconfigurable kressarrays," in *Proceeding of the 9th International Workshop on Field Programmable Logic and Applications (FPL '99)*, pp. 385–390, Glasgow, Scotland, August-September 1999.

[7] S. Erdogan, M. Eads, and T. Shaneyfelt, "Virtual hardware management for high performance signal processing," in *Proceedings of the 3rd IASTED International Conference on Circuits, Signals, and Systems (CSS '05)*, pp. 36–39, Marina del Rey, Calif, USA, October 2005.

[8] P. Garcia, M. Schulte, K. Compton, E. Blem, and W. Fu, "An overview of reconfigurable hardware in embedded systems," *EURASIP Journal of Embedded Systems*, vol. 2006, Article ID 56320, 19 pages, 2006.

[9] J. Mitola III, "Software radio architecture evolution: foundations, technology tradeoffs, and architecture implications," *IEICE Transactions on Communications*, vol. E83-B, no. 6, pp. 1165–1173, 2000.

[10] "Joint tactical radio system website," http://enterprise.spawar.navy.mil.

[11] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware HW-SW partitioning for reconfigurable architectures with partial dynamic reconfiguration," in *Proceedings of the 42nd Design Automation Conference (DAC '05)*, pp. 335–340, Anaheim, Calif, USA, June 2005.

[12] C. Sorel and Y. Lavarenne, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of the 1st ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03)*, pp. 123–132, Mont Saint-Michel, France, June 2003.

[13] M. L. Silva and J. C. Ferreira, "Support for partial run-time reconfiguration of platform FPGAs," *Journal of Systems Architecture*, vol. 52, no. 12, pp. 709–726, 2006.

[14] E. Carvalho, N. Calazans, E. Brião, and F. Moraes, "PaDReH—a framework for the design and implementation of dynamically and partially reconfigurable systems," in *Proceedings of the 17th Symposium on Integrated Circuits and Systems Design (SBCCI '04)*, pp. 10–15, Pernambuco, Brazil, September 2004.

[15] "Xapp290: two flows for partial reconfiguration: Module based or difference based," http://direct.xilinx.com/bvdocs/appnotes/xapp290.pdf.

[16] A. Brito, M. Kuhnle, M. Hubner, J. Becker, and E. Melcher, "Modelling and simulation of dynamic and partially reconfigurable systems using systemc," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, pp. 35–40, Porto Alegre, Brazil, March 2007.

[17] S. Craven and P. Athanas, "A high-level development framework for run-time reconfigurable applications," in *Proceedings of the 9th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD '06)*, Washington, DC, USA, September 2006.

[18] J. P. Diguet, G. Gogniat, J. L. Philippe, et al., "EPICURE: a partitioning and co-design framework for reconfigurable computing," *Microprocessors and Microsystems*, vol. 30, no. 6, pp. 367–387, 2006.

[19] F. Dittmann, F.-J. Rammig, M. Streubühr, C. Haubelt, A. Schallenberg, and W. Nebel, "Exploration, partitioning and simulation of reconfigurable systems," *Information Technology*, vol. 49, no. 3, p. 149, 2007.

[20] B. Steinbach, T. Beierlein, and D. Fröhlich, "Uml-based codesign for run-time reconfigurable architectures," in *Languages for System Specification: Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specifications from FDL '03*, pp. 5–19, Norwell, Mass, USA, 2004.

[21] Y. Qu, J.-P. Soininen, and J. Nurmi, "Static scheduling techniques for dependent tasks on dynamically reconfigurable devices," *Journal of Systems Architecture*, vol. 53, no. 11, pp. 861–876, 2007.

[22] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pp. 986–991, Munich, Germany, March 2003.

[23] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '02)*, pp. 187–195, Monterey, Calif, USA, February 2002.

[24] V. Fresse, O. Déforges, and J.-F. Nezan, "AVSynDEx: a rapid prototyping process dedicated to the implementation of digital image processing applications on multi-DSP and FPGA architectures," *EURASIP Journal on Applied Signal Processing*, vol. 2002, no. 9, pp. 990–1002, 2002.

[25] M. Raulet, F. Urban, J.-F. Nezan, C. Moy, O. Deforges, and Y. Sorel, "Rapid prototyping for heterogeneous multicomponent systems: an MPEG-4 stream over a UMTS communication link," *EURASIP Journal on Applied Signal Processing*, vol. 2006, Article ID 64369, 13 pages, 2006.

[26] A. Al Ghouwayel, Y. Louët, and J. Palicot, "A reconfigurable architecture for the FFT operator in a software radio context," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '06)*, pp. 181–184, Island of Kos, Greece, May 2006.

[27] 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 13th International Conference on Field-Programmable Logic and Applications (FPL '03)*, vol. 2778 of *Lecture Notes in Computer Science*, pp. 585–594, Lisbon, Portugal, September 2003.

[28] J. Harkin, T. McGinnity, and L. Maguire, "Modeling and optimizing run-time reconfiguration using evolutionary computation," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 4, pp. 661–685, 2004.

[29] "Gnu m4—macro processor," http://www.gnu.org/software/m4/.

[30] S. Le Nours, F. Nouvel, and J.-F. Hélard, "Design and implementation of MC-CDMA systems for future wireless networks," *EURASIP Journal on Applied Signal Processing*, vol. 2004, no. 10, pp. 1604–1615, 2004.

[31] "Sundance multiprocessor technology ltd," http://www.sundance.com.

[32] "Xilinx intellectual property center," http://www.xilinx.com/ipcenter/.

[33] "Early access partial reconfiguration user guide, ug208 (v1.1)," Xilinx Inc, Tech. Rep., March, 2006.

[34] "Planahead," http://www.xilinx.com.

[35] S. Evain, J.-P. Diguet, and D. Houzet, "A generic CAD tool for efficient NoC design," in *Proceedings of International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS '04)*, pp. 728–733, Seoul, Korea, November 2004.

[36] "Programming FPGA's as MicroControllers: MicroFpga," http://www.microfpga.com/joomla.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 259686, 13 pages
doi:10.1155/2008/259686

Research Article

RRES: A Novel Approach to the Partitioning Problem for a Typical Subset of System Graphs

B. Knerr, M. Holzer, and M. Rupp

Institute of Communications and Radio-Frequency Engineering, Faculty of Electrical Engineering and Information Technology, Vienna University of Technology, 1040 Vienna, Austria

Correspondence should be addressed to B. Knerr, bknerr@nt.tuwien.ac.at

Received 11 May 2007; Revised 2 October 2007; Accepted 4 December 2007

Recommended by Marco D. Santambrogio

The research field of *system partitioning* in modern electronic system design started to find strong advertence of scientists about fifteen years ago. Since a multitude of formulations for the partitioning problem exist, the same multitude could be found in the number of strategies that address this problem. Their feasibility is highly dependent on the platform abstraction and the degree of realism that it features. This work originated from the intention to identify the most mature and powerful approaches for system partitioning in order to integrate them into a consistent design framework for wireless embedded systems. Within this publication, a thorough characterisation of graph properties typical for task graphs in the field of wireless embedded system design has been undertaken and has led to the development of an entirely new approach for the system partitioning problem. The restricted range exhaustive search algorithm is introduced and compared to popular and well-reputed heuristic techniques based on tabu search, genetic algorithm, and the global criticality/local phase algorithm. It proves superior performance for a set of system graphs featuring specific properties found in human-made task graphs, since it exploits their typical characteristics such as locality, sparsity, and their degree of parallelism.

1. INTRODUCTION

It is expected that the global number of mobile subscribers will reach more than three billion in the year 2008 [1]. Considering the fact that the field of wireless communications emerged only 25 years ago, this growth rate is absolutely tremendous. Not only its popularity experienced such a growth, but also the complexity of the mobile devices exploded in the same manner. The generation of mobile devices for 3G UMTS systems is based on processors containing more than 40 million transistors [2]. Compared to the first generation of mobile phones, a staggering increase in complexity of more than six orders of magnitude has taken place [3] in the last 15 years. Unlike the popularity, the growing complexity led to enormous problems for the design teams to ensure a fast and seamless development of modern embedded systems.

The International Technology Roadmap for Semiconductors [4] reported a growth in design productivity, expressed in terms of designed transistors per staff month, of approximately 21% compounded annual growth rate (CAGR), which lags behind the growth in silicon complexity. This is known as the *design gap* or *productivity gap*. A broad range of reasons exist that hold responsible for the *design gap* [5, 6]. The extreme heterogeneity of the applied technologies in the systems adopts a predominant position among those. The combination of computation-intensive signal processing parts for ever higher data rates, a full set of multimedia applications, and the multitude of standards for both areas led to a wild mixture of technologies in a state-of-the-art mobile device: general-purpose processors, DSPs, ASICs, multibus structures, FPGAs, and analog mixed signal domains may be coexisting on the same chip.

Although a number of EDA vendors offer tool suites (e.g., ConvergenSC of CoWare, CoCentric System Studio of Synopsys, Matlab/Simulink of The MathWorks) that claim to cope with all requirements of those designs, some crucial steps are still not, or inappropriately, covered: for instance, the automatic conversion from floating-point to fixed-point representation, architecture selection, as well as system partitioning [7].

This work focuses on the problem of hardware/software (hw/sw) partitioning, that is, loosely spoken, the mapping of functional parts of the system description to architectural components of the platform, while satisfying a set of constraints like time, area, power, throughput, delay, and so forth. *Hardware* then usually addresses the implementation of a functional part, for example, performing an FIR or CRC, as a dedicated hardware unit that features a high throughput and can be very power efficient. On the other hand, a custom data path is much more expensive to design and inflexible when it comes to future modifications. Contrarily, *software* addresses the implementation of the functionality as code to be compiled for a general-purpose processor or DSP core. This generally provides flexibility and is cheaper to maintain, whereas the required processors are more power consuming and offer less performance in speed. The optimal trade-off between cost, power, performance, and chip area has to be identified. In the following, the more general term system partitioning is preferred to hw/sw partitioning, as the classical binary decision between two implementation types has been overcome by the underlying complexity as well. The short design cycles in the wireless domain boosted the demand for very early design decisions, such as architecture selection and system partitioning on the highest abstraction level, that is, the algorithmic description of the system. There is simply no time left to develop implementation alternatives [5], which was used to be carried out manually by designers recalling their knowledge from former products and estimating the effects caused by their decision. The design complexity exposed this approach unfeasible and forced research groups to concentrate their efforts on automating the system partitioning as much as possible.

For the last 15 years, system partitioning has been a research field starting with first approaches being rather theoretic in their nature up to quite mature approaches with a detailed platform description and a realistic communication model. N.B., until now, none of them has been included in any *commercial* EDA tool, although very promising strategies do exist in academic surroundings.

In this work, a new deterministic algorithm is introduced that addresses the hw/sw partitioning problem. The chosen scenario follows the work of other well-known research groups in the field, namely, Kalavade and Lee [8], Wiangtong et al. [9], and Chatha and Vemuri [10]. The fundamental idea behind the presented strategy is the exploitation of distinct graph properties like locality and sparsity, which are very typical for human-made designs. Generally speaking, the algorithm *locally* performs an exhaustive search of a restricted size while incrementally stepping through the graph structure. The algorithm shows strong performance compared to implementations of the genetic algorithm as used by Mei et al. [11], the *penalty reward* tabu search proposed by Wiangtong [9], and the GCLP algorithm of Kalavade [8] for the classical binary partitioning problem. And a discussion of its feasibility is given with respect to the extended partitioning problem.

The rest of the paper is organised as follows. Section 2 lists the most reputed work in the field of partitioning techniques. Section 3 illustrates the basic principles of system partitioning, gives an overview of typical graph representations, and introduces the common platform abstraction. It is followed by a detailed description of the proposed algorithm and an identification of the essential graph properties in Section 5. In Section 6, the sets of test graphs are introduced and the results for all algorithms are discussed. The work is concluded and perspectives to future work are given in Section 7.

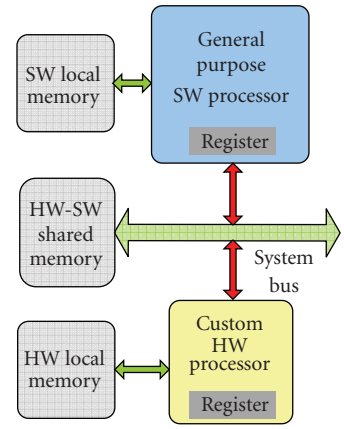


Figure 1: Common platform abstraction.

2. RELATED WORK

This section provides a structured overview of the most influential approaches in the field of system partitioning. In general, it has to be stated that heuristic techniques dominate the field of partitioning. Some formulations have been proved to be $\mathcal{NP}$ complete [12], and others are in $\mathcal{P}$ [13]. For the most formulations of partitioning problems, especially when combined with a scheduling scenario, no such proofs exist, so they are just considered as *hard*.

In 1993, Ernst et al. [14] published an early work on the partitioning problem starting from an all-software solution within the COSYMA system. The underlying architecture model is composed of a programmable processor core, memory, and customised hardware (Figure 1). The general strategy of this approach is the *hardware extraction* of the computational intensive parts of the design, especially loops, on a fine-grained basic block level, until all timing constraints are met. These computation intensive parts are identified by simulation and profiling. Internally, simulated annealing (SA) is utilised to generate different partitioning solutions. In 1993, this granularity might have been feasible, but the growth in system complexity rendered this approach obsolete. However, simulated annealing is still eligible if the granularity is adjusted, to serve as a first benchmark provider due to its simple and quickly to implement structure.

In 1995, the authors Kalavade [12] published a fast algorithm for the partitioning problem. They addressed the coarse grained mapping of processes onto an identical architecture (Figure 1) starting from a directed acyclic graph (DAG). The objective function incorporates several constraints on the available silicon area (hardware capacity),

memory (software capacity), and latency as a timing constraint. The global criticality/local phase (GCLP) algorithm is basically a greedy approach, which visits every process node once and is directed by a dynamic decision technique considering several cost functions.

In the work of Eles et al. [15], a tabu search algorithm is presented and compared to simulated annealing and a Kernighan-Lin (KL) based heuristic. The target architecture does not differ from the previous ones. The objective function concentrates more on a trade-off between the communication overhead between processes mapped to different resources and a reduction of execution time gained by parallelism. The most important contribution is the preanalysis before the actual partitioning starts. Static code analysis techniques down to the operational level are combined with profiling and simulation to identify the computation intensive parts of the functional code. A suitability metric is derived from the occurrence of distinct operation types and their distribution within a process, which is later on used to guide the mapping to a specific implementation technology.

In the later nineties, research groups started to put more effort into combined partitioning and scheduling techniques. One of the first approaches to be mentioned of Chatha and Vemuri [16] features the common platform model depicted in Figure 1. Partitioning is performed in an iterative manner on system level with the objective of minimising execution time while maintaining the area constraint. The partitioning algorithm mirrors exactly the control structure of a classical Kernighan-Lin implementation adapted to more than two implementation techniques, that is, for both hardware and software exist more than one implementation type. Every time a node is tentatively moved to another implementation type, the scheduler estimates the change in the overall execution time instead of rescheduling the task graph. By this means, a low runtime is preserved by losing reliability of their objective function since the estimated execution time is only an approximation. The authors extended their work towards combined retiming, scheduling, and partitioning of transformative applications, for example, JPEG or MPEG decoder [10].

A very mature combined partitioning and scheduling approach for directed acyclic graphs (DAG) has been published in 2002 by Wiangtong et al. [9]. The target architecture adheres to the concept given in Figure 1, but features a more detailed communication model. The work compares three heuristic methods to traverse the search space of the partitioning problem: simulated annealing, genetic algorithm, and tabu search. Additionally, the most promising technique of this evaluation, tabu search, is further improved by a so-called *penalty reward* mechanism. A reimplementation of this algorithm confirms the solid performance in comparison to the simulated annealing and genetic algorithms for larger graphs.

Approaches based on genetic algorithms have been used extensively in different partitioning scenarios: Dick and Jha [17] introduced the MOGAC cosynthesis system for combined partitioning/scheduling for periodic acyclic task graphs, Mei et al. [11] published a basic GA approach for the binary partitioning in a very similar setting to our work, and Zou et al. [18] demonstrated a genetic algorithm with a finer granularity (control flow graph level) but with the common platform model of Figure 1.

3. SYSTEM PARTITIONING

This section covers the fundamentals of system partitioning, the graph representation for the system, and the platform abstraction. Due to limited space, only a general discussion of the basic terms is given in order to ensure a sufficient understanding of our contribution. For a detailed introduction to partitioning, please refer to the literature [19, 20].

3.1. Graph representation of signal processing systems

A common ground of modern signal processing systems is their representation in dependence on their nature as data-flow-oriented systems on a macroscopic level, for instance, in opposition to a call graph representation [21]. Nearly every signal processing work suite offers a graphical block-based design environment, which mirrors the movement of data, streamed or blockwise, while it is being processed [22–24]. The transformation of such systems into a task graph is therefore straightforward and rather trivial. To be in accordance with most of the partitioning approaches in the field, we assume a graph representation to be in the form of synchronous data flow graphs (SDF), that has been firstly introduced in 1987 [25]. This form established the backbone of renowned signal processing work suites, for example, Ptolemy [23] or SPW [22]. It captures precisely multiple invocations of processes and their data dependencies and thus is most suitable to serve as a system model. In Figure 2(a), a simple example of an SDF graph $G = (\mathcal{V}, \mathcal{E})$ is depicted that is composed of a set of vertices $\mathcal{V} = \{a, \ldots, e\}$ and a set of edges $\mathcal{E} = \{e_1, \ldots, e_5\}$. The numbers on the tail of each edge e_i represent the number of samples produced per invocation of the vertex at the edge's tail, $\text{out}(e_i)$. The numbers on the head of each edge indicate the number of samples consumed per invocation of the vertex at the edge's head, $\text{in}(e_i)$. According to the data rates at the edges, such a graph can be uniquely transformed into a single activation graph (SAG) in Figure 2(b). Every vertex in an SAG stands for exactly one invocation of the process, thus the complete parallelism in the design becomes visible. Here, vertex b and d occur twice in the SAG to ensure a valid graph execution, that is, every produced data sample is also consumed. The vertices cover the functional objects of the system, or *processes*, whereas the edges mirror data transfers between different processes.

Most of the partitioning approaches in Section 2 premise the *homogeneous*, acyclic form of SDF graphs, or they state to consider simply DAGs. An SDF graph is called *homogeneous* if for all $e_i \in \mathcal{E}$, $\text{out}(e_i) = \text{in}(e_i)$. Or in other words, the SDFG and SAG exhibit identical structures. We explicitly allow for general SDF graphs in our implementations of GA, TS, and the new proposed algorithm. The transformation of general SDF graphs into homogeneous SAG graphs is described in [26], and does only affect the implementation complexity of the mechanism that schedules a *given* partitioning solution

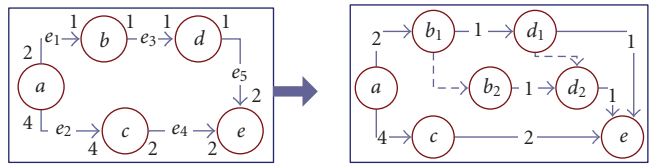


Figure 2: Simple example of a synchronous data flow graph and its decomposition into a single activation graph.

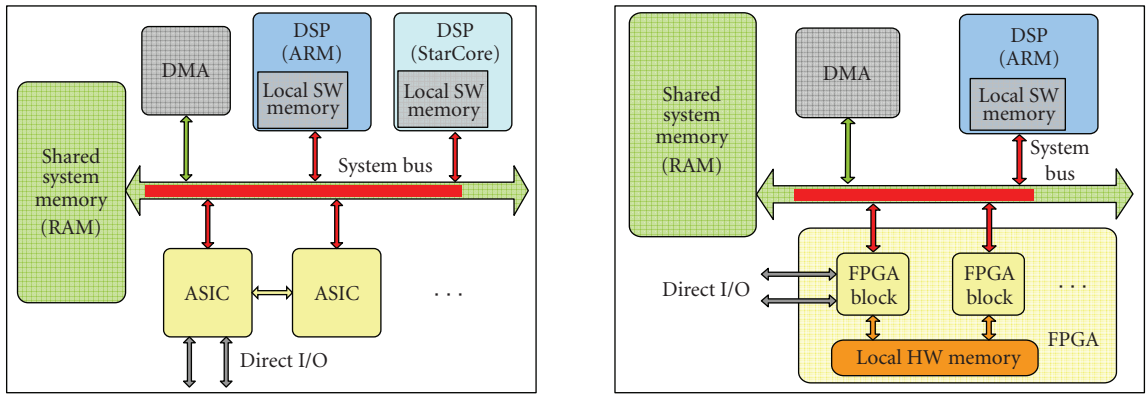


Figure 3: Origin (a) and modification (b) towards the common platform abstraction used for the algorithm evaluation.

onto a platform model. Note that due to its internal structure, the GCLP algorithm can not easily be ported to general SDF graphs and so it has been tested to acyclic homogeneous SDF graphs only.

In its current state, such a graph only describes the mathematical behaviour of the system. A binding to specific values for time, area, power, or throughput can only be performed in combination with at least a rough idea of the architecture, on which the system will be implemented. Such a platform abstraction will be covered in the following section.

3.2. Platform abstraction

The inspiration for the architecture model in this work originates from our experience with an industry-designed UMTS baseband receiver chip [27]. Its abstraction (see Figure 3(a)) has been developed to provide a maximum degree of generality while being along the lines of the industry-designed SoCs in use. The real reference chip is composed of two DSP cores for the control-oriented functionality (an ARM for the signalling part and a StarCore for the multimedia part). It features several hardware accelerating units (ASICs), for the more data-oriented and computation intensive signal processing, one system bus to a shared RAM for mixed resource communication, and optionally direct I/O to peripheral subsystems.

In Figure 3(b), the modification towards the platform concept with just one DSP and one hardware processing unit (e.g., FPGA) has been established (compare to Figure 1). This modification was mandatory for the comparison to the partitioning techniques of Wiangtong et al. [9] and Kalavade and Lee [8].

To the best of our knowledge, Wiangtong et al. [9] were the first group to introduce a mature communication model with high degree of realism. They differentiate between *load* and *store* accesses for every single memory/bus resource, and ensure a static schedule that avoids any collisions on the communication resources. Whereas, for instance, in the work of Kalavade [12], the communication between processes on the same resource is neglected completely, in the works of Chatha and Vemuri [10] or Vahid and Le [21], the system's execution time is estimated by *averaging* over the graph structure, and Eles et al. [15] do not generate a value for the execution time of the system at all, but base their solution quality mainly on the minimisation of communication between the hardware and the software resources.

Since, in this work, the achievable system time is considered as one of the key system traits, for which constraints exist, a reliable feedback on the makespan of a distinct partitioning solution is obligatory. Therefore, we adhere to a detailed communication model. Table 1 provides the example access times for reading and writing bits via the different resources of the platform in Figure 3(b). Communication of processes on the same resource uses preferably the local memory, unless the capacity is exceeded. Processes on different resources use the system bus to the shared memory. The presence of a DMA controller is assumed. In case

the designer already knows the bus type, for example, ARM AMBA 2.0, the relevant values could be modified accordingly.

With the knowledge about the platform abstraction described in Section 3.2 the system graph is enriched with additional information. The majority of the approaches assigns a set of characteristic values to every vertex as follows:

$$\forall v_i \in \mathcal{V} \; \exists \, I(v_i) = \{et^H, et^S, gc, \dots\}, \tag{1}$$

where et^H is the execution time as a hardware unit, et^S is the execution time of the software implementation, and gc is the gate count for the hardware unit and others like power consumption and so forth. Those values are mostly obtained by high-level synthesis [8] or estimation techniques like static code analysis [28, 29] or profiling [30, 31]. Unlike in the classical binary partitioning problem, in which just two implementation types for every process exist (et^H, et^S), a set of implementation types for every process is considered, comparable to the scenario chosen by Kalavade and Lee [8] and Chatha and Vemuri [10]. This is usually referred to as an extended partitioning problem. Mentor Graphics recently released the high-level synthesis tool, *CatapultC* [32], which allows for a fast synthesis of *C* functions for an FPGA or ASIC implementation. By a variation of parameters, for example, the unfolding factor, pipelining, or register usage, it is possible to generate a set of implementation alternatives $A^i_{\text{FPGA}} = \{gc, et\}$ for every single process v_i, like an FIR, featured by the consumed area in gates, the gate count gc, and the execution time et. Accordingly, for every other resource, like the ARM or the StarCore (SC) processors, sets of implementation alternatives, $A^i_{\text{ARM}} = \{cs, et\}$ and $A^i_{\text{SC}} = \{cs, et\}$, can be generated by varying the compiler options. For instance, the minimisation of DSP stall cycles is traded off against the code size cs for a lower execution time et as follows:

$$\begin{aligned}\forall v_i \in \mathcal{V} \; \exists \, \mathcal{I}^v(v_i) = \{&A^i_{\text{FPGA},1}, A^i_{\text{FPGA},2}, \dots, A^i_{\text{FPGA},k},\\ &A^i_{\text{ARM},1}, A^i_{\text{ARM},2}, \dots, A^i_{\text{ARM},l},\\ &A^i_{\text{SC},1}, A^i_{\text{SC},2}, \dots, A^i_{\text{SC},m}\}.\end{aligned} \tag{2}$$

In a similar fashion, the transfer times tt for the data transfer edges e_i are considered since several communication resources exist in the design: the bus access to the shared memory (shr), the local software (lsm), and the local hardware memory (lhm) as follows:

$$\forall e_i \in \mathcal{E} \; \exists \, \mathcal{I}^e(e_i) = \{\, tt^i_{\text{shr}}, tt^i_{\text{lsm}}, tt^i_{\text{lhm}} \,\}. \tag{3}$$

The next section finally introduces the partitioning problem for the given system graph and the platform model under consideration of distinct constraints.

3.3. Basic terms of the partitioning problem

In embedded system design, the term *partitioning* combines in fact two tasks: *allocation*, that is, the selection of architectural components, and *mapping*, that is, the binding of system functions to these components. Since in most formulations, the selection of architectural components is presumed, it is very common to use *partitioning* synonymously with *mapping*. In the remaining work, the latter will be used to be more precise. Usually, a number of requirements, or *constraints*, are to be met in the final solution, for instance, execution time, area, throughput, power consumption, and so forth. This problem is in general considered to be intractable or *hard* [33]. Arato et al. gave a proof for the $\mathcal{NP}$ completeness, but in the same work, they showed that other formulations are in $\mathcal{P}$ [13]. Our work elaborates on such an $\mathcal{NP}$-partitioning scenario combined with a multiresource scheduling problem. The latter has been proven to be $\mathcal{NP}$-complete [34, 35].

Table 1: Maximum throughput for read/write accesses to the communication/memory resources.

Communication	Read (bits/μs)	Write (bits/μs)
Local software memory	512	1024
Local hardware memory	2048	4096
Shared system bus	1024	2048
Direct I/O	4096	4096

With the platform model given in Section 3.2, the *allocation* has been established. In Figure 4, the *mapping* problem of a simple graph is depicted. The left side shows the system graph, Figure 4(a), the right side shows the platform model in a graph-like fashion, Figure 4(b). With the connecting arcs in the middle, the system graph and the architecture graph compose the *mapping* graph. The following constraints have to be met to build a valid *mapping* graph.

(i) All vertices of the system graph have to be mapped to processing components of the architecture graph.

(ii) All edges of the system graph have to be mapped to communication components of the architecture graph as follows.

  (a) Edges that connect vertices mapped to an identical processing component have to be mapped to the local communication component of this processing component.

  (b) Edges connecting vertices mapped to different processing components have to be mapped to the communication component, that connects these processing components.

(iii) Communication components are either sequential or concurrent devices. If *read* or *write* accesses cannot occur concurrently, then a schedule for these access operations is generated.

(iv) Processing components can be sequential or concurrent devices. For sequential devices a schedule has to exist.

A mapping according to all these rules is called *feasible*. However, feasibility does not ensure validity. A *valid* mapping is a *feasible* mapping that fulfills the following constraints.

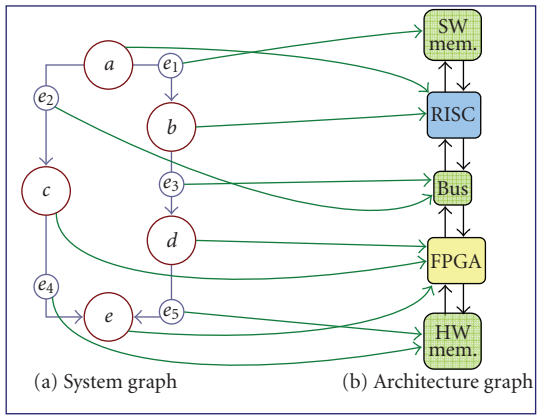


Figure 4: Mapping specification between system graph and architecture graph.

(i) A deadline T_{limit} measured in clock cycles (or μs) must not be exceeded by the makespan of the mapping solution.

(ii) Sequential processing devices have a limited instruction or code size capacity C_{limit} measured in bytes, which must not be exceeded by the required memory of mapped processes.

(iii) Concurrent processing devices have a limited area capacity A_{limit} measured in gates, which must not be exceeded by the consumed area of the mapped processes.

Other typical constraints, which have not been considered in this work in order to be comparable to the algorithms of the other authors, are monetary cost, power consumption, and reliability.

Due to the presence of sequential processing elements, bus or DSP, the mapping problem includes another hard optimisation challenge: the generation of optimal schedules for a mapping instance. For any two processes mapped to the DSP or data transfers mapped to the bus that overlap in time, a collision has to be solved. A very common strategy to solve occurring collisions in a fast and easy-to-implement manner is the deployment of a priority list introduced by Hu [36], which will be used throughout this work. As our focus lies on the performance evaluation of a mapping algorithm, a review of different scheduling schemes is omitted here. Please refer to the literature for more details on scheduling algorithms in similar scenarios [37–39].

4. SYSTEM GRAPHS PROPERTIES, COST FUNCTION, AND CONSTRAINTS

This section deals with the identification of system graph characteristics encountered in the partitioning problem. A set of properties is derived, which disclose the view to promising modifications of existing partitioning strategies and finally initiate the development of a new powerful partitioning technique. The latter part introduces the cost function to assess the quality of a given partitioning solution and the constraints such a solution has to meet.

4.1. Revision of system graph structures

The very first step to design a new algorithm lies in the acquisition of a profound knowledge about the problem. A review of the literature in the field of partitioning and electronic system design in general, regarding realistic and generated system graphs has been performed. The value ranges of the properties discussed below have been extracted from the three following sources:

(i) an industry design of a UMTS baseband receiver chip [27] written in COSSAP/C++;

(ii) a set of graph structures has been taken from Radioscape's RadioLab3G, which is a UMTS library for Matlab/Simulink [40];

(iii) three realistic examples stem from the standard task graph set of the Kasahara Lab [41].

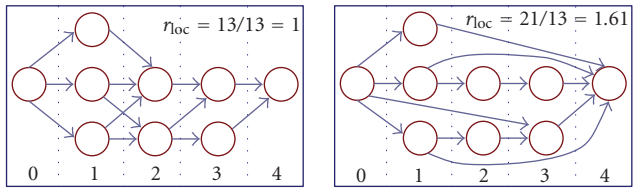
Additionally, many works incorporate one or two example designs taken from development worksuites they lean towards [8, 14]. Others introduce a fixed set of typical and very regular graph types [9, 39]. Nearly all of the mentioned approaches generate additional sets of random graphs up to hundreds of graphs to obtain a reliable fundament for test runs of their algorithms. However, truly random graphs, if not further specified, can differ dramatically from the specific properties found in human made graphs. Graphs in electronic system design, in which programmers capture their understanding of the functionality and of the data flow, can be isolated by specific values for the following set of graph properties.

Granularity

Depending on the granularity of the graph representation, the vertices may stand for a single operational unit (MAC, Add, or Shift) [14] or have the rich complexity of an MPEG or H.264 decoder. The majority of the partitioning approaches [8–10, 17] decide for medium-sized vertices that cover the functionality of FIRs, IDCTs, Walsh-Hadamards transform, shellsort algorithms, or similar procedures. This size is commonly considered as *partitionable*. The following graph properties are related to system graphs with such a granularity.

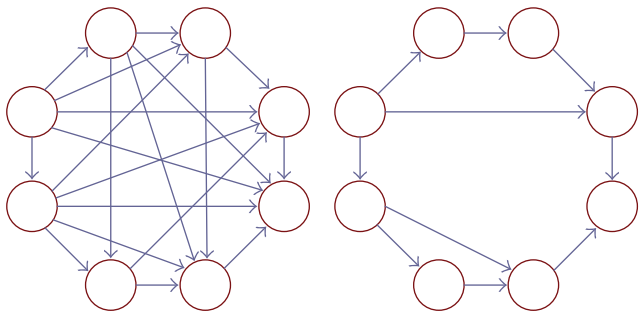
Locality

In graph theory, the term k-locality is defined as follows [42]: a locality of $k > 0$ means that when all vertices of a graph are written as elements of a vector with indices $i = 1 \ldots |\mathcal{V}|$, edges may only exist between vertices whose indices do not differ by more than k. More descriptively, human-made graphs in electronic system design reveal a strong affinity to this locality property for rather small k values compared to its number of vertices $|\mathcal{V}|$. From a more pragmatic perspective, it can be expressed as a graph's affinity to rather short edges, that is, vertices are connected to other vertices on a similar graph level. The generation of a k-locality graph is simple but the computation of the k-locality for a given graph is a hard optimisation problem itself, since k should be



(a) (b)

FIGURE 5: Examples for the *rank*-locality of two different graphs according to (4).



(a) dense (b) sparse

FIGURE 6: Density of graph structures.

the smallest possible. Hence, we introduce a related metric to describe the locality of a given graph: the *rank*-locality r_{loc}. In Figure 5, two graphs are depicted. At the bottom, the *rank* (or precedence) levels are annotated and the *rank*-locality is computed as follows:

$$r_{loc} = \frac{1}{|\mathcal{E}|} \sum_{e_i \in \mathcal{E}} \text{rank}(v_{sink}(e_i)) - \text{rank}(v_{source}(e_i)). \quad (4)$$

The *rank*-locality can be calculated very easily for a given graph. Very low values, $r_{loc} \in [1.0 \ldots 2.0]$, are reliable indicators for system graphs in signal processing.

Density

A directed graph is considered as *dense* if $|\mathcal{E}| \sim |\mathcal{V}|^2$, and as *sparse* if $|\mathcal{E}| \sim |\mathcal{V}|$ [42], see Figure 6. Here, an edge corresponds to a directed data transfer, which is either existing between two vertices or not. The possible values for the number of edges calculate to $(|\mathcal{V}| - 1) \leq |\mathcal{E}| \leq (|\mathcal{V}| - 1)|\mathcal{V}|$, and for directed *acyclic* graphs to $(|\mathcal{V}| - 1) \leq |\mathcal{E}| \leq (|\mathcal{V}| - 1)|\mathcal{V}|/2$. The considered system graphs are biased towards *sparse* graphs with a density ratio of about $\rho = |\mathcal{E}|/|\mathcal{V}| = 2 \ldots \sqrt{|\mathcal{V}|}$.

Degree of Parallelism

The degree of parallelism γ is in general defined as $\gamma = |\mathcal{V}|/|\mathcal{V}_{LP}|$, with $|\mathcal{V}_{LP}|$ being the number of vertices on the longest (critical) path [43]. In a weighted graph scenario this definition can easily be modified towards the fraction of the overall sum of the vertices' (and edges') weights divided by the sum of the weights of the vertices (and edges) encountered on the longest path. Apparently, this modification fails when the vertices and edges feature a set of varying weights since in our case, the execution times *et* and transfer times *tt* will serve as weights.

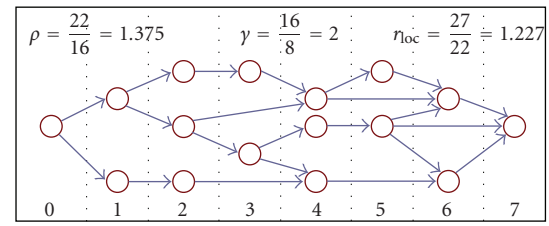


FIGURE 7: Task graph with characteristic values for ρ, r_{loc}, and γ.

Hence, for every vertex and every edge an average is built over their possible execution and transfer times, et_{avg} and tt_{avg}. These averaged values then serve as unique weights for the time-related degree of parallelism γ_t:

$$\gamma_t = \frac{\sum_{v_i \in \mathcal{V}} et^i_{avg} + \sum_{e_j \in \mathcal{E}} tt^j_{avg}}{\sum_{v_i \in \mathcal{V}_{LP}} et^i_{avg} + \sum_{e_j \in \mathcal{E}_{LP}} tt^j_{avg}}. \quad (5)$$

This property may vary to a higher degree since many chain-like signal processing systems exist as well as graphs with a medium, although rarely high, inherent parallelism, $\gamma_t = 2 \ldots \sqrt{|\mathcal{V}|}$. But for directed acyclic graphs this property can be calculated efficiently beforehand and serves as a fundamental metric that influences the choice of scheduling and partitioning strategies.

Taking these properties into account, random graphs of various sizes have been generated building up sets of at least 180 different graphs of any size.

A categorisation of the system graph according to the aforementioned properties for directed acyclic graphs can be efficiently achieved by a single breadth-first search as follows:

(i) the totalised values for area A_{total}, S_{total}, and time T_{total};
(ii) the time based degree of parallelism γ_t.
(iii) the ranks of all vertices;
(iv) the *density* ρ of the system graph.

These values can be achieved with linear algorithmic complexity $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$. A second run over the list of edges yields the *rank*-locality property in $\mathcal{O}(|\mathcal{E}|)$. The set of preconditions for the application of the following algorithm is comprised by a low to medium degree of parallelism $\gamma_t \in [2, 2\sqrt{|\mathcal{V}|}]$, a low *rank*-locality $r_{loc} \leq 8$, and a *sparse* density $\rho = 2 \ldots \sqrt{|\mathcal{V}|}$.

In Figure 7, a typical graph with low values for ρ and r_{loc} is depicted. The rank levels are annotated at the bottom of the graphic. The fundamental idea of the algorithm explained in Section 5 is that, in general, a local optimal solution, for instance, covering the rank levels 0 and 1, does probably not interfere with an optimal solution for the rank levels 6 and 7.

4.2. Cost function, constraints, and performance metrics

Although there are about as many different cost functions as there are research groups, all of the referred to approaches in Section 2 consider time and area as counteracting optimisation goals. As can be seen in (6), a weighted linear combination is preferred due to its simple and extensible structure. We have also applied Pareto point representations to seize the quality of these multiobjective optimisation problems [44], but in order to achieve comparable scalar values for the different approaches, the weighted sum seems more appropriate. According to Kalavade's work, code size has been taken into account as well. Additional metrics, for instance, power consumption per process implementation type, can just be added as a fourth linear term with an individual weight. The quality of the obtained solution, the cost value Ω_P for the best partitioning solution P, is then

$$\Omega_P = p_T(T_P)\,\alpha\,\frac{T_P - T_{\min}}{T_{\text{limit}} - T_{\min}} + p_A(A_P)\,\beta\,\frac{A_P}{A_{\text{limit}}} + p_S(S_P)\,\xi\,\frac{S_P}{S_{\text{limit}}}. \tag{6}$$

Here, T_P is the makespan of the graph for partitioning P, which must not exceed T_{limit}; A_P is the sum of the area of all processes mapped to hw, which must not exceed A_{limit}; S_P is the sum of the code sizes of all processes mapped to sw, which must not exceed S_{limit}. With the weight factors α, β, and ξ, the designer can set individual priorities. If not stated otherwise, these factors are set to 1. In the case that one of the values T_P, A_P, or S_P exceeds its limit, a penalty function is applied to enforce solutions within the limits:

$$p_A\left(\frac{A_P}{A_{\text{limit}}}\right) = \begin{cases} 1.0, & A_P \le A_{\text{limit}}, \\ \left(\dfrac{A_P}{A_{\text{limit}}}\right)^{\eta}, & A_P > A_{\text{limit}}. \end{cases} \tag{7}$$

The penalty functions for p_T and p_S are defined analogously. If not stated otherwise, η is set to 4.0.

The boolean validity value V_P of an obtained partitioning P is given by the boolean term: $V_P = (T_P \le T_{\text{limit}}) \wedge (A_P \le A_{\text{limit}}) \wedge (S_P \le S_{\text{limit}})$. A last characteristic value is the validity percentage $\Psi = N_{\text{valid}}/N$, which is the quotient of the number of valid solutions N_{valid} divided by the number of all solutions N, for a graph set containing N different graphs.

The constraints can be further specified by three ratios R_T, R_A, and R_S to give a better understanding of their strictness. The ratios are obtained by the following equations:

$$R_T = \frac{T_{\text{limit}} - T_{\min}}{T_{\text{total}} - T_{\min}}, \quad R_A = \frac{A_{\text{limit}}}{A_{\text{total}}}, \quad R_S = \frac{S_{\text{limit}}}{S_{\text{total}}}. \tag{8}$$

The totalised values for area A_{total}, code size S_{total}, and execution time T_{total} are simply built by the sum of the maximum gate counts gc, maximum code sizes cs, and maximum execution time $et_{\max}$ of every process (plus the maximum transfer time $tt_{\max}$ of every edge), respectively. The computation of $T_{\min}$ is obtained by scheduling the graph under the assumption of an implementation featuring a full parallelism, that is, unlimited FPGA resources and no conflicts on any sequential device. It has to be stated that $T_{\min}$ and T_{total} are lower and upper bounds since their exact calculation in most cases is a hard optimisation problem itself.

Consequently, a constraint is rather strict when the allowed for resource limit is small in comparison to the resource demands that are present in the graph. For instance, the totalised gate count A_{total} of all processes in the graph is $100k$ gates, if $A_{\text{limit}} = 20k$, then $R_A = 0.2$, which is rather strict, as in average, only every fifth process may be mapped to the FPGA or may be implemented as an ASIC.

The computational runtime Θ has been evaluated as well and is measured in clock cycles.

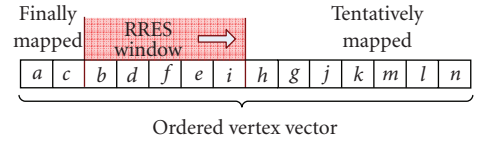


Figure 8: Moving window for the RRES on an ordered vertex vector.

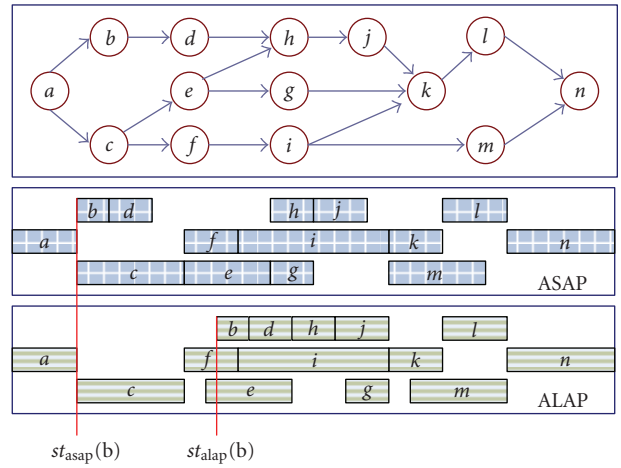


Figure 9: Two different start times for process (b) according to ASAP and ALAP schedule.

5. THE RESTRICTED RANGE EXHAUSTIVE SEARCH ALGORITHM

This section introduces the new strategy to exploit the properties of graph structures described in Section 4.1. Recall the fundamental idea sketched in the properties section of noninterfering rank levels. Instead of finding proper cuts in the graph to ensure such a noninterference, which is very rarely possible, we consider a moving window (i.e., a contiguous subset of vertices) over the topologically sorted vertices of the graph, and apply exhaustive searches on these subsets, as depicted in Figure 8. The annotations of the vertices refer to Figure 9. The window is moved incrementally along the graph structure from the start vertices to the exit vertices while locally optimising the subset of the RRES window.

The preparation phase of the algorithm comprises several necessary steps to boost the performance of the proposed

Table 2: Averaged cost $\overline{\Omega}_P$ obtained for RRES starting from different initial solutions.

Initial solution $\lvert\mathcal{V}\rvert$	Pure SW	Pure HW	Random	Heuristic	Heuristic and RRES
20	2.241	2.267	2.118	2.101	2.085
50	2.569	2.566	2.237	2.185	2.170
100	2.700	2.655	2.261	2.202	2.188

strategy. The initial solution, the very first assignment of vertices to an implementation type, has an impact on the achieved quality, although we can observe that this effect is negligible for fast and reasonable techniques to create initial solutions. In Table 2, the obtained cost values for an RRES (window length = 8, loose constraints) are depicted with varying initial solutions: pure software, pure hardware, guided random assignment according to the constraint setting, a more sophisticated but still very fast construction heuristic described in the literature [45], and when applying RRES on the partitioning solutions obtained by a preceding run with the aforementioned construction heuristic. Apparently, the local optima reached via the first two nonsensical initial solutions are substantially worse than the others. In the third column, the guided random assignment maps the vertices randomly but considers the constraint set in a simple way, that is, for any vertex, a real value in $[0, 1]$ is randomly generated and compared to a global threshold $T = (R_T + (1 - R_A) + R_S)/3$, hence leading to balanced starting partitions. The construction heuristic discussed in [45] in the fourth column even considers each vertex traits individually and incorporates a sorting algorithm with complexity $\mathcal{O}(|\mathcal{V}|\log(|\mathcal{V}|))$. In the last column, RRES has been applied twice, the second time on the solutions obtained for an RRES run with the custom heuristic. The improvement is marginal opposing the doubled run time. These examples will demonstrate that RRES is quite robust when working on a reasonable point of origin. Further on, RRES is always applied starting from the construction heuristic since it provides good solutions introducing only a small run time overhead, but even RRES with initial solution based on random assignment can compete with the other algorithms.

Another crucial part is certainly the identification of the order, in which the vertices are *visited* by the moving window. For the vertex order, a vector is instantiated holding the vertices indices. The main requirement for the ordering is that adjacent elements in the vector mirror the vicinity of readily mapped processes in the schedule. Different schemes to order the vertices have been tested: a simple rank ordering that neglects the annotated execution and transfer times; an ordering according to ascending Hu priority levels that incorporates the critical path of every vertex; a more elaborate approach is the generation of two schedules, *as soon as possible* and *as late as possible* as in Figure 9. For some vertices, we obtain the very same start times $st(v) = st_{\text{asap}}(v) = st_{\text{alap}}(v)$ for both schedules since for all $v \in \mathcal{V}_{LP}$ with $\mathcal{V}_{LP} \subseteq \mathcal{V}$ building the longest path(s) (e.g., vertex i). The start and end times are different if $v \notin \mathcal{V}_{LP}$ (e.g., b), then we chose $st(v) = (1/2)(st_{\text{asap}}(v) + st_{\text{alap}}(v))$ (e.g., vertex b).

An alignment according to ascending values of $st(v)$ yielded the best results among these three schemes, since the dynamic range of possible schedule positions is hence incorporated. It has to be stated that in the case of the binary partitioning problem, exactly two different execution times for any vertex exist, and three different transfer times for the edges (hw-sw, hw-hw, and sw-sw). In order to achieve just a single value for execution and transfer times for this consideration, again, different schemes are possible: utilising the values from the initial solution, simply calculating their average, or utilising a weighted average, which incorporates the constraints. The last technique yielded the best results on the applied graph sets. The exact weighting is given in the following equation:

$$\begin{aligned} et = \frac{1}{3}(&R_S\, et_{\text{sw}} + (1 - R_S)et_{\text{hw}} + R_A\, et_{\text{hw}} \\ &+ (1 - R_A)et_{\text{sw}} + \hat{R}_T\, et_{\text{sw}} + (1 - \hat{R}_T)et_{\text{hw}}), \end{aligned} \tag{9}$$

where $\hat{R}_T = R_T$ if $et_{\text{sw}} \geq et_{\text{hw}}$, and $\hat{R}_T = 1 - R_T$ otherwise. Note that this averaging takes place before the RRES algorithm starts to enable a good exploitation of its potential, it will not be mistaken as the method to calculate the task graph execution time during the RRES algorithm in general. Whereas during the RRES and all other algorithms, any generated partitioning solution is properly scheduled: parallel tasks and data transfers on concurrent resources run concurrently, and sequential resources arbitrate collisions of their processes or transfers by a Hu level priority list and introduce delays for the losing process or transfer.

Once the vertex vector has been generated, the main algorithm starts. In Algorithm 1 pseudocode is given for the basic steps of the proposed algorithm. Lines (1)-(2) cover the steps already explained in the previous paragraphs. The loop in lines (4)–(6) is the windowing across the vertex vector with window length W. From within the loop, the exhaustive search in line (9) is called with parameters for the window $v_i - v_j$. The swapping of the most recently added vertex v_j in line (10) is necessary to save half of the runtime since all solutions for the previous mapping of v_j have already been calculated in the iteration before. This is related to the break condition of the loop in following the line (11). Although the current window length is W, only 2^{W-1} mappings have to be calculated anew in every iteration. In line (12), the current mapping according to the binary representation of loop index i is generated. In other words, all possible permutations of the window elements are generated leading to new partitioning solutions. Any of these solutions is properly scheduled, avoiding any collisions, and the cost metric is computed. In lines (13)–(19), the checks for the best and the best valid solution are performed. The actual final mapping of the *oldest* vertex in the window v_i takes place in line (21). Here, the mapping of v_i is chosen, which is part of the best solution seen so far. When the window reaches the end of the vector, the algorithm terminates.

```
(0) RRES () {
(1)    createInitialSolution();
(2)    createOrderedVector();
(3)
(4)    for (i = 1; i <= |V| − W; i++) {
(5)       windowedExhaustiveSearch(i, i + W);
(6)    }
(7) }
(8)
(9) windowedExhaustiveSearch(int v_i, int v_j) {
(10)   swapVertex(v_j);
(11)   for (int i = 0; i < 2 (W − 1); i + +) {
(12)      createMapping (v_i, v_j, i);
(13)
(14)      if (constraints fulfilled)
   {  valid = true;}
(15)      if (cost < bestCost)
   {  storeSolution();}
(16)      if (cost < bestValCost && valid)
   {  storeValidSolution();}
(17)   }
(18)   mapVertex(v_i, bestSolution);
(19) }
```

Algorithm 1: Pseudocode for the RRES scheduling algorithm

6. RESULTS

To achieve a meaningful comparison between the different strategies and their modifications and to support the application of the new scheduling algorithm, many sets of graphs have been generated with a wider range as described in Section 4. For the sizes of 20, 50, and 100 vertices, there are graph sets containing 180 different graphs with a varying graph properties $\gamma_t = 2 \ldots 2\sqrt{|\mathcal{V}|}$, $r_{\text{loc}} = 1 \ldots 8$, and densities with $\rho = 1.5 \ldots \sqrt{|\mathcal{V}|}$. Two different constraint settings are given: *loose* constraints with $(R_T, R_A, R_S) = (0.5, 0.5, 0.7,)$, in which any algorithm found in 100% a valid solution, and *strict* constraints with $(R_T, R_A, R_S) = (0.4, 0.4, 0.5,)$ to enforce a number of invalid solutions for some algorithms. The tests with the strict constraints are then accompanied with the validity percentage $\Psi \leq 100\%$.

Naturally, the crucial parameter of RRES is the window length W, which has strong effects on both the runtime and the quality of the obtained solutions. In Figure 10, the first result is given for the graph set with the least number of vertices $|\mathcal{V}| = 20$ since a complete exhaustive search (ES) over all 2^{20} solutions is still feasible. The constraints are strict. The vertical axes show the range of the validity percentage Ψ and the best obtained cost values Ω averaged over the 180 graphs. Over the possible window lengths W, shown on the x-axis, the performance of the RRES algorithm is plotted. The dotted lines show the ES performance. For a window length of 20, the obtained values for RRES and ES naturally coincide. The algorithm's performance is scalable with the window length parameter W. The trade-off between solution quality and runtime can hence directly be adjusted by the number

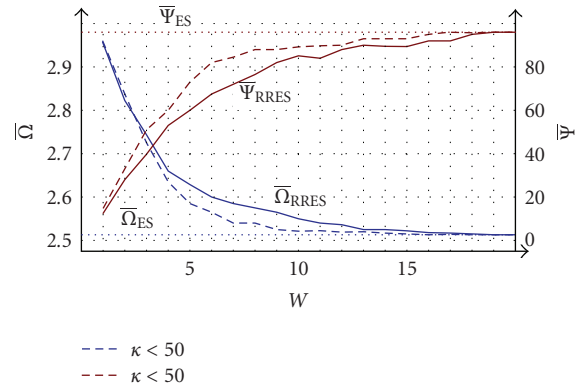


Figure 10: Validity Ψ and cost Ω for RRES, GCLP, and ES plotted over the window length W.

of calculated solutions $S = (|\mathcal{V}| - W)2^{(W-1)}$. The dashed curves are the cost and validity values over the graph subset, for which the product of rank locality and parallelism is $\kappa = \gamma\, r_{\text{loc}} < 50$. Obviously, there is a strong dependency between the proposed RRES algorithm and this product. In the last part of this section, this relation is brought into sharper focus.

For the following algorithms GA and TS that comprise a randomised structure, the outcome naturally varies. An ensemble of 30 different runs over any graph for any algorithm with a specific parameter set is performed. Since the distribution function of the cost values for these ensembles is not known, the Kolmogorov-Smirnov test [46] has been applied to any ensemble and any randomised algorithm to check whether a normal distribution of the cost values can be assumed. If so, the mean value and the standard deviation of the obtained cost values are sufficient to completely assess the performance of the algorithm. This assumption has been supported for all algorithms applied to graphs with a size equal or larger than 50 vertices. For smaller graphs of 20 vertices, this assumption turns out to be invalid for 28 out of 180 graphs. As in these cases, GA and RRES found to a large degree (near-)optimal solutions. Thus only the subset is compared by mean and standard deviation for which the normal distribution could be verified.

The parameter set of the GA implementation is briefly outlined. For a detailed description of the GA terms, please refer to the literature [47]. The chromosome coding utilises, as fundament, the very same ordered vertex vector as depicted in Figure 8. Every element of the chromosome, a gene, corresponds to a single vertex. Thus adjacent processes in the graph are also in vicinity in the chromosome. Possible gene values, or alleles, are 1 for hardware and 0 for software. Two selection schemes are provided, tournament and roulette wheel selection, of which the first showed better convergence. Mating is performed via two-point crossover recombination. Mutation is implemented as an allele flip with a probability 0.03 per gene. The population size is set to $2|\mathcal{V}|$, and the termination criterion is fulfilled after $2|\mathcal{V}|$ generations without improvement. These GA mechanisms have

Table 3: Results obtained for the four algorithms on 180 different graphs.

		GA			GCLP			TS			RRES ($W = 10$)		
		$\overline{\Omega}$	$\overline{\sigma}$	$\overline{\Psi}$	$\overline{\Omega}$	$\overline{\sigma}$	$\overline{\Psi}$	$\overline{\Omega}$	$\overline{\sigma}$	$\overline{\Psi}$	$\overline{\Omega}$	$\overline{\sigma}$	$\overline{\Psi}$
$\lvert\mathcal{V}\rvert = 20$	Strict	**2.52**	0.17	**92.2**%	3.07	0.25	69.6%	2.56	0.19	83.4%	2.56	0.18	85.5%
	loose	2.07	0.11	100%	2.74	0.17	88.9%	2.09	0.11	100%	**2.06**	0.12	100%
$\lvert\mathcal{V}\rvert = 50$	Strict	2.76	0.19	81.6%	3.11	0.11	72.5%	2.77	0.19	77.5%	**2.70**	0.18	**93.3**%
	loose	2.21	0.12	100%	2.67	0.11	97.5%	2.23	0.12	100%	**2.17**	0.11	100%
$\lvert\mathcal{V}\rvert = 100$	Strict	2.84	0.19	66.4%	3.70	0.37	25.2%	2.81	0.20	62.4%	**2.70**	0.17	**99.4**%
	loose	2.28	0.12	100%	2.80	0.17	93.0%	2.22	0.12	100%	**2.16**	0.12	100%

been selected according to similar GA implementations in literature [11, 48].

The next algorithm to benchmark against is the *penalty reward* TS from Wiangtong et al. [9]. It combines a short term memory, that is, the tabu list of recently visited regions of the search space, with a long term memory, such that frequently visited regions of the search space are penalised (*diversification*), and regions that frequently return high-quality solution are rewarded (*intensification*). An aspiration criterion is provided, which ensures that globally best solutions are accepted, even when they are flagged tabu. According to their work, we chose an identical parameter set: neighbourhood size is $S_N = \lfloor\sqrt{|\mathcal{V}|/2}\rfloor$, and the number of tabu degrees is $N_{td} = \lfloor\sqrt{|\mathcal{V}|/2}\rfloor$, so that the obtained tabu list length is $L_T = S_N N_{td} = |\mathcal{V}|/2$. The long term memory covers a range of 10 vertices, corresponding to sufficient memory for 2^{10} different regions. The termination criterion is fulfilled after $4|\mathcal{V}|$ iterations without improvement.

The third algorithm GCLP of the Ptolemy I framework features a with a very low algorithmic complexity $\mathcal{O}(|\mathcal{V}|^2)$. Its core structure is a breadth first search, that visits every process once and decides instantaneously its implementation type. The decision is led by a superposition of two characteristic values: the global criticality and the local phase value. The first gives an indication whether time, area, or code size are most critical at the current stage of the algorithm based on the decision about already mapped vertices and estimations about yet unmapped vertices. The local phase value of a vertex is an individual indicator that expresses its tendency to be implemented in either sw or hw. This superposition moderates the greediness of the concept to more balanced solutions that meet all the constraints. For exact details, please refer to the literature [8, 12].

Table 3 contains selected information about the performance of the four algorithms on all graph sets. Table 3 shows the averaged results for all graphs with the sizes $|\mathcal{V}| = 20, 50, 100$. The termination criteria of GA and TS and the window length of RRES had been adjusted so that their runtimes do not differ by more than 25%. The best values are shown in bold. On first inspection, the results expose advantages for RRES both in cost and validity for these graph structures. The GCLP algorithm trails by more than 25% in Ω and Ψ, but is 50 to 100 times faster. Consequently, this algorithm is a reasonable candidate for very large graphs, $|\mathcal{V}| > 200$, because only then its very low runtime proves truly advantageous.

A more sophisticated picture of the algorithms performance can be obtained if we plot the averaged cost values of GA, TS, and RRES of all graphs and all sizes over their rank-locality metric r_{loc} and the parallelism metric γ, respectively. Recall that we identified typical system graphs in the field to have rather low values for r_{loc} and a low to medium value for γ, while having low values for ρ. The metric κ has been calculated for all the sample graphs, and the performance of GA, TS, and RRES has been plotted against this characteristic value, as shown in Figure 11. The $\overline{\Omega}$ values are normalised to the minimum cost value $\overline{\Omega}_{\min} = \min(\overline{\Omega}_{\text{GA}}, \overline{\Omega}_{\text{TS}}, \overline{\Omega}_{\text{RRES}})$ so that the performance differences are given in percentages above 1.0. For low values of κ, the RRES algorithm yields significantly better results up to 7%. Its performance degrades slowly until it drops back behind GA for values larger than 50 and behind TS for values larger than 80. Interestingly, GA loses performance as well, for values larger than 130, hence giving an hint why Wiangtong found his TS version better suited as GA. This behaviour of GA becomes clear when we recall the intricacies of its chromosome coding. Due to the fundamental schema theorem [47] of genetic algorithms, adjacent genes in the chromosome coding have to reflect a corresponding locality in the system graph. With growing values of κ, the mapping of the graph onto the two-dimensional chromosome vector decreasingly mirrors the vicinity of the vertices within the graph. Hence GA proves very sensitive to badly fitting chromosome codings, whereas TS remains insensitive to higher values of the product κ.

Figure 12 shows the quality dependency of RRES over W and the runtime Θ in clock cycles for the graph set with $|\mathcal{V}| = 100$ in comparison to the GA. The constraint set is loose. The shaded area illustrates where RRES outperforms GA both in quality and runtime. But it is apparent that the window length should lie below 14 for the binary mapping problem.

Consequently, a relevant aspect is the consideration of the extended mapping problem when more than two implementation alternatives exist. It is obvious that the runtime of the RRES algorithm would suffer greatly from an increasing number of implementation alternatives. Assume for every process in the design exist four implementation alternatives instead of two, for instance, another DSP is made available and two FPGA implementations for a process exist trading off area versus execution time. As the runtime is then proportional to $(|\mathcal{V}| - W)4^{(W-1)}$, the window length has to be halved to keep the runtime approximately constant with

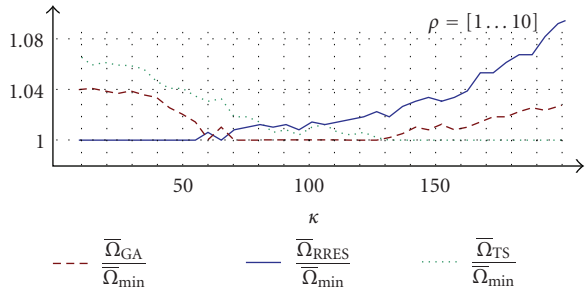


Figure 11: Dependency between the metric κ and the obtained averaged cost values for GA, TS, and RRES.

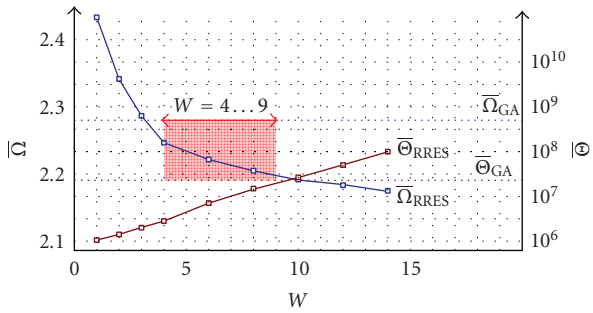


Figure 12: Quality and runtime of RRES and GA over window length for graphs with $|\mathcal{V}| = 100$.

respect to the binary case. From Figure 12, such a bisection (from $W = 10$ to $W = 5$) may still look acceptable, but it is clear that for an average number of implementation alternatives greater than four per process, RRES becomes quickly infeasible.

7. CONCLUSIONS

In this work a new heuristic for the hardware/software partitioning problem has been introduced. A thorough analysis of its behaviour related to graph properties revealed a strong performance for a distinct subset of system graphs typical in the field of electronic system design. For this subset and the binary mapping problem, the proposed RRES algorithm clearly outperforms three other popular techniques based on the concept of genetic algorithms, Wiangtong's *penalty reward* tabu search, and the well-reputed GCLP algorithm of Kalavade and Lee.

A mandatory step is the modification of RRES to the extended partitioning problem when there are more than two possible implementation types per vertex. Future work will scrutinise the run time of the RRES algorithm by revising the incremental movement of the RRES window. It may be possible to identify situations, in which more than one vertex could be mapped per movement of the window. Another interesting idea is the implementation of a short term memory for the moving window, in which the implementation type of vertices is fixed precociously due to their contribution to the recently found global best solutions.

ACKNOWLEDGMENT

This work has been funded by the Christian Doppler Laboratory for Design Methodology of Signal Processing Algorithms.

REFERENCES

[1] Y. Neuvo, "Cellular phones as embedded systems," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC '04)*, vol. 1, pp. 32–37, San Francisco, Calif, USA, February 2004.

[2] J. Hausner and R. Denk, "Implementation of signal processing algorithms for 3G and beyond," *IEEE Microwave and Wireless Components Letters*, vol. 13, no. 8, pp. 302–304, 2003.

[3] R. Subramanian, "Shannon vs Moore: driving the evolution of signal processing platforms in wireless communications," in *Proceedings of IEEE Workshop on Signal Processing Systems (SIPS '02)*, p. 2, San Diego, Calif, USA, October 2002.

[4] International SEMATECH, "International Technology Roadmap for Semiconductors," 1999, http://www.sematech.org/.

[5] M. Rupp, A. Burg, and E. Beck, "Rapid prototyping for wireless designs: the five-ones approach," *Signal Processing*, vol. 83, no. 7, pp. 1427–1444, 2003.

[6] P. Belanović, B. Knerr, M. Holzer, G. Sauzon, and M. Rupp, "A consistent design methodology for wireless embedded systems," *EURASIP Journal on Applied Signal Processing*, vol. 2005, no. 16, pp. 2598–2612, 2005.

[7] M. Holzer, B. Knerr, P. Belanović, and M. Rupp, "Efficient design methods for embedded communication systems," *EURASIP Journal of Embedded Systems*, vol. 2006, Article ID 64913, 18 pages, 2006.

[8] A. Kalavade and E. A. Lee, "The extended partitioning problem: hardware/software mapping, scheduling, and implementation-bin selection," in *Readings in Hardware/Software Co-Design*, pp. 293–312, Morgan Kaufmann, San Francisco, Calif, USA, 2002.

[9] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "Comparing three heuristic search methods for functional partitioning in hardware-software codesign," *Design Automation for Embedded Systems*, vol. 6, no. 4, pp. 425–449, 2002.

[10] K. S. Chatha and R. Vemuri, "MAGELLAN: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs," in *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES '01)*, pp. 42–47, ACM Press, Copenhagen, Denmark, April 2001.

[11] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of the 11th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC '00)*, Veldhoven, The Netherlands, November-December 2000.

[12] A. Kalavade, *System-level codesign of mixed hardware-software systems*, Ph.D. thesis, University of California, Berkeley, Calif, USA, 1995.

[13] P. Arató, Z. Á. Mann, and A. Orbán, "Algorithmic aspects of hardware/software partitioning," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 1, pp. 136–156, 2005.

[14] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design & Test of Computers*, vol. 10, no. 4, pp. 64–75, 1993.

[15] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," *Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 5–32, 1997.

[16] K. S. Chatha and R. Vemuri, "An iterative algorithm for hardware-software partitioning, hardware design space exploration and scheduling," *Design Automation for Embedded Systems*, vol. 5, no. 3-4, pp. 281–293, 2000.

[17] R. P. Dick and N. K. Jha, "MOGAC: a multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD '97)*, pp. 522–529, IEEE Computer Society, San Jose, Calif, USA, November 1997.

[18] Y. Zou, Z. Zhuang, and H. Chen, "HW-SW partitioning based on genetic algorithm," in *Proceedings of the Congress on Evolutionary Computation (CEC '04)*, vol. 1, pp. 628–633, Portland, Ore, USA, June 2004.

[19] G. de Micheli, R. Ernst, and W. Wolf, *Readings in Hardware/Software Co-Design*, Morgan Kaufman, San Francisco, Calif, USA, 2002.

[20] P. Marwedel, *Embedded System Design*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2003.

[21] F. Vahid and T. D. Le, "Extending the kernighan/lin heuristic for hardware and software functional partitioning," *Design Automation for Embedded Systems*, vol. 2, no. 2, pp. 237–261, 1997.

[22] "CoWare SPW 4," tech. rep., CoWare Design Systems, 2004, http://www.coware.com/products/signalprocessing.php.

[23] E. A. Lee, "Overview of the ptolemy project," Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, Calif, USA, March 2001, http://ptolemy.eecs.berkeley.edu/.

[24] The MathWorks Simulink, http://www.mathworks.com/products/simulink/.

[25] E. A. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, 1987.

[26] E. A. Lee, *A coupled hardware and software architecture for programmable digital signal processors*, Ph.D. thesis, EECS Department, University of California, Berkeley, Calif, USA, 1986.

[27] W. Haas, M. Hofstaetter, T. Herndl, and A. Martin, "Umts baseband chip design," in *Informationstagung Mikroelektronik*, pp. 261–266, Vienna, Austria, October 2003.

[28] M. Holzer and M. Rupp, "Static estimation of execution times for hardware accelerators in system-on-chips," in *Proceedings of International Symposium on System-on-Chip (SoC '05)*, pp. 62–65, Tampere, Finland, November 2005.

[29] J. P. Singh, A. Kumar, and S. Kumar, "A multiplier generator for Xilinx FPGAs," in *Proceedings of the 9th International Conference on VLSI Design (ICVD '96)*, pp. 322–323, Bangalore, India, January 1996.

[30] C. Brandolese, W. Fornaciari, and F. Salice, "An area estimation methodology for FPGA based designs at systemC-level," in *Proceedings of the 41st Design Automation Conference (DAC '04)*, pp. 129–132, San Diego, Calif, USA, June 2004.

[31] H. Posadas, F. Herrera, P. Sánchez, E. Villar, and F. Blasco, "System-level performance analysis in systemC," in *Proceedings of Design, Automation and Test in Europe Conference & Exhibition (DATE '04)*, vol. 1, pp. 378–383, Paris, France, February 2004.

[32] Mentor Graphics, http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/index.cfm.

[33] J. Hromkovič, *Algorithmics for Hard Problems*, Springer, New York, NY, USA, 2nd edition, 2004.

[34] M. Garey and D. Johnson, *Computers and Intractability: A Guide to NP-Completeness*, W.H. Freeman, San Francisco, Calif, USA, 1979.

[35] W. H. Kohler and K. Steiglitz, "Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems," *Journal of the ACM*, vol. 21, no. 1, pp. 140–156, 1974.

[36] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961.

[37] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal on Computing*, vol. 18, no. 2, pp. 244–257, 1989.

[38] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, 1993.

[39] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996.

[40] Radioscape, "Radiolab 3g," Licensable Block Set for The MathWorks MATLAB and Simulink products, 1999.

[41] "Standard task graph set," Kasahara Laboratory, Department of Electrical Engineering, Waseda University, 2006.

[42] R. Sedgewick, *Algorithms in C++, Part 5: Graph Algorithms*, Addison-Wesley, Reading, Mass, USA, 3rd edition, 2002.

[43] Y. Le Moullec, N. B. Amor, J.-P. Diguet, M. Abid, and J.-L. Philippe, "Multi-granularity metrics for the era of strongly personalized SOCs," in *Proceedings of Design, Automation and Test in Europe Conference & Exhibition (DATE '03)*, pp. 674–679, Munich, Germany, March 2003.

[44] M. Holzer and B. Knerr, "Pareto front generation for a trade-off between area and timing," in *Proceedings of the the Austrian National Conference on the Design of Integrated Circuits and Systems (Austrochip '06)*, pp. 131–134, Vienna, Austria, October 2006.

[45] B. Knerr, M. Holzer, and M. Rupp, "Improvements of the gclp algorithm for hw/sw partitioning of task graphs," in *Proceedings of the 4th IAESTED International Conference on Circuits, Signals, and Systems (CSS '06)*, pp. 107–113, San Francisco, Calif, USA, November 2006.

[46] I. Chakravarti, R. Laha, and J. Roy, *Handbook of Methods of Applied Statistics*, vol. 1, John Wiley & Sons, New York, NY, USA, 1967.

[47] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Boston, Mass, USA, 1989.

[48] V. Srinivasan, S. Radhakrishnan, and R. Vemuri, "Hardware software partitioning with integrated hardware design space exploration," in *Proceedings of Design, Automation and Test in Europe (DATE '98)*, pp. 28–35, IEEE Computer Society, Paris, France, February 1998.

[49] P.-A. Mudry, G. Zufferey, and G. Tempesti, "A dynamically constrained genetic algorithm for hardware-software partitioning," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pp. 769–776, ACM Press, Seattle, Wash, USA, July 2006.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 231940, 11 pages
doi:10.1155/2008/231940

Research Article

Software-Controlled Dynamically Swappable Hardware Design in Partially Reconfigurable Systems

Chun-Hsian Huang and Pao-Ann Hsiung

Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi 621, Taiwan

Correspondence should be addressed to Pao-Ann Hsiung, hpa@computer.org

Received 24 May 2007; Accepted 15 October 2007

Recommended by Toomas P. Plaks

We propose two basic wrapper designs and an enhanced wrapper design for arbitrary digital hardware circuit designs such that they can be enhanced with the capability for dynamic swapping controlled by software. A hardware design with either of the proposed wrappers can thus be swapped out of the partially reconfigurable logic at runtime in some intermediate state of computation and then swapped in when required to continue from that state. The context data is saved to a buffer in the wrapper at interruptible states, and then the wrapper takes care of saving the hardware context to communication memory through a peripheral bus, and later restoring the hardware context after the design is swapped in. The overheads of the hardware standardization and the wrapper in terms of additional reconfigurable logic resources and the time for context switching are small and generally acceptable. With the capability for dynamic swapping, high priority hardware tasks can interrupt low-priority tasks in real-time embedded systems so that the utilization of hardware space per unit time is increased.

1. INTRODUCTION

With rapid technology progress, FPGAs are getting more and more powerful and flexible in contrast to inflexible ASICs. FPGAs, such as Xilinx Virtex II/II Pro, Virtex 4, and Virtex 5, can now be partially reconfigured at run time for achieving higher system performance. Partially reconfigurable systems enable more applications to be accelerated in hardware, and thus reduces the overall system execution time [1]. This technology can now be used in real-time embedded systems for switching from a low-priority hardware task to a high-priority hardware task. However, hardware circuits are generally not designed to be switched or swapped in and out, as a result of which partial reconfigurability either becomes useless or incur significant time overhead.

In this work, we try to bridge this gap by proposing generic wrapper designs for hardware IPs such that they can be enhanced with the capability for dynamic swapping. The dynamically swappable design must solve several issues related to switching hardware IPs, including the following. (1) When must a hardware design be interrupted for switching? (2) How and where must we save the context of a hardware design? (3) How must we restore the context of a hardware design? (4) How to make the wrapper design small, efficient, and generic? (5) How must a hardware IP be modified so that it can interact with the wrapper.

For ease of explanation, henceforth we call a running hardware circuit as a hardware task. To swap out a hardware task so that it can be swapped in later, one needs to save its execution context so that it can be restored in the future. However, different from software processes, hardware tasks cannot be interrupted in each and every state of computation. Hence, a hardware task should be allowed to run until the next interruptible state, which is function-specific. The context of a hardware task is also function-specific. Nevertheless, we can use the memento design pattern [2] from software engineering, which states that the context of a task can be stored outside in a memento and then restored when the task is reloaded. We adopted this design pattern to hardware task context. To restore a saved context, the context data needs to be preloaded into the wrapper, which then loads the data to the registers in the hardware task. The wrapper architectures are generic so that any digital hardware IP that has been automatically standardized, can be interfaced with

it for dynamic swapping. The wrappers receive the software request signals through a task interface and then drive the appropriate signals to prepare the hardware task for swapping. However, the original hardware IP also needs to be enhanced so that it can interface with the wrapper, which we call *standardization*. The detailed descriptions of the wrappers and the hardware task modification are given in Section 4.

This work contributes to the state-of-the-art in the following ways.

(1) *Generic Wrapper Designs*: these proposed generic wrapper designs can be used to interface with any standardized hardware IP, thus they are reusable and reduce IP development effort significantly. We propose three different wrapper designs to get higher performance and using lesser resources under different conditions.
(2) *Swappable Hardware IP*: a hardware IP needs only to be enhanced slightly and interfaced with the wrappers for dynamic swapping.
(3) *Better Real-Time Response*: compared to state-of-the-art methods, our method saves hundreds of microseconds, which give better real-time response during the hardware-software scheduling in an operating system for reconfigurable systems.

This paper is organized as follows: Section 2 discusses related research work and compares them with our architecture. Section 3 describes the architecture of our target platform. The details of the dynamically swappable architecture are given in Section 4. A case study is used for illustrating how to make an unswappable DCT IP swappable in Section 5. We use six applications to demonstrate the validity and genericity of the architecture in Section 6. Finally, conclusions and future work are described in Section 7.

2. RELATED WORK

For partially reconfigurable systems, dynamic switching or relocation of hardware designs has been investigated in several previous work, which can be categorized into two classes, namely *reconfiguration-based* [3, 4] and *design-based* [5, 6]. Reconfiguration-based dynamic hardware switching requires no change to the hardware design that is being switched because the context is saved and restored by accessing the configuration port such that state information are extracted from the readback data stream and restored by manipulating the bitstream that is configured into the logic. Design-based dynamic hardware switching needs a switching circuitry and enhanced register access structures for context saving and restoring.

The reconfiguration-based method requires readback support from the reconfigurable logic and deep knowledge of the reconfiguration process for tasks such as state extraction from the readback stream and manipulation of the bitstreams for context restoring. As a result, this method becomes technology-dependent and thus nonportable. Another drawback is the poor data efficiency because only a maximum of about 8% of the readback data actually contains state information but all data must be readback to extract the state [4]. This data efficiency issue has been partially resolved in [3] through online state extraction and inclusion filters, but the readback time is still in the same order of magnitude as that of full data readback.

The design-based method is self-sufficient because all context switching tasks are taken care of by the hardware design itself through a switching circuitry and registers can be read out or preloaded by the switching circuitry. This method is thus technology-independent and data efficient. Only the required data are read out instead of the full data stream, which could be as large as 1,026 KB for the Xilinx Virtex II Pro XC2VP20-FF896 FPGA chip, requiring totally 20 milliseconds.

Our proposed method for dynamic hardware switching falls into the design-based category, however, we try to eliminate some of the deficiencies of this method, while retaining the advantages. Our method proposes two basic wrapper designs and an enhanced wrapper design with different standard interfaces such that any digital hardware design following the standard can be transformed automatically into dynamically switchable by interfacing with the wrappers. The proposed method can also be applied to a third-party hardware IP that was designed without following the standard, as long as we have the RTL model of the IP. Using our proposed method, we have thus not only retained the advantages of data efficiency and technology independence of design-based methods, but also acquired the advantage of reconfiguration-based methods, that is, minimal user design effort for making a hardware IP dynamically reconfigurable.

Another major contribution of this work is the design and implementation of the proposed dynamic reconfiguration framework for general hardware IPs, which most previous work in the design-based category has either delegated its implementation to future work directions [7, 8], or implemented it for application-specific cases such as the CAN-bus architecture for automobiles in [5], the hardware-software task relocation for a video decoder in [6], and the dynamic hardware-software partitioning for a DSP algorithm in [9].

Abstraction of tasks from its hardware or software implementation requires an operating system that can manage both software and hardware tasks. Such an *operating system for reconfigurable systems* (OS4RS) is an important infrastructure for successful dynamic reconfiguration. There have been several works in this regard [6, 10–13], though the actual implementations of such an OS4RS still lack generality and wide-acceptance.

3. DYNAMICALLY RECONFIGURABLE SYSTEMS

A dynamically reconfigurable system is a hardware-software system which is capable of saving and restoring the context of any system task, as and when required by the scheduler, with possible relocation. A system task is a basic unit of execution in a system and can be executed preemptively either in hardware or software depending on whether we have a configurable bitstream or an executable code. If we have both implementations, a task could switch from hardware to software and vice versa provided the system infrastructure supports it [6]. In this work, we focus on how a general hardware IP can

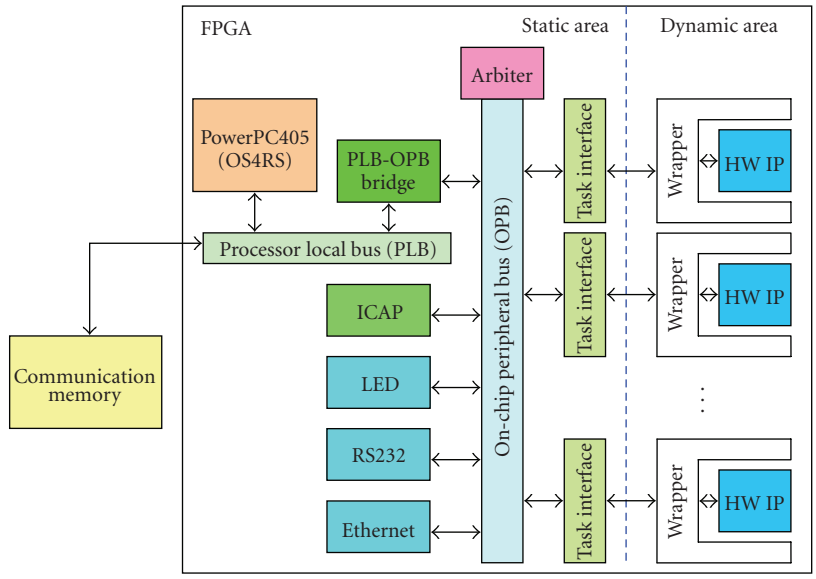


Figure 1: Dynamically reconfigurable system.

be made dynamically reconfigurable such that the context of a hardware task can be saved and restored.

The dynamically reconfigurable system, as illustrated in Figure 1, consists of a microprocessor attached to a system bus with a communication memory, and a dynamically reconfigurable logic component such as FPGA, within which hardware tasks can be configured and attached to a peripheral bus which in turn is connected through a bridge with the system bus. Each hardware task consists of a hardware IP, a wrapper, and a task interface. The hardware IP is an application-specific function such as a DCT or an H.264 video decoder. In this work, two basic wrapper designs and an enhanced wrapper design are proposed for dynamically swappable design and the implementation of one of them along with the standardizing hardware IP is used for demonstrating the practicality. The wrappers control the whole swap-out and swap-in processes of the hardware task. The task interface is an interface to a peripheral bus for data transfers in a hardware task. The task interface acts as a bus interface of the hardware task and is responsible for normal data transfer operations through the control, read, and write interfaces and for swapping and reconfiguration operations through the swap interface.

In this work, our target system is based on the Xilinx Virtex II Pro FPGA chip, with the IBM CoreConnect on-chip bus architecture. Swappable hardware tasks are attached to the *on-chip peripheral bus* (OPB), while the microprocessor and memory are attached to the *processor local bus* (PLB). The FPGA chip consists of *configurable logic blocks* (CLB), I/O blocks (IOB), embedded memory, routing resources, and an *internal configuration access port* (ICAP). Reconfiguration is achieved through the ICAP by configuring a bitstream.

The software task management in an OS4RS is similar to that in a conventional OS. The hardware task management is as shown in Figure 2, where the OS4RS uses a priority scheduling and placement algorithm. Each hardware task has a priority, arrival time, execution time, reconfiguration time, deadline, and area in columns. The OS4RS schedules and places the hardware tasks to be executed by swapping them into the reconfigurable logic and it also preempts running tasks by swapping them out and storing their contexts to the external communication memory. Reconfigurable resources, including CLB, IOBs, and routing resources, are managed and reconfiguration is controlled by the OS4RS through the ICAP.

4. DYNAMICALLY SWAPPABLE DESIGN

Given the dynamically reconfigurable system architecture described in Section 3, we focus on how a digital hardware IP can be automatically transformed into a dynamically swappable hardware task. For a nonswappable hardware IP, three major modifications required to make it swappable include the standardization of the hardware IP for interfacing with a generic wrapper, the wrapper design itself for swapping the hardware IP, and a task interface for interfacing with the peripheral bus.

4.1. Standardizing hardware IP

Since a combinational circuit is stateless, it can be swapped out from the reconfigurable logic as soon as it finishes the current computation. However, a sequential circuit is controlled by a *finite state machine* (FSM) through the present and next state registers. Generally, a hardware IP has one or more data registers for storing intermediate results of computation. The collection of the state registers and data registers constitutes the *task context*. A state is said to be *interruptible* if the hardware task can resume execution from that state after restoring the task context, either partially or fully. Not all states of a hardware task are interruptible. For the FSM of a GCD IP example given in Figure 3, only the INIT, RLD, and

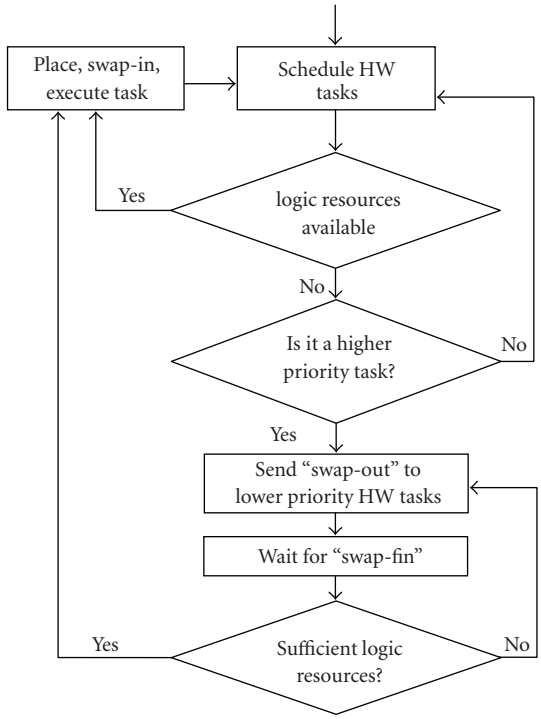


Figure 2: Hardware task scheduling and reconfiguration in an OS4RS.

CMP states are interruptible because the comparator results are not saved and hence we cannot resume from the NEG, EQ, and POS states.

The initial or the idle state is always interruptible. Any other state of a hardware IP can be made interruptible by adding or reusing registers provided the computation can be resumed after context restoring. However, extra resources are required, thus the benefit obtained by making a state interruptible should be weighed against the overhead incurred in terms of both logic resources and context saving and restoring time. In general, making a state interruptible allows the hardware task to be switched at that state, and thus the delays in executing other hardware tasks are reduced. Hence, making a state interruptible brings no benefit to the task itself, instead it may shorten the overall system schedule. The decision to make a state interruptible must be derived from an overall system analysis rather than from the perspective of the hardware task itself.

A hardware IP is standardized automatically by making the context registers accessible by the wrapper and by enhancing the FSM controller such that the IP can be stalled at each interruptible state. This is done in the same way as *design for test* (DFT) techniques that perform scan-chain insertions after the design is completed. Tool support is planned for the future. For the GCD IP, its standardized version that is dynamically swappable is shown in Figure 3, where the two registers are made accessible to the wrapper (swap circuitry) and the FSM is modified such that the IP can be stalled in the CMP state. Furthermore, a standardized hardware IP needs to be combined with either one of the basic wrapper designs [14] or an enhanced wrapper design for being enhanced with the capability for dynamic swapping. Two basic wrapper designs and an enhanced wrapper design are introduced and their interfacing is illustrated as follows.

4.2. Basic wrapper designs

Two basic wrapper architectures, namely *last interruptible state swap* (LISS) wrapper and *next interruptible state swap* (NISS) wrapper, are proposed for controlling the swapping of a hardware circuit into and out from a reconfigurable logic such that all swap circuitry is implemented within the wrappers with minimal changes to the hardware IP itself. As shown in Figure 4, the wrapper architectures consist of a *context buffer* (CB) to store context data, a data path for data transfer, a *swap controller* (SC) to manage the swap-out and swap-in activities, and some optional *data transformation components* (DTCs) for (un)packing data types. A generic wrapper architecture interfaces with a hardware IP and a standard task interface that connects with a peripheral bus. The difference between the two wrappers lies in the swap-out mechanism and the hardware state in which the IP is swapped out. The LISS wrapper stores the IP context at each interruptible state, thus the IP can be swapped out from the last interruptible state whenever there is a swap request. The NISS wrapper requires the IP to execute until the next interruptible state, store the context, and then swap out. In Figure 4, the LISS wrapper does not include the W_interrupt and swap_fin signals, while the NISS wrapper does (signals are highlighted using dotted arrows). The different swap-out processes and the same swap-in process are described as follows.

4.2.1. LISS wrapper swap-out

At every interruptible state, the context of hardware IP is stored in a *context buffer* using the Wout_State and Wout_cdata signals. When there is a swap_out request from the OS4RS for some hardware task, the wrapper sends an Interrupt signal to the microprocessor to notify the OS4RS that (1) the context data stored in the context buffer can be read and saved into the *communication memory*, and (2) the resources can be deallocated and reused (reconfigured). The swap-out process is thus completed. This wrapper can be used for hardware circuits whose context data size is less than that of the context buffer, as a result of which all context data can be stored in the context buffer using a single data transfer.

4.2.2. NISS wrapper swap-out

When there is a swap_out request from the OS4RS for some hardware task, the *swap controller* in the wrapper sends a swap signal (asserted high) to the hardware IP, which starts the whole swap_out process. However, the hardware IP might be in an unswappable state, thus execution is allowed to continue until the next swappable state is reached. At a swappable state, the context of hardware IP, including current state information and selected register data, is stored in a *context buffer* in the wrapper using the Wout_State and

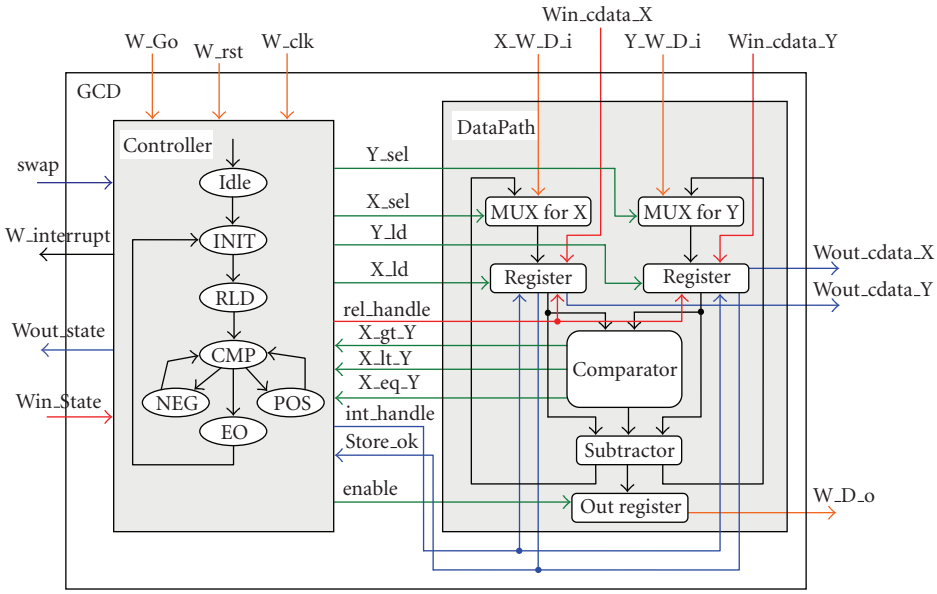


Figure 3: Swappable GCD circuit architecture.

Wout_cdata signals. The hardware IP then sends an acknowledgment W_interrupt to the wrapper that the swap-out process can continue. The wrapper sends an Interrupt signal to the microprocessor to notify the OS4RS that the context data stored in the context buffer can be read and saved into the *communication memory*. This wrapper can be used when the context data size is larger than that of the context buffer by repeating the process of storing into buffer, interrupting microprocessor, and reading into memory. Finally, when all context data have been stored into the communication memory, the wrapper sends a swap_fin signal to the task interface, thus notifying the OS4RS that the resources occupied by the IP can be deallocated and reused. The swap-out process is thus completed.

4.2.3. Swap-in

When a hardware task is scheduled to be executed, the OS4RS configures the corresponding hardware IP with wrapper and task interface into the reconfigurable logic using the *internal configuration access port* (ICAP), reloads the context data from the communication memory to the context buffer in the wrapper, and sends a swap_in request to the swap controller, which then starts to copy the context data from the buffer to the corresponding registers in the IP using Win_State and Win_cdata. After all context data are restored, the swap controller sends a swap signal (asserted low) to the hardware IP, which then continues from the state in which it was swapped out.

It must be noted here that context data might be of different sizes for different hardware IPs, so data packing and unpacking are performed using the *data transformation component* (DTC) within the wrapper. For the standardized GCD IP example given in Figure 3, there are two 8-bit X_Wout_cdata and Y_Wout_cdata signals from the IP, which are packed by the DTC in the wrapper into a 32-bit Out_context signal for storing into communication memory through the peripheral bus. The other signals in Figure 4 are used for normal IP execution.

4.3. Task interface

The task interface, as illustrated in Figure 4, acts as a bus interface of the hardware task. A task interface consists of a *read interface* and a *write interface* to control read and write operations, respectively, a *control interface* to manage IP-related control signals, a *swap interface* to manage the swapping process and reconfiguration of the hardware design, a *bus control interface* to deal with the interactions between the bus and above interfaces. The task interface presently supports the CoreConnect OPB only. The PowerPC 405 and communication memory are bound on the CoreConnect PLB bus, where the PowerPC 405 can interact with the hardware tasks on the OPB bus by utilizing the PLB-OPB bridge as shown in Figure 1. The PLB-OPB bridge is the OPB master and it is responsible for communicating the signals from the PowerPC 405 to the hardware tasks, while the swappable hardware tasks along with wrappers are the OPB slaves. In the future, we will design different task interfaces for other peripheral buses such as AMBA APB. By changing the task interface, a swappable hardware IP can be connected to different peripheral buses.

4.4. Enhanced wrapper design along with OPB IPIF

In this section, an enhanced LISS wrapper along with OPB *intellectual property interface* (IPIF) architecture is proposed, where the OPB IPIF architecture provides additional optional services to standardize functionality that is common to many hardware IPs and to reduce hardware IP development effort. As shown in Figure 5, a swappable hardware design

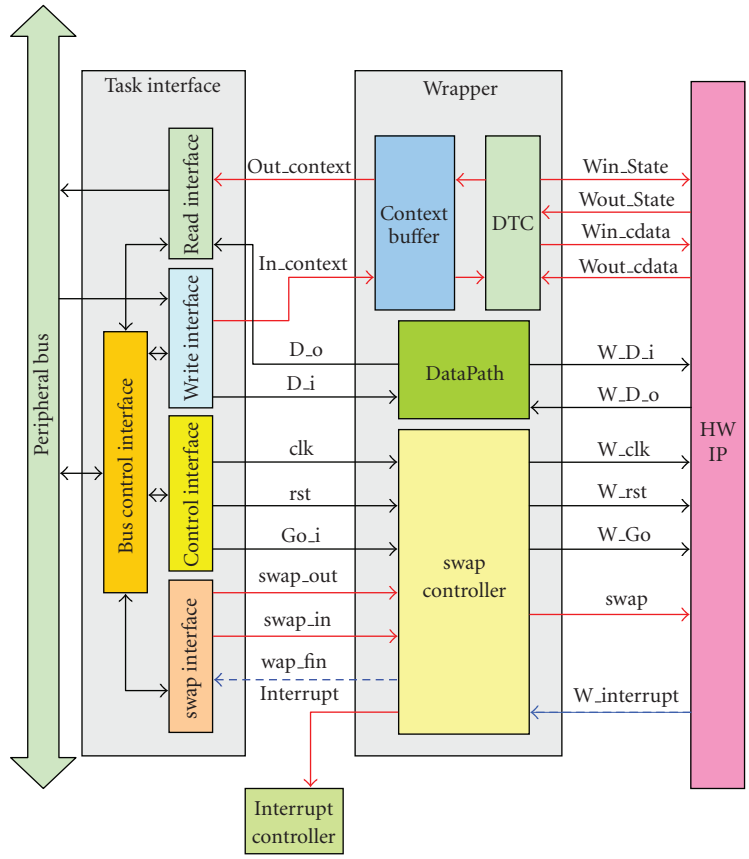


Figure 4: Wrapper architecture and interfaces.

along with the enhanced LISS wrapper is specified as an OPB IPIF slave, where the OPB IPIF architecture consists of a *reset* component to reset a hardware IP, an *Addr_decode* to decode the OPB address, a *slave interface* to provide software accessible registers, a *Write FIFO* and a *Read FIFO* for write and read data transfers, respectively, and an *IP interconnect* (IPIC) for connecting the user logic to the IPIF services.

For this enhanced LISS wrapper design, the basic data transfers are directly accessed by the *slave interface* instead of the *datapath* in the LISS wrapper, while the context data is stored in the *Write FIFO* and *Read FIFO* in place of the *context buffer* in the LISS wrapper. The DTC component in the wrapper is responsible for (un)packing data type, where the signals In_context and Out_context are used for transferring context data packages from *Write FIFO* or to *Read FIFO*. By using the Xilinx EDK tool [15], the size of *Write FIFO* and *Read FIFO* can be adjusted to fit that of the context data, which makes context data transfers to be not only unrestricted by the context buffer size, but also to provide the capability of dealing with larger context data size similar to the NISS wrapper. Furthermore, when using the Xilinx EDK tool, the number of software accessible registers is decided according to the swap-out and swap-in activities, the data transfers of a hardware IP, and all required control signals.

The swap-in and swap-out processes of the enhanced LISS wrapper are similar to those of the LISS wrapper in addition to the signal swap_fin for notifying the OS4RS to read the context data in the *Read FIFO*, instead of the signal Interrupt in the LISS wrapper. In order to demonstrate the feasibility of our swappable hardware design, a swappable IP with our enhanced LISS wrapper design, which is implemented on the Xilinx ML310 embedded development platform [16], will be introduced in Section 5.

5. CASE STUDY: A SWAPPABLE DCT HARDWARE TASK

As shown in Figure 6, a design flow for dynamically swappable hardware design is proposed, and a *discrete cosine transform* (DCT) IP with our enhanced LISS wrapper design, which is implemented on the Xilinx ML310 embedded development platform, is used for illustrating how to make an unswappable DCT IP swappable.

A DCT IP transforms an image having 128 blocks of size 8×8 pixels, in which a block is read and saved at a time into an 8×8 array, called Block_i. Another 8×8 array, called Block_o, is used for saving the results, where each result is produced in turn using all data of Block_i. After analyzing the DCT design, the context data, including all data of Block_i and the row and column indices of the present

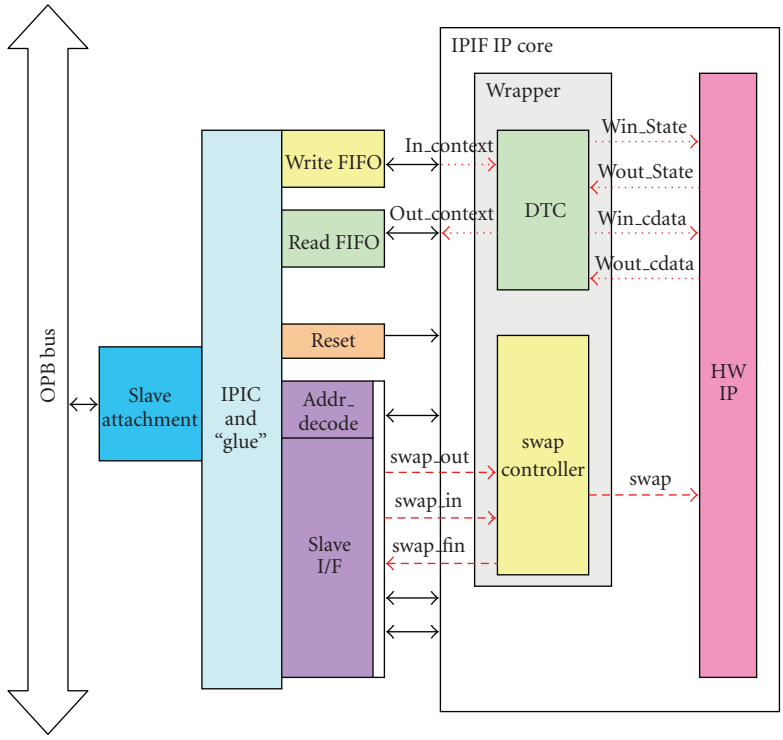


Figure 5: Enhanced LISS wrapper architecture.

iteration, are recorded. The DCT IP needs to be standardized for accessing the context data as shown in Figure 7, and combined with our enhanced LISS wrapper, as shown in Figure 5, by connecting with the Win_State, Wout_State, Win_cadta, Wout_cdata, and swap signals. By using Xilinx EDK tool, a swappable DCT hardware task, including a swappable DCT IP and our enhanced LISS wrapper, is designed as a slave attached to the OPB bus, where the size of FIFOs and the number of software accessible registers are decided according to the analysis results of context data.

The design flow for swappable hardware design is illustrated in Figure 6 and designed on follows. Owing to the Xilinx EDK tool being suitable only for full chip design, the netlist of the swappable DCT IP with the wrapper is extracted. Furthermore, the HDL of top module is modified for fitting the constraint on partial reconfiguration design flow and bus macros are added to reconnect a swappable DCT hardware task with the OPB bus. Finally, the netlist of the new top module is regenerated. After following the above process and then using the partial reconfiguration design flow [17], the full bitstream and the partial bitstream of swappable DCT task are generated. The design flow for dynamically swappable hardware design is thus completed. The complete result of a dynamically swappable DCT hardware task in a partially reconfigurable system is shown in Figure 8, where the dynamic module of the swappable DCT hardware task, and the static module including two PowerPC405 microprocessors, an ICAP, a PLB bus, and an OPB bus, and the bus macros for connecting the dynamic module with the static module, are highlighted for displaying the relative location of each component in the FPGA.

6. EXPERIMENTS

In order to demonstrate the feasibility of our proposed swappable hardware design, six different hardware IPs are used for analyzing the overhead of IP standardization and comparing the time for context switching with that required by reconfiguration-based method.

6.1. *Resource overhead analysis*

We performed all our experiments on the Xilinx Virtex II Pro XC2VP20-FF896 FPGA chip that is organized as a CLB matrix of 56 rows and 46 columns, including 18,560 LUTs and 18,560 flip-flops. All swappable hardware tasks are connected to a 32-bit CoreConnect OPB bus operating at 133 MHz. For the experiments, we synthesized and simulated the swappable versions of the hardware IPs. The OS4RS running on the PowerPC was based on an in-house extension of the Linux OS. There was no specific application running to avoid inaccuracies in experimental results.

We standardized six different hardware IPs, as described in Section 4.1, implemented the generic wrappers, as discussed in Sections 4.2 and 4.4. We used the Synplify synthesis tool and the ModelSim simulator to verify the correctness of the wrapper and the modified hardware IP designs. We compared the original hardware IP designs with the new swappable ones for each example.

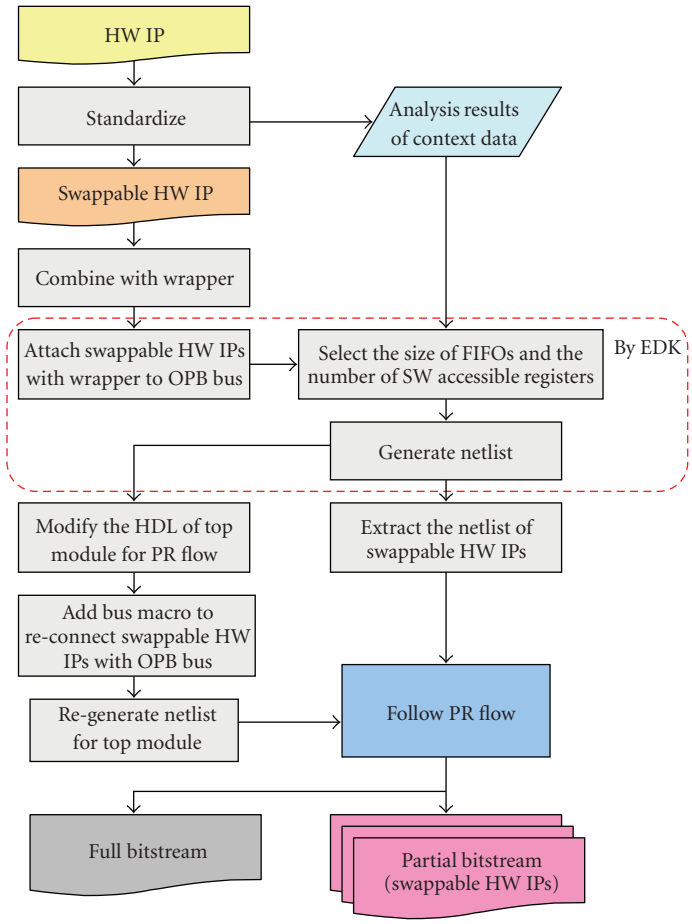


Figure 6: Design flow for swappable hardware designs.

The examples included two GCDs as shown in Figure 3, a traffic light controller (TLC), a multiple lights controller (MLC), and a *data encryption standard* (DES) design, and a DCT as shown in Figure 7. The GCD can be swapped out in the middle of calculating the greatest common divisor of two 8-bit or 32-bit integers and swapped in to continue the computation. The computation results were verified correct for all test cases. The TLC drives the red light for 9 clock cycles, the yellow light for 2 clock cycles, and the green light for 6 clock cycles. The TLC can be swapped out and continue from where it left. The MLC is an extension of the TLC with more complex light switching schemes. The DES is a more complex design that can effectively demonstrate the practicality of the proposed swappable design. The DCT design transforms an image having 128 blocks of size 8×8 pixels. All the IPs were made swappable, interfaced with the wrapper and the swapping was verified correct in the sense that they finished their computations correct irrespective of when they were interrupted.

The resource overhead required for making a hardware IP swappable includes the extra resources required to make the context registers and the current state register visible. Our synthesis results and comparisons are given in Table 1, where making a hardware IP to interact with the enhanced LISS wrapper and that with the LISS wrapper are the same so that the first three examples include only two cases. We can observe that the overheads in making the IPs swappable seem to be around 60% for the simple 8-bit GCD and the TLC examples, while for the more complex 32-bit GCD and MLC examples the overhead is only 22%~33%, which shows that the overhead in resources depends only on the amount of context data to be saved and restored and the number of interruptible states, and does not depend on the complexity of the full hardware design. The original DES design is synthesized into thirty-two 64×1 ROMs. Making the DES design swappable, it needs an extra 51% or 47% flip-flops but only 2% LUTs, in terms of the available FPGA resources, the overhead is quite small. The swappable DCT needs 33% more flip-flops, but -13% or -14% less LUTs due to synthesis compiler optimization. One can observe that flip-flop overheads are high, but the LUTs overheads are low. The increase in flip-flop is mainly due to the need for extra I/O registers for storing context data. However, since there are usually a large number of unused flip-flops in the CLBs of a synthesized circuit, the design after placement and routing will not result in a significant increase in the CLB count. The reduction in LUTs after standardization of the DCT circuit is due to all context registers being made accessible in

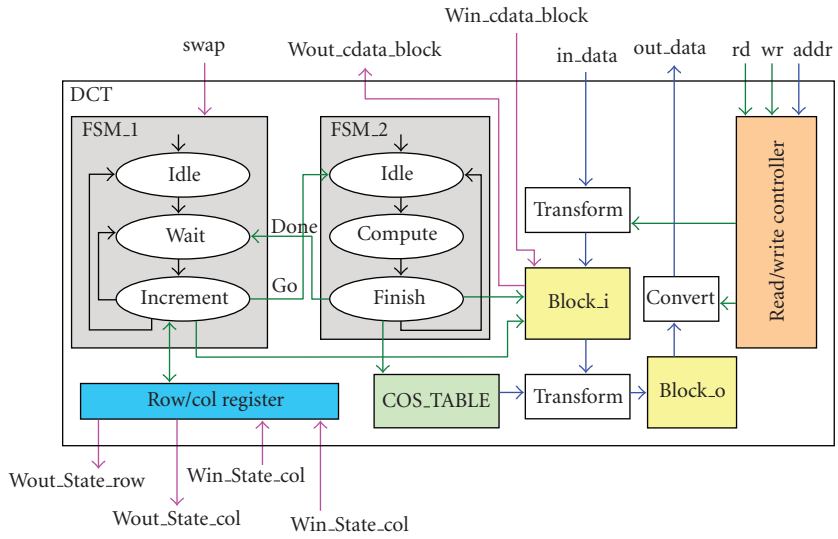


Figure 7: Swappable DCT circuit architecture.

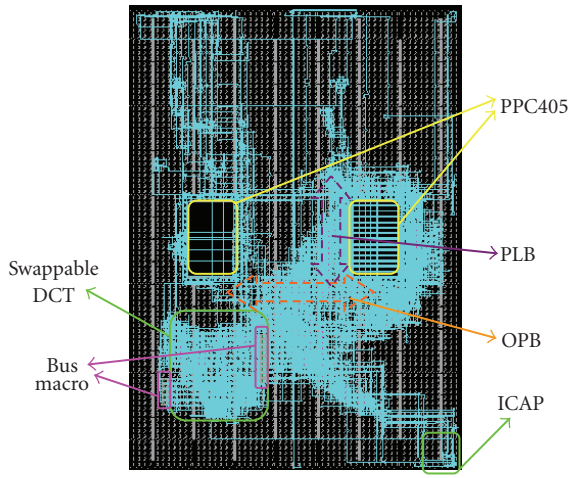


Figure 8: Swappable DCT design along with enhanced LISS wrapper.

Table 1: Synthesis results and resource overheads.

HW	V	D_C (bits)	FF IP	FF SIP	FF +%	LUT IP	LUT SIP	LUT +%
TLC	*N*	3	6	10	66	24	43	79
	L/E			10	66		39	62
MLC	*N*	3	13	17	30	63	77	22
	L/E			17	30		77	22
G_8	*N*	19	31	53	70	80	122	52
	L/E			48	54		114	42
G_{32}	*N*	67	103	169	64	270	360	33
	E			168	63		365	35
DES	*N*	836	137	207	51	589	603	2
	E			202	47		603	2
DCT	*N*	1030	1573	2094	33	1339	1152	−13
	E			2103	33		1140	−14

V: version, D_C: context data size, G_8: 8-bit GCD, G_{32}: 32-bit GCD, *L*: LISS wrapper, *N*: NISS wrapper, *E*: enhanced LISS wrapper, IP: IP resource usage, SIP: swappable IP resource usage, +%: % of overheads in SIP compared to IP.

parallel, which results in the elimination of multipliers and multiplexers and thus fewer LUTs in the swappable circuit. The complex DCT design more explicitly shows the feasibility of our proposed swappable design. For task G_8, the FF and LUT overheads are 54% and 42% for LISS, and 70% and 52% for NISS, respectively. We can observe that the overheads in making the IPs swappable for interfacing with the LISS wrapper are smaller than that for interfacing with the NISS wrapper. This is due to the lesser number of signals in LISS wrapper and the more complex circuitry in NISS wrapper for transferring context data of sizes greater than that of context buffer. The implementation results obviously show that the extra FPGA resources required for making a hardware IP swappable are only dependent on the amount of context data and the number of interruptible states, where the amount of resource overhead compared to the original hardware IP are getting lesser and lesser for more and more complex hardware designs, and when compared to the total available FPGA resource the overheads are negligible.

6.2. Efficiency analysis

We now analyze the performance of the proposed wrappers. Given context data of D_C-bits, context buffer of D_B-bits, each FIFO entry of D_F-bits, data transformation rate of R_T bits/cycle, buffer data load rate of R_B bits/cycle, FIFO entry load rate of R_F bits/cycle, peripheral bus data transfer rate of R_P bits/cycle, peripheral bus access time of T_A cycles,

Table 2: Time overheads for swap-out and swap-in.

	V	T_E	swap-out			swap-in			T_R	T_{SO}	T_{SI}	Task relocate	
			T_B	T_P	T'_{SO} (ns)	T_B	T_P	T'_{SI} (ns)	(ns)	(μs)	(μs)	Our (μs)	RMB (μs)
TLC	N	17	3	3	64	2	3	50	46,336	46.4	46.3	92.7	496.7
	L		2	3	39	2	3	39	42,025	46.4	46.3	92.7	
	E		2	3	39	2	3	39		46.4	46.3	92.7	
MLC	N	33	3	3	64	2	3	50	83,243	83.3	83.2	166.5	582.5
	L		2	3	50	2	3	50	83,243	83.3	83.2	166.5	
	E		2	3	50	2	3	50		83.3	83.2	166.5	
G_8	N	511	4	3	46	2	3	38	131,465	131.5	131.5	263.0	619.9
	L		2	3	38	2	3	38	122,844	131.5	131.5	263.0	
	E		2	3	38	2	3	38		131.5	131.5	263.0	
G_{32}	N	1671	11	9	157	5	9	108	387,931	388.0	388.0	776.0	1038.1
	E		9	9	140	3	9	92	401,589	401.7	401.6	803.3	
DES	N	1,424	84	81	962	55	81	840	649,784	650.7	650.6	1301.3	2183.8
	E		58	81	917	29	81	763	649,784	650.7	650.6	1301.3	
DCT	N	71,552	100	99	1,600	66	99	1,309	1,267,481	1269.0	1268.7	2537.8	4278.2
	E		68	99	1,292	34	99	1,018	1,254,278	1255.5	1255.2	2510.7	

RBM: Reconfiguration-based method, T_E: execution time (in IP clock cycles), $T_B = (D_B/R_T) + (D_B/R_B)$ or $T_B = (D_F/R_T) + (D_F/R_F)$ (in IP clock cycles), $T_P = T_A + (D_B/R_P)$ or $T_P = T_A + (D_F/R_P)$ (in bus cycles), $T'_{SO} = T_{SO} - T_R$ (in nanoseconds), $T'_{SI} = T_{SI} - T_R$ (in nanoseconds).

transition time of T_I cycles to go to an interruptible state (T_I is 0 for LISS), and reconfiguration time of T_R cycles, the swap-out and swap-in processes require time T_{SO} and T_{SI}, respectively, for both the NISS and the LISS wrappers as shown in (1), while that for the enhanced LISS wrapper is as shown in (2):

$$T_{SO} = T_I + \left\lceil \frac{D_C}{D_B} \right\rceil \times \left(\frac{D_B}{R_T} + \frac{D_B}{R_B} + T_A + \frac{D_B}{R_P} \right) + T_R,$$
$$T_{SI} = T_R + \left\lceil \frac{D_C}{D_B} \right\rceil \times \left(\frac{D_B}{R_T} + \frac{D_B}{R_B} + T_A + \frac{D_B}{R_P} \right). \quad (1)$$

Both swap times are dominated by the reconfiguration time T_R. For Xilinx XC2VP20-FF896 FPGA chip, the reconfiguration clock runs at 50 MHz such that a byte can be configured in 20 nanoseconds, however a full bitstream is 1,026,820 bytes, which means a full chip configuration requires around 20 milliseconds. However, all other times in (1) and (2) are only a few cycles, in the nanoseconds order of magnitude. The wrapper overhead as shown in the experiments accounts for at most 2 cycles assuming that the context buffer can be loaded in 1 cycle. Our design-based dynamic reconfiguration approach is very data-efficient because the readback time required by reconfiguration-based methods [3, 4] is also in the same order of magnitude as the reconfiguration time T_R:

$$T_{SO} = \left\lceil \frac{D_C}{D_F} \right\rceil \times \left(\frac{D_F}{R_T} + \frac{D_F}{R_F} + T_A + \frac{D_F}{R_P} \right) + T_R,$$
$$T_{SI} = T_R + \left\lceil \frac{D_C}{D_F} \right\rceil \times \left(\frac{D_F}{R_T} + \frac{D_F}{R_F} + T_A + \frac{D_F}{R_P} \right). \quad (2)$$

As shown in Table 2, the time overheads in swapping out and swapping in for all the examples consume only a few cycles and are in the order of nanoseconds. From Table 2, we can observe that not only is swapping faster with the LISS wrapper or the enhanced LISS wrapper, but their simpler circuities also require lesser reconfiguration time T_R, compared to NISS. However, as mentioned before, LISS wrappers can only be used when the IP context size is not greater than that of the context buffer size, but the enhanced LISS wrapper can be unrestricted to the context buffer size and efficient than the NISS wrapper when the IP context size is greater than that of the context buffer size. We can thus conclude that the enhanced LISS wrapper is suitable for dynamically swappable hardware design irrespective of the context data size. It is assumed here that $T_I = 0$ because the time to transit to a swappable state is not a fixed one and depends on when the OS4RS sends in the swap signals. We assume typical OPB read and write data transfers for swap-out and swap-in, respectively; hence, each of them needs 3 bus cycles for a single 32-bit data transfer. Comparing the time required for a task relocation, that is, one swap-out and one swap-in, our proposed design-based method performs better than the reconfiguration-based methods (RBM) [3]. From the experimental results, RBM methods not only require a reconfiguration time of 648 microseconds for DES and 1473.2 μs for DCT, but they also require a readback time of 887.8 μs for DES and 1331.8 microseconds for DCT, while we reduce 40.4% and 40.6% for the NISS wrapper, and 40.4% and 41.3% for the enhanced LISS wrapper, respectively, of the time required by reconfiguration-based methods, respectively, for the larger DES and DCT examples. We are thus saving much time, which is important for hard real-time systems. Even though additional reconfiguration time

is required, the swappable design would enable more hardware tasks to fit their deadline constraints, which makes the hardware-software scheduling in an OS4RS more flexible for achieving higher system performance.

7. CONCLUSIONS

We have proposed a method for the automatic modification and enhancement of a hardware IP such that it becomes dynamically swappable under the control of an operating system for reconfigurable systems. We have designed two basic wrapper designs and an enhanced LISS wrapper design, and analyzed the conditions for using the wrappers. We have also proposed how the hardware IP can be minimally changed by only making the state and context registers visible. The proposed method and architectures were implemented and verified. Our experiment results show that the resource and time overheads of making an IP swappable are quite small compared to the amount of reconfigurable resources available and the configuration time of the IP, respectively.

REFERENCES

[1] Xilinx. XAPP290—two flows for partial reconfiguration module-based or difference-based, 2004.

[2] R. Gamma, R. Helm, R. Johnson, and J. Vissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Addison-Wesley, Reading, Mass, USA, 1994.

[3] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL '05)*, vol. 2005, pp. 223–228, Tampere, Finland, August 2005.

[4] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA coprocessors," in *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL '00)*, pp. 121–130, Villach, Austria, August 2000.

[5] M. Ullmann, B. Grimm, M. Hübner, and J. Becker, "An FPGA run-time system for dynamical on-demand reconfiguration," in *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW '04)*, Santa Fe, NM, USA, April 2004.

[6] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of the Design Automation and Test in Europe (DATE '03)*, vol. 1, pp. 986–991, Munich, Germany, March 2003.

[7] G. Brebner, "The swappable logic unit: a paradigm for virtual hardware," in *Proceedings of the 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FPGA '97)*, pp. 77–86, Napa Valley, Calif, USA, April 1997.

[8] 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.

[9] D. Kearney and R. Kiefer, "Hardware context switching in a signal processing application for an FPGA custom computer," in *Proceedings of the 4th Australasian Computer Architecture Conference (ACAC '99)*, pp. 35–46, Auckland, New Zealand, January 1999.

[10] H.-Y. Sun, "Dynamic hardware-software task switching and relocation for reconfigurable systems," M.S. thesis, Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, 2007.

[11] P.-A. Hsiung, C.-H. Huang, and Y.-H. Chen, "Hardware task scheduling and placement in operating systems for dynamically reconfigurable SoC," to appear in *Journal of Embedded Computing*.

[12] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*, p. 174, Nice, France, April 2003.

[13] 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.

[14] C.-H. Huang, K.-J. Shih, C.-S. Lin, S.-S. Chang, and P.-A. Hsiungt, "Dynamically swappable hardware design in partially reconfigurable systems," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '07)*, pp. 2742–2745, New Orleans, La, USA, May 2007.

[15] Xilinx. Embedded system tools reference manual—embedded development kit EDK 8.1i, 2005.

[16] Xilinx. ML310 User Guide, 2007.

[17] Xilinx. UG208—Early Access Partial Reconfiguration User Guide, 2006.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 562326, 13 pages
doi:10.1155/2008/562326

Research Article

DART: A Functional-Level Reconfigurable Architecture for High Energy Efficiency

Sébastien Pillement,[1] Olivier Sentieys,[1] and Raphaël David[2]

[1] *IRISA/R2D2, 6 Rue de Kerampont, 22300 Lannion, France*
[2] *CEA, LIST, Embedded Computing Laboratory, Mailbox 94, F-91191 Gif-sur-Yvette, France*

Correspondence should be addressed to Sébastien Pillement, sebastien.pillement@irisa.fr

Received 4 June 2007; Accepted 15 October 2007

Recommended by Toomas P. Plaks

Flexibility becomes a major concern for the development of multimedia and mobile communication systems, as well as classical high-performance and low-energy consumption constraints. The use of general-purpose processors solves flexibility problems but fails to cope with the increasing demand for energy efficiency. This paper presents the DART architecture based on the functional-level reconfiguration paradigm which allows a significant improvement in energy efficiency. DART is built around a hierarchical interconnection network allowing high flexibility while keeping the power overhead low. To enable specific optimizations, DART supports two modes of reconfiguration. The compilation framework is built using compilation and high-level synthesis techniques. A 3G mobile communication application has been implemented as a proof of concept. The energy distribution within the architecture and the physical implementation are also discussed. Finally, the VLSI design of a 0.13 x2009μm CMOS SoC implementing a specialized DART cluster is presented.

1. INTRODUCTION

Rapid advances in mobile computing require high-performance and energy-efficient devices. Also, flexibility has become a major concern to support a large range of multimedia and communication applications. Nowadays, digital signal processing requirements impose extreme computational demands which cannot be met by off-the-shelf, general-purpose processors (GPPs) or digital signal processors (DSPs). Moreover, these solutions fail to cope with the ever increasing demand for low power, low silicon area, and real-time processing. Besides, with the exponential increase of design complexity and nonrecurring engineering costs, custom approaches become less attractive since they cannot handle the flexibility required by emerging applications and standards. Within this context, reconfigurable chips such as field programmable gate arrays (FPGAs) are an alternative to deal with flexibility, adaptability, high performance, and short time-to-market requirements.

FPGAs have been the reconfigurable computing mainstream for a couple of years and achieved flexibility by supporting gate-level reconfigurability; that is, they can be fully optimized for any application at the bit level. However, the flexibility of FPGAs is achieved at a very high silicon cost interconnecting huge amount of processing primitives. Moreover, to be configured, a large number of data must be distributed via a slow programming process. Configurations must be stored in an external memory. These interconnection and configuration overheads result in energy waste, so FPGAs are inefficient from a power consumption point of view. Furthermore, bit-level flexibility requires more complex design tools, and designs are mostly specified at the register-transfer level.

To increase optimization potential of programmable processors without the fine-grained architectures penalties, functional-level reconfiguration was introduced. *Reconfigurable processors* are a more advanced class of reconfigurable architectures. The main concern of this class of architectures is to support high-level flexibility while reducing reconfiguration overhead.

In this paper, we present a new architectural paradigm which aims at associating flexibility with performance and low-energy constraints. High-complexity application domains, such as mobile telecommunications, are particularly

targeted. The paper is organized as follows. Section 2 discusses mechanisms to reduce energy waste during computations. Similar approaches in the context of reconfigurable architectures are presented and discussed in Section 3. Section 4 describes the features of the DART architecture. The dynamic reconfiguration management in DART is presented in Section 5. The development flow associated with the architecture is then introduced. Section 7 presents some relevant results coming from the implementation of a mobile telecommunication receiver using DART and compares it to other architectures such as DSP, FPGA, and a reconfigurable processor. Finally, Section 8 details the VLSI (very large-scale integration) implementation results of the architecture in a collaborative project.

2. ENERGY EFFICIENCY OPTIMIZATION

The energy efficiency (EE) of an architecture can be defined by the number of operations it performs when consuming 1 mW of power. EE is therefore proportional to the computational power of the architecture given in MOPS (millions of operations per second) divided by the power consumed during the execution of these operations. The power is given by the product of the elementary dissipated power per area unit P_el, the switching frequency F_{clk}, the square of the power supply voltage V_{DD}, and the chip area. The latter is the sum of the operator area, the memory area, and the area of the control and configuration management resources. P_el is the sum of two major components: dynamic power which is the product of the transistor average activity and the normalized capacitance per area unit, and static power which depends on the mean leakage of each transistor.

These relations are crucial to determine which parameters have to be optimized to design an energy-efficient architecture. The computational power cannot be reduced since it is constrained by the application needs. Parameters like the normalized capacitance or the transistor leakage mainly depend on technology process, and their optimization is beyond the scope of this study.

The specification of an energy-efficient architecture dictates the optimization of the remaining parameters: the operator area, the storage and control resources area, as well as the activity throughout the circuit and the supply voltage. The following paragraphs describe some useful mechanisms to achieve these goals.

2.1. Exploiting parallelism

Since EE depends on the square of the supply voltage, V_{DD} has to be reduced. To compensate for the associated performance loss, full use must be made of parallel processing.

Many application domains handle several data sizes during different time intervals. To support all of these data sizes, flexible functional units must be designed, at the cost of latency and energy penalties. Alternatively, functional units can be optimized for only a subset of these data sizes. Optimizing functional units for 8- and 16-bit data sizes allows to design subword processing (SWP) operators [1]. Thanks to these operators, the computational power of the architecture can be increased during processing with data-level parallelism, without reducing overall performances at other times.

Operation- or instruction-level parallelism (ILP) is inherent in computational algorithms. Although ILP is constrained by data dependencies, its exploitation is generally quite easy. It requires the introduction of several functional units working independently. To exploit this parallelism, the controller of the architecture must specify simultaneously to several operators the operations to be executed as in very long instruction word (VLIW) processors.

Thread-level parallelism (TLP) represents the number of threads which may be executed concurrently in an algorithm. TLP is more complicated to be exploited since it strongly varies from one application to another. The tradeoff between ILP and TLP must thus be adapted for each application running on the architecture. Consequently, to support TLP while guaranteeing a good computational density, the architecture must be able to alter the organization of its processing resources [2].

Finally, application parallelism can be considered as an extension of thread parallelism. The goal is to identify the applications that may run concurrently on the architecture. Contrary to threads, applications executed in parallel run on distinct datasets. To exploit this level of parallelism, the architecture can be divided into clusters which can work independently. These clusters must have their own control, storage, and processing resources.

Exploiting available parallelism efficiently (depending on application) can allow for some system-level optimization of the energy consumption. The allocation of tasks can permit the putting of some part of architecture into idle or sleep modes [3] or the use of other mechanisms like clock gating to save energy [4].

2.2. Reducing the configuration distribution cost

Control and configuration distribution has a significant impact on the energy consumption. Therefore, the configuration data volume as well as the configuration frequency must both be minimized. The configuration data volume reflects on the energy cost of one reconfiguration. It may be minimized by reducing the number of reconfiguration targets. Especially, the interconnection network must support a good tradeoff between flexibility and configuration data volume. Hierarchical networks are perfect for this purpose [5].

If there are some redundancies in the datapath structure, it is possible to reduce the configuration data volume, by distributing simultaneously the same configuration data to several targets. This has been defined as the single configuration multiple data (SCMD) concept. The basic idea was first introduced in the Xilinx 6200 FPGA. In this circuit, configuring "cells" in parallel with the same configuration bits were implemented using wildcarding bits to augment the cell address/position to select several cells at the same time for reconfiguration.

The 80/20 rule [6] asserts that 80% of the execution time are consumed by 20% of the program code, and only 20% are consumed by the remaining source code. The time-consuming portions of the code are described as being

regular and typically nested loops. In such a portion of code, the same computation pattern is repeated many times. Between loop nests, the remaining irregular code cannot be optimized due to lack of parallelism. Adequate configuration mechanisms must thus be defined for these opposite kinds of processing.

2.3. Reducing the data access cost

Minimizing the data access cost implies reducing the number of memory accesses and the cost of one memory access. Thanks to functional-level reconfiguration, operators may be interconnected to exploit temporal and spatial localities of data. Spatial locality is exploited by connecting operators in a data-flow model. Producers and consumers of data are directly connected without requiring intermediate memory transactions. In the same way, it is important to increase the locality of reference, and so to have memory close to the processing part.

Temporal locality may be exploited—thanks to broadcast connections. This kind of connection transfers one item of data towards several targets in a single transaction. This removes multiple accesses to data memories. The temporal locality may further be exploited—thanks to registers used to build delay chains. These delay chains reduce the number of data memory accesses when several samples of the same vector are concurrently handled in an application.

To reduce data memory access costs while providing a high bandwidth, a memory hierarchy must be defined. The high-bandwidth and low-energy constraints dictate the integration of a large number of small memories. To provide large storage space, a second level of hierarchy must be added to supply data to the local memories. Finally, to reduce the memory management cost, address generation tasks have to be distributed along with the local memories.

3. RELATED WORKS

Functional-level reconfigurable architectures were introduced to trade off flexibility against performance, while reducing the reconfiguration overhead. This latter is mainly obtained using reconfigurable operators instead of LUT-based configurable logic blocks. Precursors of this class of architectures were KressArray [7], RaPid [8], and RaW machines [9] which were specifically designed for streaming algorithms.

These works have led to numerous academic and commercial architectures. The first industrial product was the Chameleon Systems CS2000 family [10], designed for application in telecommunication facilities. This architecture comprises a GPP and a reconfigurable processing fabric. The fabric is built around identical processing tiles including reconfigurable datapaths. The tiles communicate through point-to-point communication channels that are static for the duration of a kernel. To achieve a high throughput, the reconfigurable fabric has a highly pipelined architecture. Based on a fixed 2D topology of interconnection network, this architecture is mainly designed to provide high speeds in the telecommunication domain regardless of other constraints.

The extreme processor platform (XPP) [11] from PACT is based on a mesh array of coarse-grained processing array elements (PAEs). PAEs are specialized for algorithms of a particular domain on a specific XPP processor core. The XPP processor is hierarchical, and a cluster contains a 2D array of PAEs, which can support point-to-point or multicast communications. PAEs have input and output registers, and the data streams need to be highly pipelined to use the XPP resources efficiently.

The NEC dynamically reconfigurable processor (DRP-1) [12] is an array of tiles constituted by an 8×8 matrix of processing elements (PEs). Each PE has an 8-bit ALU, an 8-bit data management unit, and some registers. These units are connected by programmable wires specialized by instruction data in a point-to-point manner. Local data memories are included on the periphery of each tile. Data flow needs to be carefully designed to take advantage of this architecture. NEC DRP-1 provides sixteen contexts, by implementing a 16-deep instruction memory in each PE. This approach permits the reconfiguration of the processor in one cycle, but at the price of a very high cost in configuration memory.

The XiRisc architecture [13] is a reconfigurable processor based on a VLIW RISC core with a five-stage pipeline, enhanced with an additional run-time configurable datapath, called pipelined configurable gate array (PiCoGA). PiCoGA is a full-custom designed unit composed of a regular 2D array of multicontext fine-grained reconfigurable logic cells (RLCs). Thus, each row can implement a stage of a customizable pipeline. In the array, each row is connected to other rows with configurable interconnection channels and to the processor register file with six global busses. Vertical channels have 12 pairs of wires, while horizontal ones have only 8 pairs of wires. PiCoGA supports dynamic reconfiguration in one cycle by including a specific cache, storing four configurations for each RLC. The reconfiguration overhead can be optimized by exploiting partial run-time reconfiguration, which gives the opportunity for reprogramming only a portion of the PiCoGA.

Pleiades [14] was the first reconfigurable platform taking into account the energy efficiency as a design constraint. It is a heterogeneous coarse-grained platform built around satellite processors which communicate through a hierarchical reconfigurable mesh structure. All these blocks communicate through point-to-point communication channels that are static for the duration of a kernel. The satellite processors can be embedded FPGAs, configurable operators, or hardwired IPs to support specific operations. Pleiades is designed for low power but it needs to be restricted to an application domain to be very efficient. The algorithms in the domain are carefully profiled in order to find the kernels that will eventually be implemented as a satellite processor.

Finally, the work in [15] proposes some architectural improvements to define a low-energy FPGA. However, for complex applications, this architecture is limited in terms of attainable performance and development time.

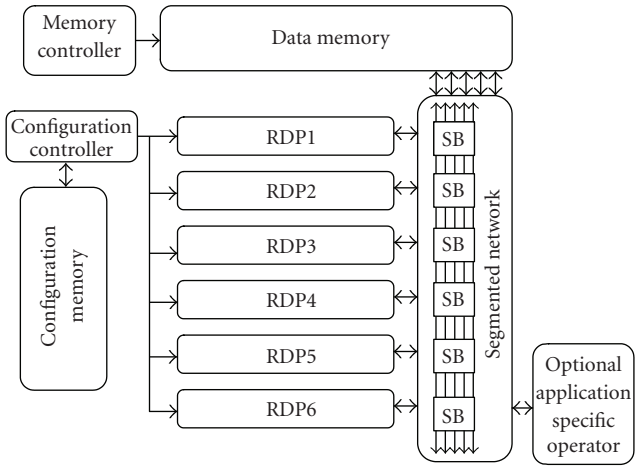


Figure 1: Architecture of a DART cluster.

4. DART ARCHITECTURE

The association of the principles presented in Section 3 leads to the first definition of the DART architecture [16]. Two visions of the system level of this architecture can be explored. The first one consists in a set of autonomous clusters which have access to a shared memory space, managed by a task controller. This controller assigns tasks to clusters according to priority and resources availability constraints. This vision leads to an autonomous reconfigurable system. The second one, which is the solution discussed here, consists in using one cluster of the reconfigurable architecture as a hardware accelerator in a reconfigurable system-on-chip (RSoC). The RSoC includes a general-purpose processor which should support a real-time operating system and control the whole system through a configurable network. At this level, the architecture deals with the application-level parallelism and can support operating system optimization such as dynamic voltage and frequency scaling.

4.1. Cluster architecture

A DART cluster (see Figure 1) is composed of functional-level reconfigurable blocks called *reconfigurable datapaths* (RDPs); see Section 4.2.

DART was designed as a platform-based architecture so at the cluster level, we have a defined interface to implement user dedicated logic which allows for the integration of application-specific operators or an FPGA core to efficiently support bit-level parallelism, for example.

The RDPs may be interconnected through a segmented network, which is the top level of the interconnection hierarchy. According to the degree of parallelism of the application to be implemented, the RDPs can be interconnected to carry out high-complexity tasks or disconnected to work independently on different threads. The segmented network allows for dynamic adaptation of the instruction-level and thread-level parallelisms of the architecture, depending on the processing needs. It also enables communication between the application-specific core and the data memory or the chaining of operations between the RDPs and the user dedicated logic.

The hierarchical organization of DART allows the control to be distributed. Distributing control and processing resources through predefined hierarchical interconnection networks is more energy-efficient for large designs than that through global interconnection networks [5]. Hence, it is possible to efficiently connect a very large number of resources without being penalized too much by the interconnection cost.

All the processing primitives access the same data memory space. The main task of the configuration controller is to manage and reconfigure the RDP sequentially. This controller supports the above-mentioned SCMD concept. Since it sequences configurations rather than instructions, it does not have to access an instruction memory at each cycle. Memory reading and decoding do happen occasionally when a reconfiguration occurs. This drastic reduction of the amount of instruction memory reading and decoding leads to significant energy savings.

4.2. Reconfigurable datapath architecture

The arithmetic processing primitives in DART are the RDPs (see Figure 2). They are organized around functional units (FUs) followed by a pipeline register and small SRAM memories, interconnected via a powerful communication network. Each RDP has four functional units in the current configuration (two multipliers/adders and two arithmetic and logic units) supporting subword processing (SWP); see Section 4.3. FUs are dynamically reconfigurable and can execute various arithmetic and logic operations depending on the stored configuration.

FUs process data stored in four small local memories, on top of which four local controllers are in charge of providing the addresses of the data handled inside the RDPs. These address generators (AGs) share a zero-overhead loop support and they are detailed in Section 4.4. In addition to the memories, two registers are also available in every RDP. These registers are used to build delay chains, and hence realizing time data sharing.

All these resources communicate through a fully connected network. This offers high flexibility and it is the second level of the interconnection hierarchy. The organization of DART keeps these connections relatively small, hence limiting their energy consumption. Thanks to this network, resources can communicate with each other in the RDP. Furthermore, the datapath can be optimized for several kinds of calculation patterns and can make data sharing easier. Since a memory can simultaneously be accessed by several functional units, some energy savings can be realized. Finally, connections with global busses allow for the use of several RDPs to implement massively parallel processing.

4.3. Architecture of the functional units

The design of efficient functional units is of prime importance for the efficiency of the global architecture. DART is based on two different FUs which use the SWP [1] concept

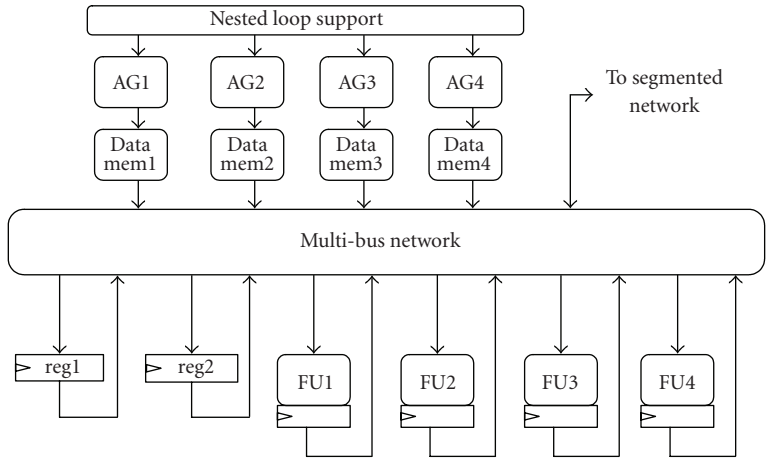


Figure 2: Architecture of a reconfigurable datapath (RDP).

justified by the numerous data sizes that can be found in current applications (e.g., 8 and 16 bits for video and audio applications). Consequently, we have designed arithmetic operators that are optimized for the most common data format (16 bits) but which support SWP processing for 8-bit data.

The first type of FU implements a multiplier/adder. Designing a low-power multiplier is difficult but well known [17]. One of the most efficient architectures is the *Booth-Wallace* multiplier for word lengths of at least 16 bits. The designed FU includes the saturation of signed results in the same cycle as the operation evaluation. Finally, as the multiplication has a 32-bit result, a shifter implements basic scaling of the result. This unit is shown in Figure 3.

As stated before, FUs must support SWP. Synthesis and analysis of various architectures have shown that implementing three multipliers (one for 16-bit data and two for the SWP processing on 8-bit data) leads to a better tradeoff between area, time, and energy than the traditional 4-multiplier decomposition [18].

To decrease switching activity in the FU, inputs are latched depending on whether SWP is used or not, leading to a 5% area overhead, but the power consumption is optimized (-23% for 16-bit operations and -72% for 8-bit multiplications). Implementing addition on the various multipliers is obvious and requires only a multiplexer to have access to the adder tree.

The second type of functional unit implements an arithmetic and logic unit (ALU) as depicted in Figure 4. It can perform operations like ADD, SUB, ABS, AND, XOR, and OR and it is mainly based on an optimized adder. For this latter, a Sklansky structure has been chosen due to its high performance and power efficiency 11. Implementation of subtraction is made by using two's complement arithmetic. Finally, SWP is implemented by splitting the tree structure of the Δ elements of the Sklansky adder. The FU has a 40-bit wide operator to limit overflow in the case of long accumulation. As for the multiplier, the unit can perform saturation in the same processing cycle.

Two shifters at the input and at the output of the arithmetic unit can perform left or right shifts of 0, 1, 2, or 4 bits in the same cycle to scale the data. As for the multiplier, inputs are latched to decrease switching activity. Table 1 summarizes performance results of the proposed functional units on 0.18 μm technology from STMicroelectronics (Geneva, Switzerland). The critical path of the global RDP comes from the ALU implementation, and so pipelining the multiplier unit is not an issue.

4.4. Address generation units

Since the controller task is limited to the reconfiguration management, DART must integrate some dedicated resources for address generation. These units must provide the addresses of the data handled in the RDPs for each data memory (see Figure 2) during the task processing. To be efficient in a large spectrum of applications, the address generators (AGs) must support numerous addressing patterns (bit reverse, modulo, pre-/postincrement, etc.). These units are built around an RISC-like core in charge of sequencing the accesses to a small instruction memory (64×32 bits). In order to minimize the energy consumption, these accesses will take place only when an address has to be generated. For that, the sequencer may be put in an idle state. Another module is then in charge of waking up the sequencer at the right time.

Even if this method needs some additional resources, interest in it is widely justified by the energy savings. Once the instruction has been read, it is decoded in order to control a small datapath that will supply the address. On top of the four address generation units of each RDP (one per memory), a module provides a zero-overhead loop support. Thanks to this module, up to four levels of nested loop can be supported, with each loop kernel being able to contain up to eight instructions without any additional cycles for its management. Two address generation units are represented in Figure 5 with the shared zero-overhead loop support.

5. DYNAMIC RECONFIGURATION

DART proposes a flexible and dynamic control of reconfiguration. The distinction between regular and irregular codes

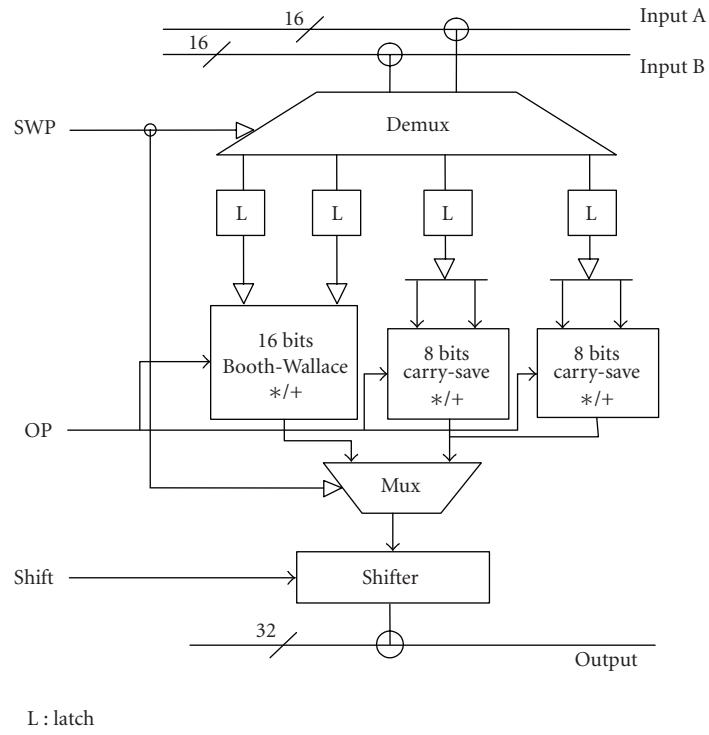


Figure 3: Multiplication functional unit.

leads to the definition of two reconfiguration modes. Regular processing is the time-consuming part of algorithms and it is implemented—thanks to "hardware reconfigurations" (see Section 5.1). On the other hand, irregular processing has less influence on performance and it is implemented—thanks to "software reconfigurations" (see Section 5.2).

5.1. Hardware reconfiguration

During regular processing, complete flexibility of the RDPs is provided by the full use of the functional-level reconfiguration paradigm at the cost of a higher reconfiguration overhead. In such a computation model, the dataflow execution paradigm is optimal. By allowing the modification of interconnections between functional units and memories, the architecture can be optimized for the computation pattern to be implemented. The SCMD concept exploits the redundancy of the RDPs by simultaneously distributing the same configuration to several RDPs, and thus reducing the configuration data volume. According to the regularity of the computation pattern and the redundancy of configurations, 4 to 19 52-bit instructions are required to reconfigure all the RDPs and their interconnections. Once these configuration instructions have been specified, no other instruction reading and decoding have to occur until the end of the loop execution. The execution is controlled by the AGs which sequence input data and save the output in terminal memories.

For example, in Figure 6, the datapath is configured to implement a digital filter based on MAC operations. Once this configuration has been specified, the dataflow computation model is maintained as long as the filter needs this pattern. At the end of the execution, a new computing pattern can be specified to the datapath, for example, the square of the difference between $x(n)$ and $x(n-1)$ in Figure 6. In that case, 4 cycles are required to reconfigure a single RDP. This hardware reconfiguration fully optimizes the datapath structure at the cost of reconfiguration time (19 cycles for the overall configuration without SCMD), and no additional control data are necessary.

5.2. Software reconfiguration

Irregular processing represents the control-dominated parts of the application and requires to change RDP configurations at each cycle; a so-called software reconfiguration is used. To reconfigure the RDPs in one cycle, their flexibility is limited to a subset of the possibilities. As in VLIW processors, a calculation pattern of *read-modify-write* type has been adopted. In that case, for each operator needed for the execution, the data are read and computed, then the result is stored back in memory.

The software reconfiguration is only concerned with the functionality of the operators, the size of the data, and their origin. Thanks to these limitations on flexibility, the RDP may be reconfigured at each cycle with only one 52-bit instruction. This is illustrated in Figure 7 which represents the reconfiguration needed to replace an addition of data stored in the memories Mem1 and Mem2 by a subtraction of data stored in the memories Mem1 and Mem4.

Due to the reconfiguration modes and the SCMD concept, DART can be fully optimized to efficiently support both dataflow intensive computation processing and irregular

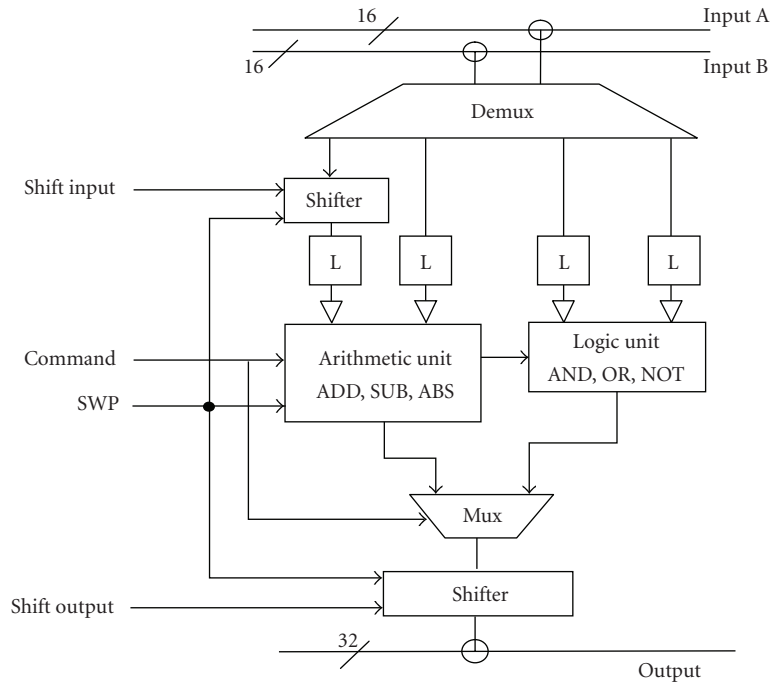


Figure 4: Arithmetic and logic functional unit.

Table 1: Implementation results and performances of the functional units.

	Area (μm^2)	Delay (ns)	Energy ($10^{-12}J$)
Multiplier functional unit	37 000	3.97	88.90
ALU functional unit	28 850	5.33	39.62

processing for control parts. Moreover, the two reconfiguration modes can be mixed without any constraints, and they have a great influence on the development methodology. Besides the design of the architecture, a compilation framework has been developed to exploit these architecture and reconfiguration paradigms. The joint use of retargetable compilation and high-level synthesis techniques leads to an efficient methodology.

6. DEVELOPMENT FLOW

To exploit the computational power of DART, the design of development flow is the key to enhance the status of the architecture. In that way, we developed a compilation framework based on the joint use of a front end allowing for the transformation and the optimization of C code, a retargetable compiler to handle compilation of the software configurations, and high-level synthesis techniques to generate the hardware reconfiguration of the RDP [19].

As in most of development methodologies for reconfigurable hardware, the key issue is to identify the different kinds of processing. Based on the two reconfiguration modes of the DART architecture, our compilation framework uses two separate flows for the regular and irregular portions of code. This approach has already been successfully used in the PICO (program in, chip out) project developed at HP labs to implement regular codes into a systolic structure, and to compile irregular ones for an VLIW processor [20]. Other projects such as Pleiades [21] or GARP [22] are also using this approach.

The proposed development flow is depicted in Figure 8. It allows the user to describe its applications in C. These high-level descriptions are first translated into control and dataflow graph (CDFG) by the front end, from which some automatic transformations (loop unrolling, loop kernel extractions, etc.) are done to reduce the execution time. After these transformations, the distinction between regular codes, irregular ones, and data manipulations permits the translation of the high-level description of the application into configuration instructions—thanks to compilation and architectural synthesis.

6.1. Front end

The front end of this development flow is based on the SUIF framework [23] developed at Stanford. It aims to generate an internal representation of the program from which other modules can operate. Moreover, this module has to extract the loop kernels inside the C code and transmit them to the module (gDART) in charge of transforming the regular portions of code into HW configurations. To increase the parallelism of each loop kernel, some specific algorithms have been developed inside the SUIF front end to unroll the loops according to the number of functional units available in the cluster. Finally, in order to increase the temporal locality of the data, other loop transformations have also been

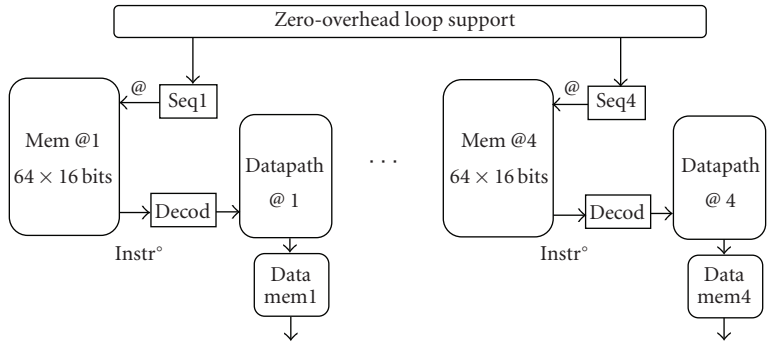


Figure 5: Address generation units with zero-overhead loop support.

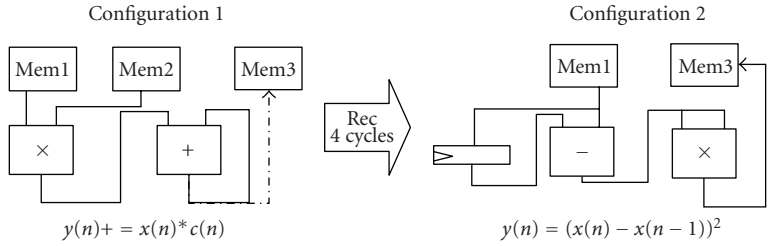


Figure 6: Hardware reconfiguration example.

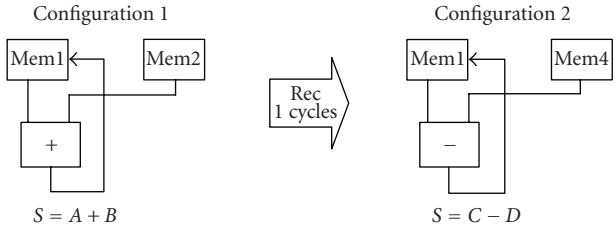


Figure 7: Software reconfiguration example.

developed to decrease the amount of data memory accesses and hence the energy consumption [24, 25].

6.2. cDART compiler

In order to generate the software reconfiguration instructions, we have integrated a compiler, cDART, into our development flow. This tool was generated—thanks to the CALIFE tool suite which is a retargetable compiler framework based on the ARMOR language, developed at INRIA [26]. DART was first described in the ARMOR language.This implementation description arises from the inherent needs of the three main compiling activities which are the code selection, the allocation, and the scheduling, and from the architectural mechanisms used by DART. It has to be noticed that the software reconfigurations imply some limitations about the RDPs flexibility, and hence the architecture subset concerned with this reconfiguration is very simple and orthogonal. It is made up of four independent functional units working on four memories in a very flexible manner; that is, there are no limitations on the use of the instruction parallelism.

The next step in generating cDART was to translate the DART ARMOR description into a set of rules able to analyze expression trees in the source code, thanks to the ARMORC tool. Finally, to build the compiler, the CALIFE framework allowed us to choose the different compilation passes (e.g., code selection, resource allocation, scheduling, etc.) that had to be implemented in cDART. In CALIFE, while the global compiler structure is defined by the user, module adaptations are automatically performed by the framework. Within CALIFE, the efficiency of each compiler structure can easily be checked and new compilation passes can quickly be added or subtracted from the global compiler structure. Thanks to CALIFE framework, we have designed a compiler which automatically generates the software configurations for DART.

6.3. gDART synthesizer

If the software reconfiguration instructions can be obtained—thanks to classical compilation schemes—the hardware reconfiguration instructions have to be generated according to more specific synthesis tasks. In fact, as mentioned previously, hardware reconfiguration can be specified by a set of instructions that exhibits the RDP structure. Hence, the developed tool (gDART) has to generate a datapath configuration in adequacy with the processing of the loop kernel represented by a dataflow graph (DFG). Since the parallelism has been exhibited during the SUIF transformations, the only task that must be done by gDART is to find the datapath structure allowing for the DFG implementation and to translate it into an HW configuration.

Due to the RDP structure, the main constraint on the efficient scheduling of the DFG is to compute the critical loops

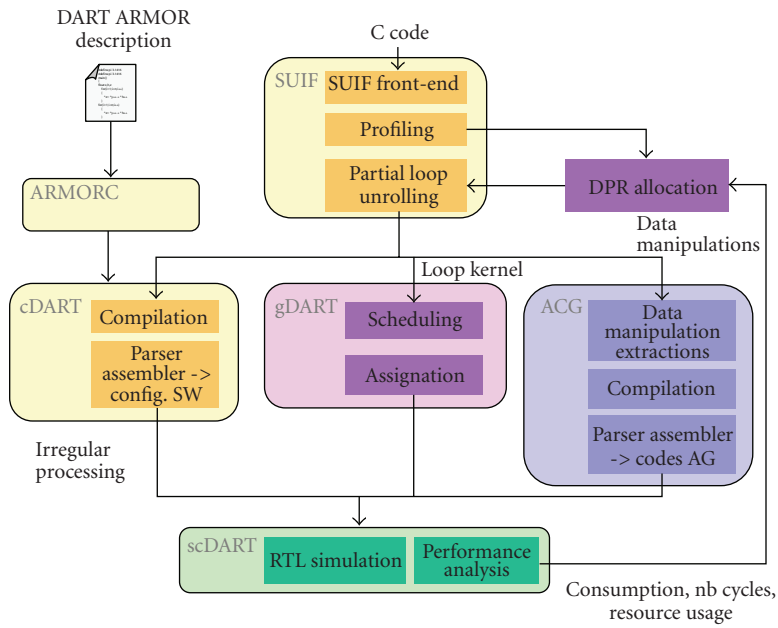


Figure 8: DART development flow.

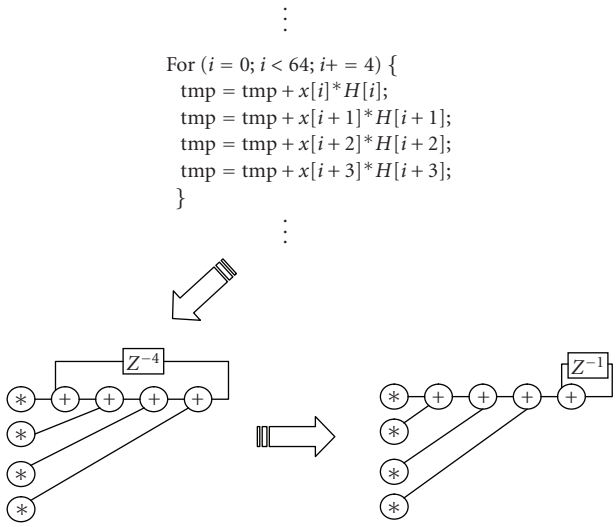


Figure 9: Critical loop reduction.

of the DFG in a single cycle. Otherwise, if data are shared over several clock cycles, local memories have to be used, and that decreases energy efficiency. To give more flexibility in this regard, registers were added to the RDP datapath (see reg1 and reg2 in Figure 2). This problem can be illustrated by the example of the finite impulse response (FIR) filter dataflow graph represented in Figure 9 which mainly concerns the accumulations. In this particular case, the solution is to transform the graph in order to reduce the critical loop timing to only one cycle by swapping the additions. This solution can be generalized by swapping the operations of a critical loop according to the associativity and distributivity rules associated with the operators.

The DFG has next to be optimized to reduce the pipeline latency according to classical tree height reduction techniques. Finally, calculations have to be assigned to operators and data accesses to memory reading or writing. These accesses are managed by the address generators.

6.4. Address code generator

If gDART and cDART allow for the definition of the datapath, they do not take into consideration the data access. Hence, a third tool, address code generator (ACG), has been developed in order to obtain the address generation instructions which will be executed on the address generators of each RDP. Since the address generators architectures are similar to tiny RISCs (see Section 4.4), the generation of these instructions can be done by classical compilation steps—thanks to CALIFE. The input of the compiler is this time a subset of the initial input code which corresponds to data manipulations, and the compiler is parameterized by the ARMOR description of the address generation unit.

6.5. scDART simulator

The different configurations of DART can be validated—thanks to a bit-true and cycle-true simulator (scDART), developed in SystemC. This simulator also generates some information about the performance and the energy consumption of the implemented application. In order to have a good relative accuracy, the DART modeling has been done at the register-transfer level and each operator has been characterized by an average energy consumption per access—thanks to gate-level estimations realized with *Design Power* from Synopsys (Calif, USA).

7. WIRELESS BASE STATION

In this section, we focus on the implementation of a wireless base station application as a proof of concept. The base station is based on wideband code division multiple access (WCDMA) which is a radio technology used in third-generation (3G) mobile communication systems.

When a mobile device needs to send data to the base station, a radio access link is set up with a dedicated channel providing a specific bandwidth. All data sent within a channel have to be coded with a specific code to distinguish the data transmitted in that channel from the other channels. The number of codes is limited and depends on the total capacitance of the cell, which is the area covered by a single base station. To be compliant with the radio interface specification (universal terrestrial radio access (UTRA)), each channel must achieve a data rate of at least 128 kbps. The theoretical total number of concurrent channels is 128. As in practice, only about 60% of the channels are used for user data; the WCDMA base station can support 76 users per carrier.

In this section, we present and compare the implementation of a 3G WCDMA base-station receiver on DART, on an Xilinx XC200E VIRTEX II Pro FPGA and on the Texas Instrument C62x DSP. Energy distribution between different components of the DART architecture is also discussed. The figures presented in this section were extracted from logical synthesis on 0.18 μm CMOS technology with 1.9 V power supply, and from the cycle-accurate bit-accurate simulator of the architecture scDART. Running at a frequency of 130 MHz, a DART cluster is able to provide up to 6240 MOPS on 8-bit data.

7.1. WCDMA base-station receiver

WCDMA is considered as one of the most critical applications of third-generation telecommunication systems. Its principle is to adapt signals to the communication support by spreading its spectrum and sharing the communication support between several users by scrambling communications [27]. This is done by multiplying the information by private codes dedicated to users. Since these codes have good autocorrelation and intercorrelation properties [28], there is virtually no interference between users, and consequently they may be multiplexed on the same carrier frequency.

Within a WCDMA receiver, real and imaginary parts of data received on the antenna, after demodulation and digital-to-analog conversion, are first filtered by two real FIR *shaping filters*. These two 64-tap filters operate at a high frequency (15.36 MHz), which leads to a high complexity of 3.9 GOPS (giga operations per second). Next, a *rake receiver* has to extract the usable information in the filtered samples and retrieve the transmitted symbol. Since the transmitted signal reflects on obstacles like buildings or trees, the receiver gets several replicas of the same signal with different delays and phases. By combining the different paths, the decision quality is greatly improved, and consequently a rake receiver is constituted of several *fingers* which have to despread one part of the signal, corresponding to one path of the transmitted information. This task is realized at a chip rate of 3.84 MHz.

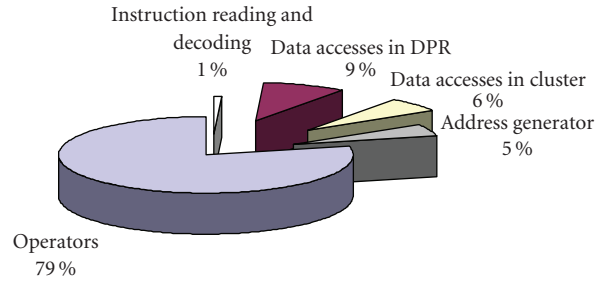


Figure 10: Power repartition in DART for the WCDMA receiver.

The decision is finally made on the combination of all these despread paths. The complexity of the despreading is about 30 MOPS for each finger. Classical implementations use 6 fingers per user. For all the preceding operations, we use 8-bit data with a double precision arithmetic during accumulations, which allows for subword processing.

A base station keeps the transactions of multiple users (approximately 76 per carrier), so each of the above-mentioned algorithms has to be processed for each of the users in the cell.

7.2. Implementation results and energy distribution

The effective computation power offered by a DART cluster is about 6.2 GOPS on 8-bit data. This performance level comes out of the flexibility of the DART interconnection network which allows for an efficient usage of the RDP internal processing resources.

Dynamic reconfiguration has been implemented on DART, by alternating different tasks issued from the WCDMA receiver application (shaping FIR filtering, complex despreading implemented by the rake receiver, chip-rate synchronization, symbol-rate synchronization, and channel estimation). Between two consecutive tasks, a reconfiguration phase takes place. Thanks to the configuration data volume minimization on DART, the reconfiguration overhead is negligible (3 to 9 clock cycles). These phases consume only 0.05% of the overall execution time.

The power needed to implement the complete WCDMA receiver has been estimated to about 115 mW. If we consider the computational power of each task, the average energy efficiency of DART is 38.8 MOPS/mW. Figure 10 represents the distribution of power consumption between various components of the architecture. It is important to notice that the main source of power consumption is that of the operators (79%). Thanks to the configuration data volume minimization and the reconfiguration frequency reduction, the energy wastes associated with the control of the architecture are negligible. During this processing, only 0.9 mW is consumed to read and decode control data; that is, the flexibility cost is less than 0.8% of the overall consumption needed for the processing of a WCDMA receiver.

The minimization of local memory access energy cost, obtained by the use of a memory hierarchy, allows for the consumption due to data accesses (20%) to be kept under control. At the same time, connections of *one-towards-all*

type allow for a significant reduction in the amount of data memory accesses. In particular, on filtering or complex despreading applications, broadcast connections allow for the reduction by a factor of six the amount of data memory accesses. The use of delay chains allows for the exploitation of data temporal locality and skipping a number of data memory accesses.

In order to compare our results to widely accepted architectures, we consider in the rest of the section the FIR filter and the rake receiver. In the case of the FIR filter, 63% of the DART cluster are used, and the energy efficiency reaches 40 MOPS/mW. Arithmetic operators represent more than 80% of the total energy, thus minimizing the energy wastes. For the rake receiver, the use of SWP operators permits the real-time implementation of 206 fingers per cluster, that is, up to 33 users per cluster. Since this algorithm uses more intensive memory accesses, the energy efficiency is reduced to 31 MOPS/mW.

According to the traditional tradeoff of FPGA designs, we need to choose particular circuits within the whole range of available chips. Two scenarios can be addressed: one FPGA implements the complete application or one smaller FPGA implements each task of the receiver and is reconfigured between two consecutive tasks. The first solution optimizes reconfiguration overhead but comes with a high power consumption. The second approach reduces performances but will also decrease the power consumption.

A shaping filter on an Xilinx VIRTEX II architecture can support 128 channels with a length of 27 symbols, 8 samples by symbols [21]. For the rake receiver, the VIRTEX II supports 512 complex tracking fingers (7 bits on I and Q). This receiver requires 3500 slices. For a design running at 128 MHz, the FPGA solution can support up to 64 channels with a 2 Mbps frequency sampling. Using an XC2V1000 FPGA at 1.5 V, the consumed power is almost 2 W, and the energy efficiency is 7.7 MOPS/mW.

We have implemented the same design in a smaller XC2V250 FPGA and used reconfiguration between the tasks. It results in an estimated power consumption of 570 mW for the filter and 470 mW for the rake receiver. The energy efficiencies are then 6.8 and 0.4 MOPS/mW for the two tasks, respectively. These results do not take the reconfigurations into account.

According to the real-time constraints of the application, the FIR filter cannot be implemented on a DSP processor. The TMS320C62 VLIW DSP running at 100 MHz can support a 6-finger rake receiver for a bandwidth of 16 KBps per channel [29]. This solution supports UMTS requirements, but for multiple users, it is necessary to implement one DSP per user. Consuming 600 mW, its energy efficiency is 0.3 MOPS/mW.

The same design has been implemented and optimized into the TMS320C64 VLIW DSP [30]. This processor is a high-performance DSP from Texas Instruments that can run at a clock frequency up to 720 MHz and consumes 1.36 W. It includes 8 independent operators and can reach a peak performance of 5760 MIPS. The C64x DSP provides SWP capabilities which increase performance for 8-bit data. The energy efficiency is 0.15 MOPS/mW for the rake receiver for one user, but this architecture can support up to 5 users per chip.

The XPP reconfigurable processor has a structure which is close to the concepts of DART, but without exploiting memory and interconnection hierarchies. For 0.13 μm technology at 1.5 V, it can run at 200 MHz. The use of 48 PAEs processing 8-bit data in SWP mode consumes 600 mW and enables the implementation of 400 rake fingers at the chip rate of 3.84 MHz [31]. While achieving twice the pick performance of DART, its energy efficiency is 50% less and achieves 20 MOPS/mW.

8. VLSI INTEGRATION AND FIGURES

The VLSI integration of DART has been made in a collaborative project dedicated to a 4G telecommunication application platform in the context of the 4-More European project. The fresh architecture is an NoC-based system-on-chip for application prototyping designed by the CEA-LETI [30]. This architecture contains 23 IPs connected to a 20-node network (called Faust) [32] for a total complexity of 8-million gates (including RAMs). The circuit has been realized using 0.13 μm CMOS technology from STMicroelectronics. IPs from different partners were implemented:

(i) an ARM946ES core which is included in an AMBA bus subsystem;
(ii) two intensive data processing blocks, a turbo encoder from France Telecom R&D, and a convolutional decoder (Viterbi) from Mitsubishi/ITE;
(iii) IPs for OFDM communications designed by the CEA-LETI;
(iv) a reconfigurable controller designed by the CEA LIST;
(v) a DART cluster designed by IRISA and implemented by CEA LIST.

Figure 11 shows the die photo of the fresh chip and the floorplan of the different IP blocks.

In this circuit, DART is intended to implement the channel estimation of the OFDM application. To achieve this goal, we have integrated two division operators into the *application-specific area* of the cluster. Memory hierarchy has been modified to respond to the communication paradigm of the used network. It integrates two FIFOs serving as a network interface. All local memories are dual-port RAMs to enable read/write accesses in the same cycle, while facilitating the design by avoiding multiphase clocks. Due to accuracy concerns, the cluster was modified to support 32 bits in the datapath and in the local memories. These specific modifications had a significant impact on the power figures, but they had increased the quality of the processing and simplified the chip design.

Including all the above modifications, the resulting DART circuit presents a complexity of 2.4 M gates including the whole memory hierarchy. Synthesis, place and route, design check, and validation process have taken 108 hours. The cluster requires a total power of 709 mW, including a 100 mW leakage power. DART can run at a 200 MHz clock frequency, and then it reaches 4800 MOPS for 32-bit operations and up to 9600 MOPS for 16-bit operations. These first

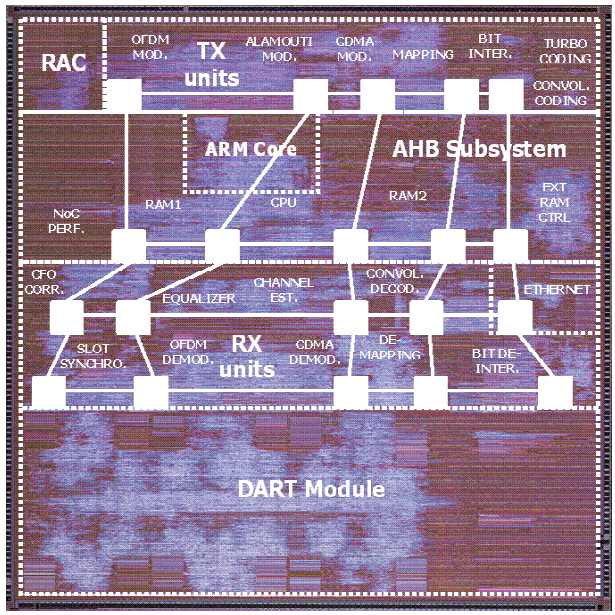


Figure 11: The fresh circuit die photo including the DART IP.

Table 2: Characteristics of the DART IP.

Technology	0.13 μm CMOS from STMicrolectronics
Supply voltage	1.2 V
Die size	2.68 mm $\times$ 8.3 mm
Clock frequency	200 MHz
Average total power	709 mW
Transistor count	2.4-million transistors
Computing performances	4800 MOPS on 32-bit data
	9600 MOPS on 16-bit data

results confirm the energy efficiency of the proposed architecture.

9. CONCLUSIONS

In this paper, we have shown that functional-level reconfiguration offers opportunities to improve energy efficiency of flexible architectures. This level of reconfiguration allows for the reduction of energy wastes in control by minimizing reconfiguration data volume. The coarse-grain paradigm of computation optimizes storage as well as computation resources from an energy point of view. The association of key concepts and energy-aware design has led us to the definition of the DART architecture. This architecture deals concurrently with high-performance, flexibility, and low-energy constraints. We have validated this architecture by presenting implementation results coming from a WCDMA receiver and demonstrated its potential in the context of multimedia mobile computing applications. Finally, a chip was designed and fabricated including a DART cluster to validate the concept.

REFERENCES

[1] J. Fridman, "Sub-word parallelism in digital signal processing," *IEEE Signal Processing Magazine*, vol. 17, no. 2, pp. 27–35, 2000.

[2] J. P. Wittenburg, P. Pirsch, and G. Meyer, "A multithreaded architecture approach to parallel DSPs for highperformance image processing applications," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS '99)*, pp. 241–250, Taipei, Taiwan, October 1999.

[3] 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.

[4] I. Brynjolfson and Z. Zilic, "FPGA clock management for low power applications," in *Proceedings of the 8th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '00)*, pp. 219–225, Monterey, Calif, USA, February 2000.

[5] H. Zhang, M. Wan, V. George, and J. Rabaey, "Interconnect architecture exploration for low-energy reconfigurablesingle-chip DSPs," in *Proceedings of the IEEE Computer Society Workshop On VLSI (VLSI '99)*, pp. 2–8, Orlando, Fla, USA, April 1999.

[6] J. Villarreal, D. Suresh, G. Stitt, F. Vahid, and W. Najjar, "Improving software performance with configurable logic," *Design Automation for Embedded Systems*, vol. 7, no. 4, pp. 325–339, 2002.

[7] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Using the kressarray for reconfigurable computing," in *Configurable Computing: Technology and Applications*, vol. 3526 of *Proceedings of SPIE*, pp. 150–161, Bellingham, Wash, USA, November 1998.

[8] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD—reconfigurable pipelined datapath," in *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL '96)*, vol. 1142 of *Lecture Notes in Computer Science*, pp. 126–135, Darmstadt, Germany, September 1996.

[9] E. Waingold, M. Taylor, D. Srikrishna, et al., "Baring it all to software: raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[10] B. Salefski and L. Caglar, "Re-configurable computing in wireless," in *Proceedings of the 38th Conference on Design Automation (DAC '01)*, pp. 178–183, Las Vegas, Nev, USA, June 2001.

[11] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—a self-reconfigurable data processing architecture," *The Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[12] M. Suzuki, Y. Hasegawa, Y. Yamada, et al., "Stream applications on the dynamically reconfigurable processor," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 137–144, Brisbane, Australia, December 2004.

[13] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1876–1886, 2003.

[14] H. Zhang, V. Prabhu, V. George, et al., "A 1-V heterogeneous reconfigurable DSP IC for wireless baseband digital signal processing," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1697–1704, 2000.

[15] V. George, *Low energy field-programmable gate array*, Ph.D. thesis, University of California, Berkeley, San Diego, USA, 2000.

[16] R. David, D. Chillet, S. Pillement, and O. Sentieys, "DART: a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02)*, pp. 156–163, Fort Lauderdale, Fla, USA, April 2002.

[17] H. Meier, *Analysis and design of low power digital multipliers*, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, Pa, USA, August 1999.

[18] M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, San Francisco, Calif, USA, 2004.

[19] R. David, D. Chillet, S. Pillement, and O. Sentieys, "A compilation framework for a dynamically reconfigurable architecture," in *Proceedings of 12th International Conference on the Reconfigurable Computing is Going Mainstream, Field-Programmable Logic and Applications*, vol. 2438 of *Lecture Notes in Computer Science*, pp. 1058–1067, Springer, Montpellier, France, September 2002.

[20] R. Schreiber, S. Aditya, S. Mahlke, et al., "PICO-NPA: high-level synthesis of non programmable hardware accelerators," Tech. Rep. HPL-2001-249, Hewlett-Packard Labortories, Palo Alto, Calif, USA, 2001.

[21] D. Nicklin, "Wireless base-station signal processing with a platform FPGA," in *Proceedings of the Wireless Design Conference*, London, UK, May 2002.

[22] J. R. Hauser, *Augmenting a microprocessor with reconfigurable hardware*, Ph.D. thesis, University of California, Berkeley, San Diego, USA, 2000.

[23] R. P. Wilson, R. S. French, C. S. Wilson, et al., "SUIF: an infrastructure for research on parallelizing and optimizing compilers," Tech. Rep., Computer Systems Laboratory, Stanford University, Stanford, Calif, USA, May 1994.

[24] A. Fraboulet, K. Kodary, and A. Mignotte, "Loop fusion for memory space optimization," in *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS '01)*, pp. 95–100, Montreal, Canada, September-October 2001.

[25] A. Fraboulet, G. Huard, and A. Mignotte, "Loop alignment for memory accesses optimization," in *Proceedings of the 12th International Symposium on System Synthesis (ISSS '99)*, p. 71, San Jose, Calif, USA, November 1999.

[26] F. Charot and V. Messe, "A flexible code generation framework for the design of application specific programmable processors," in *Proceedings of the 17th International Workshop on Hardware/Software Codesign (CODES '99)*, pp. 27–31, Rome, Italy, May 1999.

[27] T. Ojanpera and R. Prasad, *Wideband CDMA for Third Generation Mobile Communications*, Artech House, Norwood, Mass, USA, 1998.

[28] E. H. Dinan and B. Jabbari, "Spreading codes for direct sequence CDMA and wideband CDMA cellular networks," *IEEE Communications Magazine*, vol. 36, no. 9, pp. 48–54, 1998.

[29] Texas instruments, "*TMS320C64x technical overview*," Texas instruments, February 2000.

[30] Y. Durand, C. Bernard, and D. Lattard, "FAUST: on-chip distributed architecture for a 4G baseband modem SoC," in *Proceedings of Design and Reuse (IP-SOC '05)*, pp. 51–55, Grenoble, France, December 2005.

[31] PACT, "The XPP white paper : a technical perspective," Release 2.1, PACT, March 2002.

[32] E. Beigné, 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 (ASYNC '05)*, pp. 54–63, New York, NY, USA, March 2005.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 369040, 10 pages
doi:10.1155/2008/369040

Research Article

Exploiting Process Locality of Reference in RTL Simulation Acceleration

Aric D. Blumer and Cameron D. Patterson

Virgina Polytechnic Institute and State University, Blacksburg, VA 24061, USA

Correspondence should be addressed to Aric D. Blumer, aric@vt.edu

Received 1 June 2007; Revised 5 October 2007; Accepted 4 December 2007

Recommended by Toomas Plaks

With the increased size and complexity of digital designs, the time required to simulate them has also increased. Traditional simulation accelerators utilize FPGAs in a static configuration, but this paper presents an analysis of six register transfer level (RTL) code bases showing that only a subset of the simulation processes is executing at any given time, a quality called *executive locality of reference*. The efficiency of acceleration hardware can be improved when it is used as a process cache. Run-time adaptations are made to ensure that acceleration resources are not wasted on idle processes, and these adaptations may be affected through process migration between software and hardware. An implementation of an embedded, FPGA-based migration system is described, and empirical data are obtained for use in mathematical and algorithmic modeling of more complex acceleration systems.

1. INTRODUCTION

The capacity of integrated circuits has increased significantly in the last decades, following a trend known as "Moore's Law" [1]. The code bases describing these circuits have likewise become larger, and design engineers are stressed by the opposing requirements of short time to market and bug-free silicon. Some industry observers estimate that every 18 months the complexity of digital designs increases by at least a factor of ten [2], with verification time taking at least 70% of the design cycle [3]. Compounding the problem are the large, nonrecurring expenses of producing or correcting an integrated circuit, which makes the goal of first-time working silicon paramount. Critical to satisfying both time to market and design quality is the speed at which circuits can be simulated, but simulation speed is "impractical" or at best "inadequate" [4, 5].

As in the software industry, the efforts to mitigate the effect of growing hardware complexity have resulted in the use of greater levels of abstraction in integrated circuit descriptions. The most common level of abstraction used for circuit descriptions is the register transfer level (RTL). RTL code represents the behavior of an integrated circuit through the use of high-level expressions and assignments that infer logic gates and flip-flops in a fabricated chip. While the synthesis of descriptions above RTL is improving, it is not yet as efficient as human-coded RTL [6, 7], so RTL still forms a necessary part of current design flows.

While simulating circuit descriptions at higher levels of abstraction is faster than RTL simulation, another method of improving verification times is the hardware acceleration of RTL code. The traditional approach to hardware-based simulation acceleration is to use an array of field programmable gate arrays (FPGAs) to emulate the device under test [8]. The RTL code of the device is mapped to the FPGAs causing a significant increase in execution speed. There are, however, two drawbacks to this approach. First, the RTL code must be mapped to the hardware before the simulation can begin. Second, once the mapping is complete, it is static. Because it is static, the speedup is directly related to the amount of the design that fits into the acceleration hardware. For example, if only one half of the design can be accelerated, Amdahl's law states that the maximum theoretical speedup is two. Enough hardware must be purchased then to contain most if not all of the device under test, but hardware acceleration systems are expensive [8].

FPGA smay be configured many times with different functionality, but they are designed to be statically configured.

That is, the power-up or reset configuration of the FPGA can be changed, but the configuration remains static while the device is running. This is the reason that existing accelerators statically map the design to the FPGA array. It is possible, however, to change the configuration of an FPGA at run time, and this process is aptly termed *run-time reconfiguration* (RTR). A run-time reconfigurable logic array behaves as a sandbox in which one can construct and tear down structures as needed. By instantiating and removing processors in the array during run time, users can establish a dynamically adaptable parallel execution system. The system can be tuned by migrating busy processes into and idle processes out of the FPGA and establishing communication routes between processes on demand. The processors themselves can be reconfigured to support a minimum set of operations required by a process, thus reducing area usage in the FPGA. Area can also be reduced by instantiating a shared functional unit that processes migrate to and from when needed. Furthermore, certain time-consuming or often-executed processes can be synthesized directly to hardware for further speed and area efficiency.

This paper presents the research done in the acceleration of RTL simulations using process migration between hardware and software. Section 2 gives an overview of existing acceleration methods and systems. Section 3 develops the properties that RTL code must exhibit in order for run-time mapping to acceleration hardware to be beneficial. Section 4 presents the results of profiling six sets of RTL code to determine if they have exploitable properties. Section 5 describes an implementation of an embedded simulator that utilizes process migration. Section 6 develops an algorithmic model based on empirical measurements of the implementation of Section 5. Section 7 concludes the paper.

2. EXISTING ACCELERATION TECHNOLOGY

Much progress has been made in the area of hardware acceleration with several commercial offerings in existence. Cadence Design Systems (Calif, USA) offers the Palladium emulators [9] and the Xtreme Server [10], both being parts of verification systems which can provide hardware acceleration of designs as well as emulation. Palladium was acquired with Quickturn Design Systems, Inc., and the Xtreme Server was acquired with Verisity, Inc. While information about their technology is sketchy, they do make it clear in their marketing "datasheets" that the HDL code is partitioned into software and hardware at compile time. State can be swapped from hardware to software for debugging, but there is apparently no run-time allocation of hardware resources occurring.

One publication by Quickturn [11], while not specifically mentioning Palladium, describes a system to accelerate event-driven simulations by providing low-overhead communication between the behavioral portions and the synthesized portions of the design. The system targets synthesizable HDL code to FPGAs, and then partitions the result across an array of FPGAs. The behavioral code is compiled for a local microprocessor run alongside the FPGA array. Additionally, the system synthesizes logic into the FPGAs to detect trigger conditions for the simulator's event loop, thus off-loading event detection from the software. The mapping to the hardware is still static.

EVE Corporation produces simulation accelerators in their ZeBu ("zero bugs") product line [12]. EVE differentiates itself somewhat from its competitors by offering smaller emulation systems such as the ZeBu-UF that plugs into a PCI slot of a personal computer, but their flow is similar, requiring a static synthesis of the design before simulation.

Another approach to hardware acceleration is to assemble an array of simple processing elements in an FPGA, and to schedule very long instruction words (VLIWs) to execute the simulation as in the SimPLE system [13]. Each processing element (PE) is a two-input lookup table (LUT) with a two-level memory system. The first level memory is a register file dedicated to each PE with a second-level spill-over memory shared among a group of PEs. The PEs are compiled into an FPGA which is controlled by a personal computer host through a PCI bus. The host schedules VLIWs to the FPGA, and each instruction word contains opcodes and addresses for all the PEs. On every VLIW execution, a single logic gate of the simulation is evaluated on each PE. The design flow is to synthesize the HDL design into a Verilog netlist, translate the netlist into VLIW instructions using their SimPLE compiler, and schedule them into the FPGA to execute the simulation, resulting in a gate-level simulation rather than RTL-level. An EDA startup company, Liga Systems, Inc., (Calif, USA) has been founded to take advantage of this technology [14].

These systems all suffer from a common shortcoming, albeit to varying degrees. The design must be compiled to a gate-level netlist or to FPGA structures before it can be simulated. In some cases, the mapping is to a higher abstraction level, thus shortening the synthesis time, but some prior mapping is required. Furthermore, all the systems except the SimPLE compiler map to the hardware resources statically. That is, once the mapping to hardware is established, the logic remains there throughout the life of the simulation. If the hardware resources are significant, then this is an acceptable solution, but if a limited amount of hardware needs to be used efficiently, this can be problematic. The next section presents a property of RTL code that can be exploited to reduce the amount of hardware that is required.

3. EXECUTIVE LOCALITY OF REFERENCE

An RTL simulation consists of a group of processes that mimic the behavior of parallel circuitry. Typical RTL simulation methods view processes as a sequence of statements that calculate the process outputs from the process inputs when a trigger occurs. The inputs are generally expressed as signal values in the right-hand side of expressions or in branching conditionals. The trigger, which does not necessarily contain the inputs, is expressed as a *sensitivity list*, which in synchronous designs is often the rising edge of a clock, but it may be a list of any number of *events*. These events determine when a process is to be run, and thus the simulation method is called *event-driven simulation* [15]. When all the events at the current simulation time are serviced, the simulator advances to the time of the next

event. Every advance marks the end of a *simulation cycle* (abbreviated as "simcycle"). In synchronous designs, the outputs of synchronous process can only change on a clock edge, so *cycle-based simulation* can improve simulation times by only executing processes on clock edges [16].

The two most common languages used for RTL coding are Verilog and VHDL, and in both cases, during a simulation cycle, the processes do not communicate while they are executing. Rather, the simulator propagates outputs to inputs between process executions [17, 18]. Furthermore, the language standards specify that the active processes may be executed in any order [17, 18], including simultaneous. It is, Through this simultaneous execution of processes, hardware accelerators provide a simulation speedup. Traditional hardware accelerators place both idle and active processes in the hardware with a static mapping, and the use of the accelerator is less efficient as a result.

A simulator that seeks to map only active processes to an accelerator at run time relies fundamentally on the ability to delineate between active and idle processes. Locality of reference is a term used to describe properties of data that can be exploited by caches [19]. Temporal locality indicates that an accessed data item will be accessed again in the near future. Spatial locality indicates that items near an accessed item will likely be accessed as well. These properties are exploited by keeping often-accessed data items in a cache along with their nearest neighbors. For executing processes, there is *executive locality of reference*. Temporal executive locality is the property that an executing process will be executed again in the near future. Spatial executive locality is the property that processes receiving input from an executing process will also be executed. These properties can be exploited by keeping active processes in a parallel execution cache.

Spatial executive locality is easily determined in RTL code using the "fanout" or dependency list of a process's output, a structure a simulator already maintains. Temporal executive locality, however, is not quite so obvious. Take, for instance, a synchronous design in which every process is triggered by the rising edge of a clock. An unoptimized cycle-based simulator will run every process at every rising edge of the clock, so every process is always active. If, however, no inputs to a process have changed from the previous cycle, then the output of the process will not change. Therefore, it does not need to be run again until the inputs do change. During this period, the process is idle. This is the fundamental characteristic on which temporal locality depends, and detecting changes on the inputs can be accomplished with an *input monitor*.

Input monitoring has been used to accelerate the software execution of gate simulations (a technique often called "clock suppression") [20], but it has not been used to determine how hardware acceleration resources should be used. Whether the inputs change or not is readily determinable by a simulator, but the question remains whether activity of the processes varies enough to be exploitable. Memory caches are inefficient if all the cachable memory is always accessed. Similarly, an execution cache is inefficient if all the processes are always active. Even though the processes exhibit good temporal locality, the effect of process caching is diminished in this case. But are RTL processes always active during a simulation? The next section addresses that question.

```
(1) always   @B   $check_sig(21, 0, B);
(2) always   @C   $check_sig(21, 1, C);
(3) always   @ (posedge  CLK);
(4)        if ($shall_run(21));
(5)             A <= B + C;
```

Algorithm 1: Input monitor instrumentation.

4. PROFILING AND RESULTS

The RTL code of six Verilog designs from OpenCores [21] was instrumented and profiled using Cadence's LDV 5.1. Since the source code for LDV's ncsim is not available, every input of the `always` blocks is monitored with a PLI function call. The previous value of the inputs are kept, and the process is run only if at least one of the inputs changes from the previous cycle. A sequence of code demonstrating the method is shown in Algorithm 1. Lines (1), (2), and (4) were added for this study, but RTL code will require no modifications if this functionality is part of the simulator itself. The arguments to `$check_sig`() are the process number, the signal number for this process, and the signal's value. Each process is assigned a unique number, and multiple instantiations of the same process are also delineated. If there are any changes detected by `$check_sig`() for a process, then the next call of `$shall_run`() returns 1. Otherwise, it returns 0, and the code is not run. With this instrumentation, the test benches of the designs were run to verify correct functionality.

The code that was analyzed included a PCI bridge, Ethernet MAC, memory controller, AC97 controller, ATA disk controller, and a serial peripheral interface (SPI). They represent a range of functionality and code sizes (shown in Figure 1, excluding the test bench). Activity data were recorded on a per-test basis, and the metrics used for evaluation are the activity ratio (AR) of each process and the number of lines of code in each process. The AR, expressed as a percentage, is the ratio of the number of times a process is executed to the number of times it was triggered. In other words, it is the ratio of the number of times `$shall_run`() returns 1 to the number of times it is called. The AR, however, does not show the entire picture. If the inactive processes are short sequences of code, then the bulk of the design is still active. Figure 2 shows a representative example of the activity ratios and the line counts of the PCI controller for a single test. These graphs show that a large number of the processes are inactive for the test, a demonstration of temporal executive locality. The graph of the line counts per process shows that the idle processes do comprise a significant part of the design. Note that this graph's Y-axis is limited to show the detail of the majority of the processes;

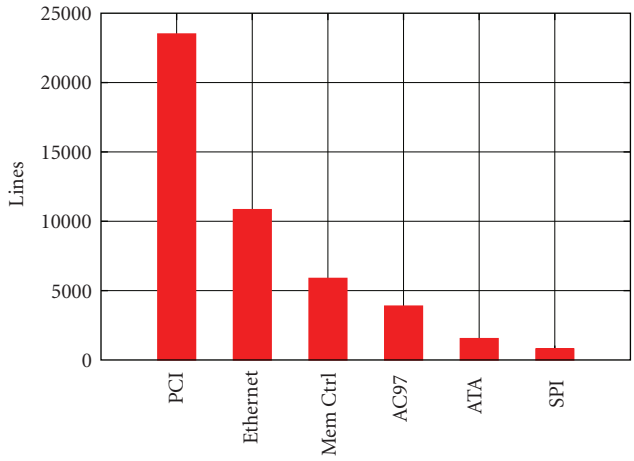


Figure 1: Sizes of profiled code.

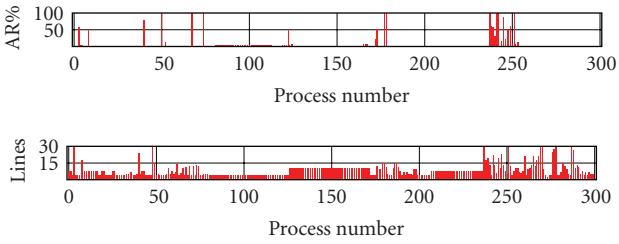


Figure 2: Process activity ratios and line counts for the PCI controller.

several are larger than 30 lines. It is inefficient to place these idle processes into acceleration resources.

The number of lines of HDL code is not a precise indication of the actual run-time size of a process, but it does provide a good estimate. A simulation system would likely use metrics such as compiled code size or actual process run times. Nevertheless, using data such as that shown in Figure 2, it can be determined what percentage of the total lines of code are active during a test. Whether a process is considered active or not is determined by an AR threshold. For example, one can determine how many lines of code there are in all the processes that have an AR above 10%. This is called the *activity footprint*. The smaller the activity footprint, the less hardware is required to accelerate it. The activity footprints for a number of tests are shown in Figure 3 with AR thresholds of 1%, 10%, 20%, and 30%. A system can vary the AR threshold until the activity footprint fits within the acceleration hardware.

The PCI code shows a small activity footprint for the seventeen tests shown. Tests 8 and 9 are PCI target tests, and they show the smallest footprints of 17.9% with a 1% AR threshold. Therefore, for these tests, only enough acceleration hardware is required to hold 17.9% of the process code. Test 16, a bus transaction ordering test, shows the largest footprint, exceeding 50% in all cases. For almost all the tests, the 10% AR threshold yields a footprint less than 50%. The Ethernet code has larger activity footprints in general, but the footprint varies from approximately 29%

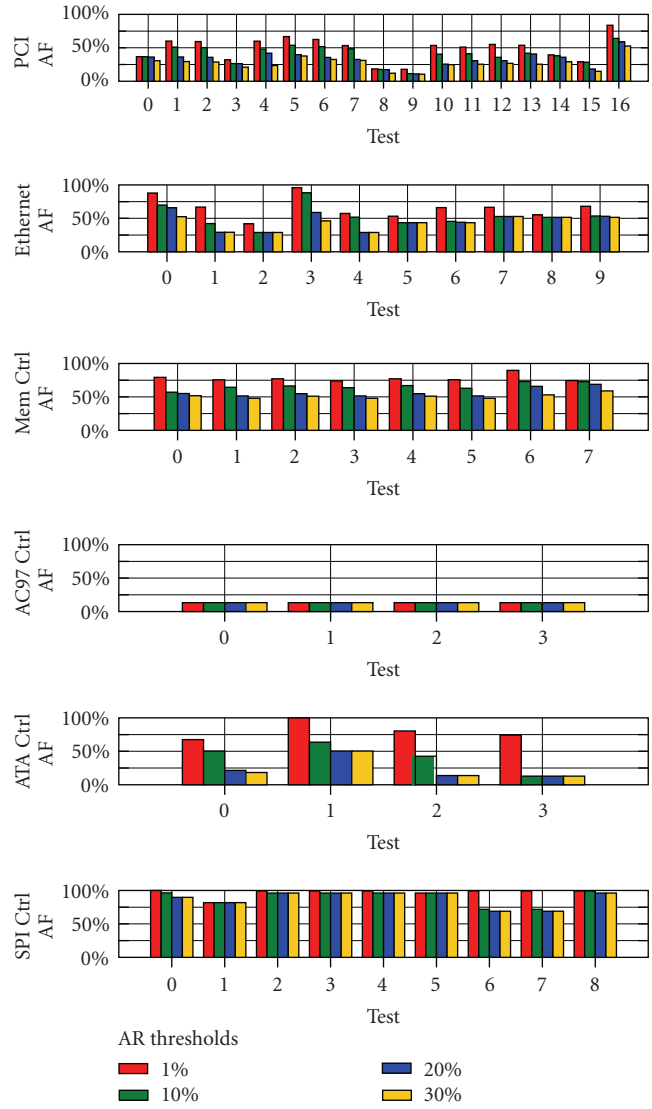


Figure 3: Activity footprints for each design.

for Test 2 (a walking-ones register test) to 88% for Test 3 (a register reset test) when the AR threshold is 10%. The higher values for Test 3 are partly attributable to the fact that the test is relatively short. Even so, the Ethernet code exhibits footprints of approximately 50% or less for a large number of the tests when the AR threshold is 10%.

The behavior of the memory controller is comparable with the Ethernet controller. Its characteristics are largely attributable to a single-state machine process that is 789 lines of code and is always active. In contrast, the AC97 controller has a small activity footprint of just over 13% regardless of the AR threshold. A small percentage of its code handles data flow which the majority of each test exercises.

The ATA and SPI code bases are relatively small with less than 25 processes each. Despite the small code size, they do demonstrate some exploitable locality. In the case of the ATA controller, an AR threshold of 10% gives results that are comparable to the other designs. However, the SPI device,

which is the smallest, has rather large footprints; but these footprints are a percentage of a small code base. It is intuitive that as designs get smaller, they exhibit less exploitable locality because the design is not large enough to have independent code that is idle during a test. These smaller designs would generally not be candidates for hardware acceleration, but acceleration may still prove useful for a large number of small devices such as those found in a system on a chip (SoC) design.

The amount of data gathered from the profiling is significant, and only a portion is shown here. These results, however, show that executive locality of reference does exist in these OpenCores designs, whose code demonstrates various functionality and code size. RTL code is normally proprietary and unavailable to researchers for analysis, but by inference, one would be expected executive locality of reference to be a general rule rather than an exception demonstrated by these particular designs. The remainder of this paper shows how this locality can be exploited.

5. IMPLEMENTING PROCESS MIGRATION

A process migration system consists of communicating processes, processors, and a communication infrastructure. If the processes are to be run in either software or hardware contexts, they must be in a form that is portable. A common instruction set (CIS) serves that purpose. Processes compiled to a CIS can then be executed in a system composed of simple virtual machines (VMs) and real machines (RMs) as shown in Figure 4. A set of processes begins running entirely in VMs, one per process. The use of VMs allows the system to begin execution immediately, and it monitors the activity of the processes to determine which processes should be migrated to the RMs or synthesized directly to the hardware. The algorithms used to determine when and where to migrate processes is left to future work. If there are more active processes that do not fit in the hardware, they can be compiled to native code in the VMs. Idle processes in VMs will not be run and can therefore be left in CIS form until they become active. Also note that the state of any process is available at any time for debugging.

5.1. Implementation details

The infrastructure required to implement a complete simulator would likely require several man-years of effort; but before such an investment should be made in both time and expense, the benefits and drawbacks of such a system must be understood. To this end, the simulation of a simple sort is presented which not only demonstrates the feasibility of a migratory simulator, but also provides the ability to measure system performance for empirical and algorithmic modeling. Modeling allows the impact of further developments to be evaluated.

To this end, a process migration system was implemented on an XUPV2P board developed for the Xilinx University Program [22]. The board is populated with a Virtex-II Pro FPGA (XC2VP30) and 512 megabytes of DDR memory. The XC2VP30 contains two PowerPC 405 processors along

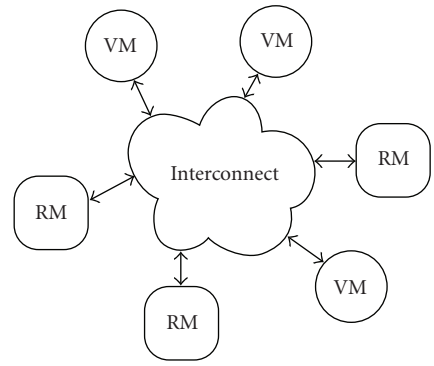


Figure 4: The VMs and RMs execute processes collaboratively. The RMs execute in parallel.

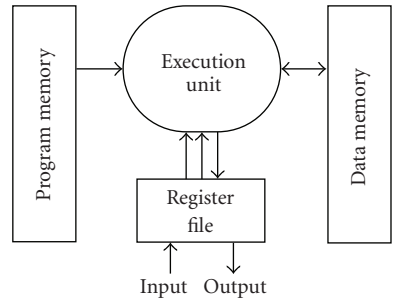


Figure 5: VM and RM block diagram.

with 30,816 logic cells and 136 block RAMs. Using the Xilinx EDK version 7.1i, the design was instantiated with a DDR memory controller, the on-chip peripheral Bus (OPB) infrastructure, the internal configuration access port (ICAP), an ICAP controller, and an array of processors. A VM-based cycle simulator runs the RTL simulation on one PowerPC, and it migrates the state of the VMs to and from the RMs as needed.

The internal structure of both the VMs and RMs is shown in Figure 5. Migration is the transfer of the state in program memory, data memory, and the register file between VMs and RMs using RTR. It is possible in a system with only RMs and VMs to migrate state without RTR, but there are two reasons for using it as follows. (1) The use of only RMs is the first step to a complete migration system that also uses runtime synthesis of processes directly to hardware. It is more efficient to measure migration overheads first with a simpler RM-only system before committing to further work. (2) The additional logic required for migration without RTR would increase the size of the infrastructure, reducing the number of RMs and making timing closure more difficult.

The original RMs were fully pipelined using flip-flops for the register file, but Virtex-IIs do not provide a way to update flip-flop state on a per-RM basis. The `GRESTORE` configuration command updates all flip-flops in the FPGA including critical infrastructure [23]. A solution is to use LUT-distributed RAM instead. LUT-based RAMs can be updated through the ICAP on a per-machine basis, but they cannot provide the single-in-double-out interface required

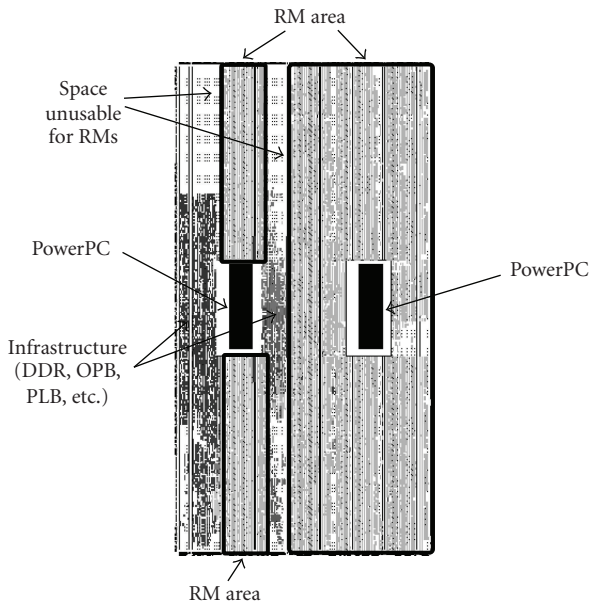


Figure 6: Device floor plan of the migration system.

by full pipelining without doubling the memories. We elected to execute an instruction every two clock cycles in a nonpipelined fashion, and the absence of data hazards makes the RMs smaller.

There is another limitation of LUT RAMs, however, that must be considered. A frame is the minimum configuration unit in an FPGA. In Virtex-II FPGAs, frames span the entire height of the device [23, page 337], and some of the logic used to load the frames is used by LUT RAMs during normal operation. Hence, no LUT RAMs within a frame can be read or written while the frame is being configured. This poses a problem with the infrastructure logic such as the DDR memory controller and the OPB bus interfaces that use LUT RAMs: just reading the frames used by them during run time causes a malfunction. Careful floorplanning is required to ensure that the RMs do not share any frames with this infrastructure. This issue does not affect the RMs themselves, however, because migration to and from RMs only occurs between simulation cycles when the RMs are idle. Each RM is constrained to occupy a 16×16 slice area, and they are floorplanned into columns with other frames reserved entirely for infrastructure. After protective floorplanning, 35 RMs fit with an 82% slice utilization. The resource utilization, including the unusable area due to protective floorplanning, is shown in Figure 6.

5.2. The simulator application

The even-odd transposition (EOT) sort [24] is a good application to measure the performance of the simulator because it allows the number of RMs and the number of simulation cycles to be varied orthogonally while still giving correct results. Each iteration of the EOT sort is a swapping—or transposition—of the numbers to place the greater number on the right. When executed in parallel, it requires a maximum of n cycles to sort n numbers when there are $n/2$ transpositions per cycle. The sort was implemented in VHDL as an array of identical processes, each comparing its own output with its left or right neighbor's output on alternating cycles. In an RTL simulator, this code would be compiled into one process for each instance (barring optimization), each running the same code.

The VHDL was translated into the CIS, which is executable in either VMs or RMs. In this implementation, the CIS is much like typical RISC assembly languages, but the CIS is an abstraction layer that may vary across migration system implementations depending on the possible migration destinations. To make the RMs as small as possible, they have 16-bit instructions and 16-bit data. Each RM also has eight 16-bit registers with some aliased to inputs and outputs.

The PowerPC in the Virtex-II Pro operates at 300 MHz while the remaining logic operates at 100 MHz. The PowerPC executes the VMs and the simulation infrastructure with the "standalone" software package provided in Xilinx's EDK version 7.1i. Inputs and outputs of the RMs are connected to the on-chip peripheral bus (OPB) through a mailbox. Between simulation cycles, the simulator reads the outputs of the RMs and writes their inputs, a process called *software connectivity*. On the other hand, *hardware connectivity* is the joining of RM outputs to RM inputs directly in the FPGA. To measure the difference in overhead between the two types, a number of tests were run with hardware connectivity written into RTL code. A complete system would instead implement these connections at run time, and that behavior is modeled in Section 6.2.

5.3. Results for empirical modeling

The FPGA holds 35 RMs, so the EOT sort was run with 35 processes. Each process is instantiated in a VM connected to its left and right neighbors and is assigned an initial random value. These values are sorted in no more than 35 simulation cycles, but longer runs with correct results were also done for performance measurements.

To evaluate pure parallel performance, the speedups with no migration were measured for both software and hardware connectivity. Figure 7 shows the resulting plots. Because hardware connectivity is currently manual, its performance was measured with 0, 24, 30, 34, and 35 RMs, and Marquardt-Levenberg curve fitting provides the other points. The speedup is the run time without RMs (T_0) divided by the run time with RMs (T_r) [19]. One result of executing code in VMs without native just-in-time (JIT) compilation is a "super-linear" speedup because the RMs execute their processes faster than the VMs. Not shown in the plots due to scaling disparity is the speedup of 1,005 when using hardware connectivity with 35 RMs. To illustrate how this occurs, consider a serial program that executes 10 tasks, each taking 1 second. If those tasks are parallelized on 10 processors, they all execute in 1 second for a speedup of 10. If, however, the parallel processors are ten times faster than the serial processor, the tasks complete in 0.1 second for a speedup of 100. Note also that the relatively low speedups on the left side of the plots demonstrate

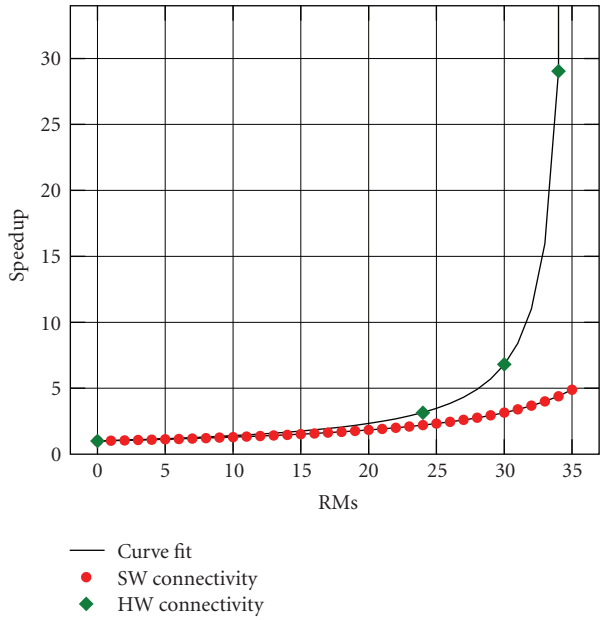


Figure 7: Speedup of HW and SW connectivity (35 simcycles).

Table 1: Modeling parameters.

Param.	Description	Value
C	Simcycles	*Varies*
P	Processes	*Varies*
r	Number of RMs	*Varies*
M	Number of migrations	*Varies*
t_v	VM execution time	83.13 $\frac{\mu s}{\text{simcycle}}$
t_r	RM execution time	17.03 $\frac{\mu s}{\text{simcycle}}$
t_o	Simulator overhead	2.96 $\frac{\mu s}{\text{simcycle}}$
t_m	Migration overhead	6.02 $\frac{\text{ms}}{\text{migration}}$

Amdahl's law, a behavior exhibited by all parallel systems, and is not unique to this system. Figure 7 shows clearly that hardware connectivity is a necessary improvement. These direct connections could be accomplished with a reconfigurable crossbar switch, network on a chip, or run-time routing.

The time required to execute the sorting simulation with software connectivity is

$$T(r) = C(t_o + rt_r + [P - r]t_v) + rMt_m, \quad (1)$$

where t_o, t_r, and so forth are described in Table 1. The sort tests use $C = P = 35$, while r varies from 0 to 35; and t_r includes any RM-related overhead (including the software connectivity accesses) plus any time spent waiting for the RMs to complete. The CIS code consists of 24 instructions, but not all are executed each simulation cycle due to branching. A pessimistic estimate of 50 instructions per simulation cycle gives an RM execution time of 1 microsecond. Since the RMs are notified to execute a simulation cycle just before the PowerPC executes the VMs, the RMs are expected to be finished before the VMs.

Migration overhead, t_m, is affected by reconfiguration time, and the Virtex-II architecture does not lend itself well to quick reconfiguration. Each frame spans the entire height of the FPGA, thus incurring large overheads for the migration of small amounts of state. Additionally, every read and write of configuration frames requires a "dummy" frame to be read or written, and the ICAP has only an 8-bit interface run at a lower clock rate [25, page 4]. Multiple frames must be read and written to migrate state to a single RM, and the unoptimized migration time per machine was measured to exceed 220 milliseconds. However, the full-height span of frames can be exploited. The RMs were floorplanned into columns of 8, except for the rightmost column which contains 11. For each RM, an RLOC constraint places the program, data, and register file memories into the same frames. This optimization reduces the number of frames containing memories from 65 to 16. Sharing frames among RMs also allows frame caching, which minimizes ICAP accesses by reading and writing a shared frame only once during a series of migrations. These optimizations reduced the average migration time from 220 milliseconds to 6.0 milliseconds, and the maximum time for a single migration—which frame caching does not help—is 18.9 milliseconds.

Speedup is defined as $S = T_0/T(r)$, which can be modeled as follows for hardware connectivity, where T_0 is the measured time without RMs:

$$S = \begin{cases} \dfrac{T_0}{C(t_o + Pt_v)} & \text{for } r = 0, \\ \dfrac{T_0}{C(t_o + t_r + [P - r]t_v) + rMt_m} & \text{for } 0 < r < P, \\ \dfrac{T_0}{Ct_o + rMt_m} & \text{for } r = P. \end{cases} \quad (2)$$

The hardware connectivity execution time is calculated differently than software (1) because processes in RMs with direct connections cause no overhead in the simulator. Equation (2) is piecewise because there are no RM overheads t_r and t_m when zero RMs are in use. For the middle region, there is a single t_r for the single RM that has hardware connectivity on one side and software connectivity on the other. The remaining RMs have only hardware connectivity. For the final case, nothing is run in VMs, so there is only the simulator and migration overhead.

As mentioned previously, the EOT sort gives correct results for runs longer than 35 simulation cycles. Figure 8 shows speedup plots for 35, 1024, and 10240 simulation cycles, including all migration overheads for one migration per RM ($M = 1$), along with plots of (2). These simulation lengths are reasonable based on the test benches of the OpenCores code, which contain simulations ranging from dozens to millions of simulation cycles per test. Larger simcycle-to-migration ratios (SMRs) amortize the migration overhead over a longer period of time, improving the speedup. For small SMRs (e.g., 35 : 1), the migration time exceeds the simulation run time, and the system exhibits a

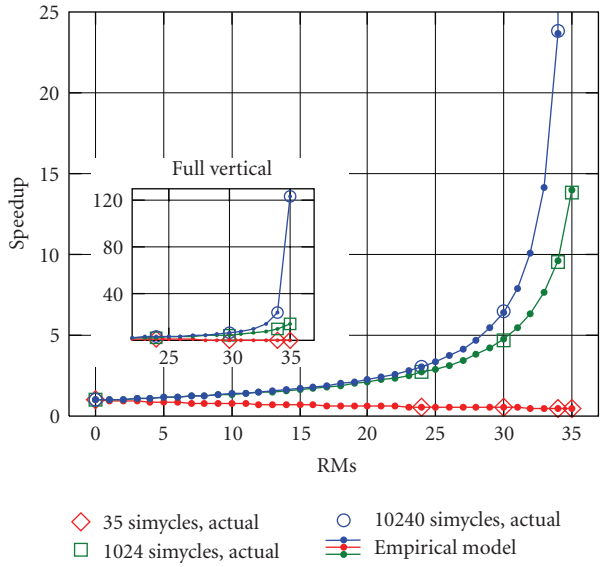


Figure 8: HW connectivity speedups compared to the empirical model.

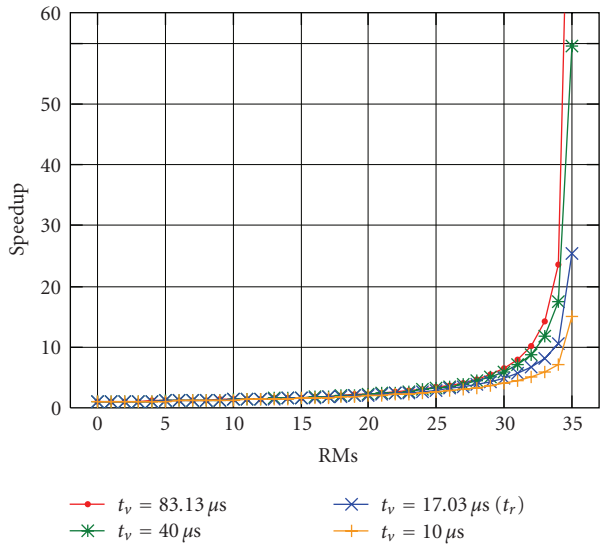


Figure 9: Modeled effect of t_v on speedup (10 k simcycles).

speedup less than one. Using the measured speedups with (2), the Marquardt-Levenberg curve fitting algorithm solves for t_r, t_v, and t_m as shown in Table 1. t_o was calculated directly from the third case of (2) when $M = 0$.

No complete system would run CIS code in VMs without compiling it to native code, so (2) is used to explore the effect of VM-related run times (t_v). Figure 9 shows that speedups of 15 are possible even if VM execution is faster than RM execution ($t_v < t_r$). Since t_v includes the time to execute a process and to update process inputs and outputs, it is unlikely to be less than t_r.

6. ALGORITHMIC MODELING

A mathematical model based upon the empirical data obtained in Section 5 is useful when simplifying assumptions are made about the behavior of the system. Since an implementation that can simulate larger code bases would require a significant investment in infrastructure such as JIT compilers and run-time routers, a more complex model is required to better understand the behavior of complete systems. Some behavior, such as the effect of frame caching, is difficult to model with mathematical formulas. This section explains an algorithmic model based on (2) that is executed as a C program, allowing fixed parameters to be replaced with functions. To ensure that the model reflects the behavior of the system, the graph of Figure 8 was reproduced using the model, and the average percent error for the 1024- and 10240-simcycle plots from the measured speedups is less than 1%. For the 35 simulation cycle case, the average percent error is 2.1%. The subsequent modeling scenarios assume the pessimistic value of $t_v = 16$ microseconds and a migration time of $t_m = 12.5$ milliseconds (the average of 6 milliseconds and 18.9 milliseconds from Table 1).

6.1. Active process ratios

In Section 4, ARs of six designs were measured to demonstrate exploitable executive locality of reference. It is the activity ratio of the processes that identifies the best opportunity for acceleration, and an AR threshold determines which processes are to be migrated to the acceleration hardware. The ARs were measured on a per-process basis in the six designs, but for the purposes of modeling, process activity is specified as two sets of processes, those with a 100% AR, and those with a 0% AR. The ratio of the active processes to the total number of processes is the active process ratio (APR), and if the processes are assumed to be the same size, it is equivalent to the activity footprint. For example, an APR of 60% means that 60% of the processes are always active, and 40% of the processes are never active. In the *C* model, process activity is determined through a call to a function which returns zero for inactive processes and non-zero for active processes for every simulation cycle.

APRs of 17 : 35 and 30 : 35 are shown in Figure 10 with an APR of 1 : 1 as a baseline. This single graph demonstrates the benefit to speedup by exploiting executive locality. It shows that the lower the APR, the less acceleration hardware is required. The maximum speedup for each plot also shows that overhead is reduced because inactive processes are not migrated to hardware.

6.2. Route calculation

Another consideration in a migration system is the effect of establishing interconnect. It is difficult to estimate the time required to build connections between RMs at run time without a full implementation, but the graph of Figure 11 shows that if the time is long enough, the speedup can be less than one. This graph shows the route times in terms of

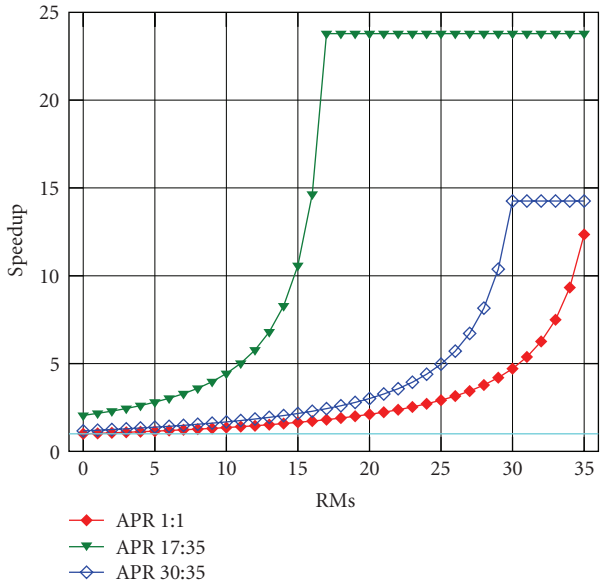


Figure 10: Modeled effect of APR on speedup.

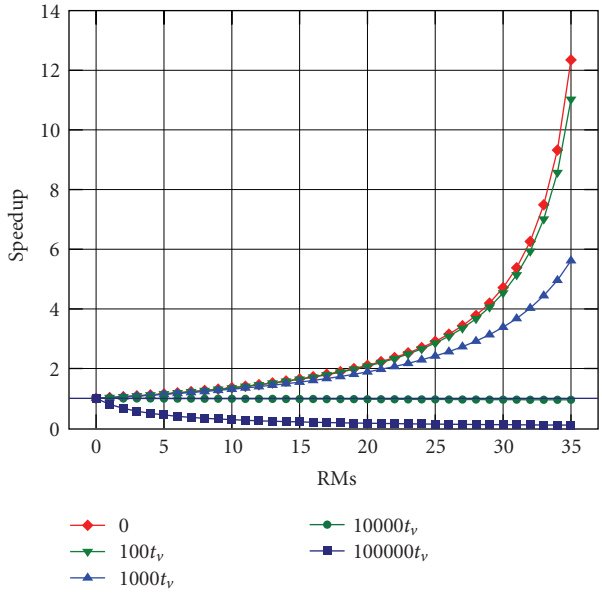


Figure 11: Serial route calculation (10 k simcycles).

t_v because the processor speed, which is reflected in t_v, has a direct effect on the route calculation time.

A possible optimization is to continue the simulation using software connectivity while the routes between RMs are calculated in parallel by another processor. (The implementation of Section 5.1 contains two PowerPC processors.) In the algorithmic model, the behavior is modeled by keeping track of the amount of time left until a route is complete. While the time remaining to complete a route is greater than zero, the model uses software connectivity. When the timer reaches zero, the model uses hardware connectivity. The results of such an algorithm are shown in Figure 12. The knees in the plots of the shorter simulations are due to

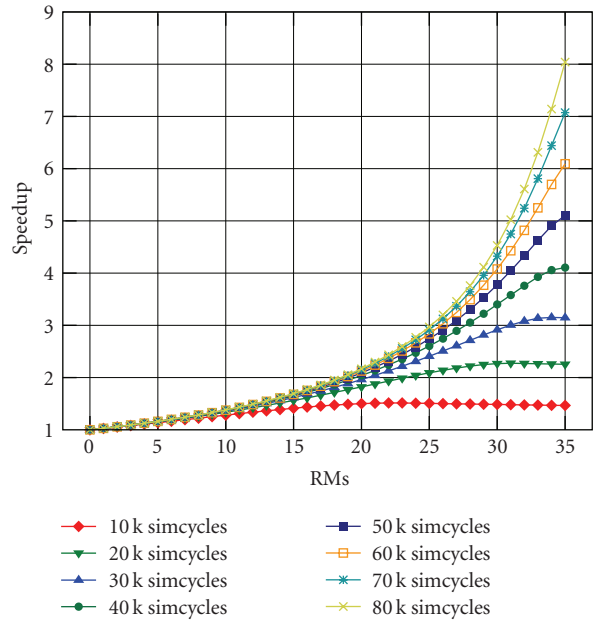


Figure 12: Parallel route calculation (10000 t_v).

insufficient time to complete the parallel route calculation. That is, the simulation ends before the route calculations are complete. Longer simulations allow the parallel route calculation to complete, and they benefit from the faster hardware connectivity.

7. CONCLUSION

We have shown that the RTL code of various designs demonstrates executive locality of reference which can be exploited by a process cache. The larger designs exhibit activity footprints below 50% in many cases and less than 75% in almost all cases. The smaller designs—SPI and ATA in particular—do not exhibit as much inactivity, but smaller designs are also less likely to require simulation acceleration. Traditional RTL accelerators require a mapping of the RTL code to the hardware prior to simulation, and that mapping remains static. Since many processes are idle, and since activity can change during simulation prior, static mapping to hardware does not maximize efficiency. Hardware/software process migration is one means of implementing a process cache to ensure that only active processes are accelerated.

Existing FPGAs, however, are not designed to be efficiently run-time reconfigurable; nevertheless an implementation using an existing FPGA allows us to determine and model the hurdles to efficient processes migration. The largest hurdles are the migration time due to reconfiguration and, potentially, the building of run-time routes. As run-time reconfiguration becomes more prevalent, and as the FPGA architectures are developed to improve reconfiguration performance, the migration time is expected to decrease. The potential bottleneck of run-time routing may also be overcome through the use of on-chip networks where communication occurs during simulator overheads.

Nevertheless, speedups of greater than ten are exhibited in many of the modeled scenarios. Further work is also warranted in comparing the trade-offs between a large number of small, communicating processors, and run-time synthesis of processes.

ACKNOWLEDGMENT

This work has been graciously funded through Virginia Tech College of Engineering and Bradley Fellowships.

REFERENCES

[1] Intel Corporation, "Moore's law: raising the bar," 2005, ftp://download.intel.com/museum/Moores_Law/Printed_Materials/Moores_Law_Backgrounder.pdf.

[2] S. Kakkar, "Proactive approach needed for verification crisis," EETimes, April 2004.

[3] A. Raynaud, "The new gate count: what is verification's real cost?" Electronic Design, October 2003.

[4] B. Bower, "The 'what and why' of TLM," EETimes, March 2006.

[5] P. Varhol, "Is software the new hardware?" EETimes, August 2006.

[6] R. Goering, "Tools ease transaction-level modeling," EETimes, January 2006.

[7] C. Chang, J. Wawrzynek, and R. W. Brodersen, "BEE2: a high-end reconfigurable computing system," *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.

[8] G. D. Peterson, "Evaluating simulation acceleration techniques," in *Proceedings of the Enabling Technology for Simulation Science V*, vol. 4367 of *Proceedings of SPIE*, pp. 127–136, Orlando, Fla, USA, April 2001.

[9] Cadence Design Systems, "Palladium accelerator/emulator," 2003, http://www.cadence.com/products/functional_ver/palladium/index.aspx.

[10] Cadence Design Systems, "Xtreme server," 2005, http://www.cadence.co.in/products/functional_ver/xtreme_server/index.aspx.

[11] J. Bauer, M. Bershteyn, I. Kaplan, and P. Vyedin, "A reconfigurable logic machine for fast event-driven simulation," in *Proceedings of 35th Design Automation Conference (DAC '98)*, pp. 668–671, San Francico, Calif, USA, June 1998.

[12] EVE Corporation, http://www.eve-team.com/.

[13] S. Cadambi, C. S. Mulpuri, and P. N. Ashar, "A fast, inexpensive and scalable hardware acceleration technique for functional simulation," in *Proceedings of 39th Design Automation Conference (DAC '02)*, pp. 570–575, ACM Press, New Orleans, La, USA, June 2002.

[14] R. Goering, "Startup liga promises to rev simulation," EETimes, 2006.

[15] R. D. Smith, *Simulation Article*, Encyclopedia of Computer Science, Nature Publishing, New York, NY, USA, 4th edition, 2000.

[16] D. Döhler, K. Hering, and W. G. Spruth, "Cycle-based simulation on loosely-coupled systems," in *Proceedings of the 11th Annual IEEE International ASIC Conference*, pp. 301–305, Rochester, NY, USA, September 1998.

[17] IEEE Computer Society, "IEEE Standard VHDL Language Reference Manual (Std 1076-2002)," Institute of Electrical and Electronics Engineers, 2002.

[18] IEEE Computer Society, "IEEE Standard for Verilog® Hardware Description Language (Std 1364-2005)," Institute of Electrical and Electronics Engineers, 2006.

[19] J. Hennesey and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 1990.

[20] R. Razdan, G. P. Bischoff, and E. G. Ulrich, "Clock suppression techniques for synchronous circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 10, pp. 1547–1556, 1993.

[21] Opencores.org, 2006, http://www.opencores.org/.

[22] XilinxInc, "Xilinx University Program: Xilinx XUP Virtex-II Pro Development System," 2005, http://www.xilinx.com/univ/xupv2p.html.

[23] XilinxInc, "Virtex-II Pro and Virtex-II Pro X FPGA User Guide," Xilinx, Inc., March 2005.

[24] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Addison Wesley, New York, NY, USA, 2nd edition, 2003.

[25] D. R. Curd, "Xapp660: dynamic reconfiguration of RocketIO MGT attributes," February 2004, http://www.xilinx.com/support/documentation/application_notes/xapp660.pdf.