International Journal of Reconfigurable Computing

Selected Papers from the 17th Reconfigurable Architectures Workshop (RAW2010)

Guest Editors: Aravind Dasu, João M. P. Cardoso, Eli Bozorgzadeh, and Jürgen Becker

Selected Papers from the 17th Reconfigurable Architectures Workshop (RAW2010)

International Journal of Reconfigurable Computing

Selected Papers from the 17th Reconfigurable Architectures Workshop (RAW2010)

Guest Editors: Aravind Dasu, João M. P. Cardoso,
Eli Bozorgzadeh, and Jürgen Becker

Editorial Board

Contents

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 574972, 2 pages
doi:10.1155/2011/574972

Editorial

Selected Papers from the 17th Reconfigurable Architectures Workshop (RAW2010)

Aravind Dasu,[1] João M. P. Cardoso,[2] Eli Bozorgzadeh,[3] and Jürgen Becker[4]

[1] *Energy Dynamics Laboratory, Utah 84341, USA*
[2] *Departamento de Engenharia Informática, Faculdade de Engenharia Universidade do Porto (FEUP), Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal*
[3] *UC Irvine, Irvine, CA 92697, USA*
[4] *Karlsruhe Institute of Technology (KIT), 76021 Karlsruhe, Germany*

Correspondence should be addressed to Aravind Dasu, aravind.dasu@gmail.com

Received 18 April 2011; Accepted 20 April 2011

This special issue of IJRC includes papers selected from the 17th Reconfigurable Architectures Workshop (RAW 2010) held in Atlanta, GA, USA, in April 29-30, 2010. RAW 2010 was associated with the 24th Annual International Parallel & Distributed Processing Symposium (IPDPS 2010) and was sponsored by the IEEE Computer Society Technical Committee on Parallel Processing. The workshop is one of the major meetings for researchers and practitioners to present ideas, results, and ongoing research, on both theoretical and practical advances in reconfigurable computing.

The program of RAW 2010 provided a venue to facilitate creative and productive interaction between most disciplines of reconfigurable computing. High-quality research papers, focusing models and architectures, devices, algorithms and applications, and design technologies and tools were presented. This special issue aims at providing an important venue to show most of the best RAW's 2010 scientific contributions to reconfigurable computing, here in extended and improved versions. This special issue includes extended and improved papers selected from the peer-reviewed papers that were presented at RAW 2010 and consists of the seven following papers.

The paper "A Vector-like reconfigurable floating-point unit for the logarithm," by N. Alachiotis and A. Stamatakis, presents a pipelined datapath for calculating the logarithm function of data represented in single and double-precision floating point in FPGAs. A vector-like extension is also presented, which is capable of calculating two results/cycle. They compare the performance of the architectures with the software implementations of GNU and Intel Math Kernel Library (MKL). Their approach is able to accelerate the logarithm function even when comparing to highly optimized software solutions running in current microprocessors.

The paper "A streaming high-throughput linear sorter system with contention buffering," by J. Ortiz and D. Andrews, presents a sorting unit consisting of multiple linear sorters and implemented in an FPGA. The linear sorters run in parallel to achieve higher throughput. The architecture presented in this paper is able to process multiple data values per clock cycle and presents speedups over a quicksort software solution running in a MicroBlaze softcore.

The paper "The potential for a GPU-like overlay architecture for FPGAs," by J. Kingyens and J. G. Steffan, proposes a soft processor inspired by GPUs. The architecture supports multithreading, vector operations, predication, and pipelining. They highlight the mechanisms for managing threads and register files that maximize data-level and instruction-level parallelism. An efficient approach to overcome the challenges of port limitations of the FPGA block memories is also presented in this paper. The proposed architecture can be programmed in the same programming model of GPUs and can be an interesting option for accelerating applications suitable to be mapped to GPU-like architectures.

The paper "Boosting parallel applications performance on applying DIM technique in a multiprocessing environment," by Mateus Rutzig et al., extends the DIM (dynamic instruction merging) approach to be used in a multiprocessor scenario. They present experimental evidence that

the acceleration achieved by the parallelism met using multi-core architectures can be effectively improved by augmenting the ILP (instruction-level parallelism) in each core.

The paper "High-level synthesis of in-circuit assertions for verification, debugging, and timing analysis," by J. Curreri et al., presents an approach to support the integration of ANSI-C assertions when synthesizing C code to FPGAs. The approach enables the implementation of assertions in hardware. This can be important to verify and debug circuits generated by high-level synthesis. Furthermore, the approach also supports timing assertions and thus, can be used to verify timing properties.

The paper "Floorplacement for partial reconfigurable FPGA-based systems," by A. Montone et al., presents a framework for floorplacement which is aware of resources and configuration tailored for Xilinx V4 and V5 FPGAs. Their approach identifies reconfigurable functional units that can share the same area of the FPGA and highlights the reduction of external wirelengths and the reuse of communication links compared to using purely area-driven approaches.

The paper "Operating system for runtime reconfigurable multiprocessor systems," by D. Göhringer et al., presents an operating system able to do scheduling, resource allocation, and reconfiguration and to manage accesses to the internal configuration access port (ICAP). The operating system is used in the context of RAMPSoC, a runtime adaptive multiprocessor system-on-chip. In this SoC, the hardware tasks are transferred to the reconfigurable hardware via the configuration access port, and the software tasks can be loaded into the local memory of each microprocessor by using also a configuration access port or the on-chip communication infrastructure.

Aravind Dasu
João M. P. Cardoso
Eli Bozorgzadeh
Jürgen Becker

Acknowledgments

We are grateful to the anonymous reviewers of this special session and of ARC 2010. We also thank René Cumplido, the Editor-in-Chief of IJRC, for his help and support from the very beginning. Eman Bastawy, from the editorial staff of Hindawi Publishing Corporation, was always very helpful. She was also our support to have this special issue on time. We hope you enjoy reading these papers, and we also hope that these papers contribute to your work.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 341510, 12 pages
doi:10.1155/2011/341510

Research Article

A Vector-Like Reconfigurable Floating-Point Unit for the Logarithm

Nikolaos Alachiotis and Alexandros Stamatakis

The Exelixis Lab, Scientific Computing Group, Heidelberg Institute for Theoretical Studies, 69118 Heidelberg, Germany

Correspondence should be addressed to Nikolaos Alachiotis, n.alachiotis@gmail.com

Received 27 July 2010; Accepted 14 January 2011

Academic Editor: Aravind Dasu

The use of reconfigurable computing for accelerating floating-point intensive codes is becoming common due to the availability of DSPs in new-generation FPGAs. We present the design of an efficient, pipelined floating-point datapath for calculating the logarithm function on reconfigurable devices. We integrate the datapath into a stand-alone LUT-based (Lookup Table) component, the LAU (Logarithm Approximation Unit). We extended the LAU, by integrating two architecturally independent, LAU-based datapaths into a larger component, the VLAU (vector-like LAU). The VLAU produces 2 results/cycle, while occupying the same amount of memory as the LAU. Under single precision, one LAU is 12 and 1.7 times faster than the GNU and Intel Math Kernel Library (MKL) implementations, respectively. The LAU is also 1.6 times faster than the FloPoCo reconfigurable logarithm architecture. Under double precision, one LAU is 20 and 2.6 times faster than the respective GNU and MKL functions and 1.4 times faster than the FloPoCo logarithm. The VLAU is approximately twice as fast as the LAU, both under single and double precision.

1. Introduction

The use of FPGAs as accelerators for compute-intensive codes is driven by their potential for implementing deeply pipelined architectures and for executing hundreds of operations in parallel. As the devices become larger, new fabrics, in particular DSPs, allow for a wider range of applications, in particular floating-point intensive codes, to be efficiently executed/accelerated on FPGAs.

A large number of scientific applications rely on the frequent and efficient computation of the logarithm function. For instance, multimedia codes need to estimate log-likelihood scores for Gaussian mixture models [1], or bioinformatics programs for evolutionary reconstruction under the maximum likelihood model [2] need to compute log-likelihood scores of evolutionary trees. The logarithm is also commonly used to avoid numerical underflow (especially in statistics) by replacing multiplications via additions.

Many of the applications that rely on the logarithm are either highly compute intensive, such as the phylogenetic likelihood function which represents an important computational kernel in computational Biology [3, 4], or exhibit real-time constraints, such as real-time image processing applications [5] or skin segmentation algorithms [6]. Irrespective of the specific type of application, the deployment of reconfigurable logic (FPGAs) represents a common technique to either speed up applications, prototype hardware designs, or to meet real-time requirements of time-critical applications.

When an FPGA is used for accelerating floating-point intensive applications, a thorough exploration of the performance and precision tuning parameter space for the required arithmetic operators can eventually lead to significant performance improvements. In fact, implementations of simple floating-point operators like adders or multipliers may require fairly complex reconfigurable architectures. Furthermore, the amount of hardware resources used by a floating-point operator generally increases with precision/accuracy requirements. However, accuracy requirements of a specific operator may depend on the application at hand.

In previous work on calculating the logarithm function [7] using reconfigurable logic, we focused on the design of a pipelined Logarithm Approximation Unit (LAU). We demonstrated that the LAU is sufficiently accurate for

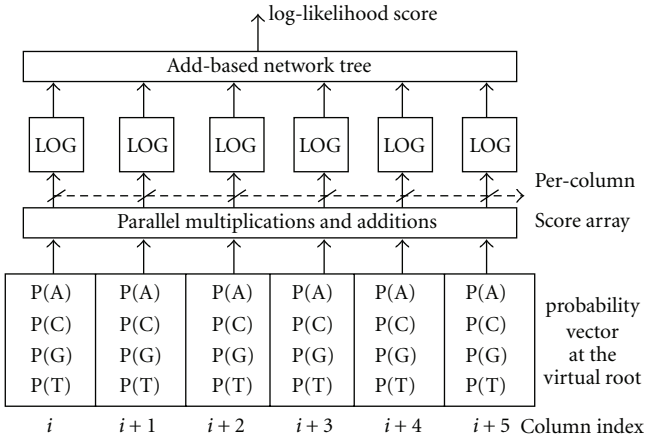


Figure 1: Parallel placement of logarithm units for the calculation of the log-likelihood score for a phylogenetic tree topology.

computing the phylogenetic maximum likelihood (ML) function on a reconfigurable coprocessor for RAxML [8]. RAxML is a widely used bioinformatics code for reconstructing evolutionary trees (evolutionary histories or simply phylogenies) from DNA or protein data under the ML criterion. The LAU architecture utilizes look up tables (LUTs) for calculating the logarithm and can be conveniently adjusted to provide the desired/required application-specific accuracy. The LAU is based on the ICSILog approximation method (Vinyals and Friedland in [9]) that is available as open-source code. If not stated otherwise, in this paper, we use the term LUT for referring to the look up table required by the ICSILog approximation method rather than to the low-level hardware LUTs on the FPGA device.

As already mentioned, several computational units must be placed on the FPGA and operate in parallel to efficiently exploit the available computational resources. Thus, the resource requirements of a component/unit (e.g., a simple floating-point operator or a more complex arithmetic function) need to be minimized to allow for placing several instances on the chip that will then operate in parallel. The input/output (I/O) requirements can be accommodated by parallel I/O ports, for instance, by organizing the embedded memory blocks of a device into several smaller parallel blocks that can provide a sufficient throughput with respect to the arrangement of the parallel components. In Figure 1, we provide a representative example for the potential arrangement of logarithm components and the respective parallel execution of the logarithm function. The block diagram depicts a fine-grain parallelization of log-likelihood score computations for a given evolutionary tree topology under a likelihood-based model (used, e.g., in maximum likelihood or Bayesian phylogeny programs).

In the current paper, we present a significantly extended and optimized vector-like LAU implementation. The vector-like Logarithm Approximation Unit (VLAU) can calculate two logarithms within the same clock cycle. Using the VLAU is more resource-efficient compared to instantiating and using two simple independent LAUs in parallel. The underlying idea of the VLAU consists of exploiting the dual-port configuration option of embedded memory blocks. This implementation option allows for sharing LUTs between two, otherwise completely independent, LAU-based pipelines. Furthermore, a detailed analysis of resources requirements and performance impact with respect to the latency of the LAU has been conducted for the single and double precision versions. We also extended the C implementation of the ICSILog algorithm (International Computer Science Institute) to support double precision (DP) arithmetics.

Throughout the paper, we denote IEEE-754 single precision arithmetics as SP and IEEE-754 double precision arithmetics as DP. We denote the single precision software implementation of ICSILog (version 0.6 beta) as SP-ICSILog and our DP software implementation as DP-ICSILog. By SP-LAU, DP-LAU, SP-VLAU, and DP-VLAU we denote the SP and DP FPGA implementations of the LAUs and VLAUs, respectively.

The DP-ICSILog C code as well as the hardware descriptions for the LAUs and the VLAUs (including all available latency variants) are available as an open source code for download at http://wwwkramer.in.tum.de/exelixis/logFPGA.tar.bz2. The default hardware configuration that supports both, Virtex 4 and Virtex 5 FPGAs uses an LUT with 4,096 entries. We also provide several COE files for different LUT sizes, such that the LAU/VLAU can be conveniently reconfigured and adapted to the precision required by the respective target application.

The remainder of this paper is organized as follows. Section 2 describes the underlying ideas of the ICSILog algorithm. In Section 3, we review related work on logarithmic units for FPGAs. The LAU architecture is described in Section 4, and the VLAU architecture is introduced in Section 5. In the following Section 6, we present speed and accuracy measurements for LAUs and VLAUs with a LUT-size of 4,096 and provide a detailed evaluation of LAU implementations with latencies ranging between 5 and 22 clock cycles. We also analyze performance and resource utilization of the FPLog implementations and assess the numerical stability of RAxML in software using DP-ICSILog. We conclude in Section 7.

2. The ICSILog Algorithm

The underlying idea of the ICSILog algorithm consists of increasing the speed of the logarithm computation by using an LUT that resides entirely in the CPU cache. The algorithm exploits the floating point number representation of the IEEE-754 standard. An IEEE floating-point number consists of three fields: the sign (sgn), the exponent (exp), and the mantissa (man). The decimal floating-point value of a number (num) is represented by the sign, followed by the product of the mantissa and the factor 2^{exp}:

$$\text{num} = (\pm)2^{\text{exp}} * \text{man}. \quad (1)$$

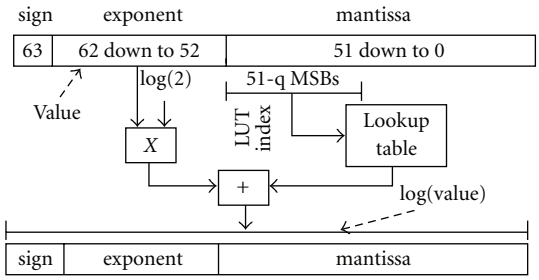


Figure 2: Outline of the ICSILog Algorithm.

In order to calculate the logarithm of num, one can use the multiplicative property of the logarithmic function and decompose the computation as follows:

$$\begin{aligned}\log(\text{num}) &= \log(2^{\text{exp}} * \text{man}) \\ &= \log(2^{\text{exp}}) + \log(\text{man}) \\ &= \text{exp} * \log(2) + \log(\text{man}).\end{aligned} \tag{2}$$

Since the real-valued logarithm is only defined for positive numbers, the sign bit can be discarded. The factor by which exp is multiplied is a constant and only depends on the base of the logarithm; one may use $\log_e(2)$, $\log_2(2)$, or $\log_{10}(2)$ for instance. Thus, the calculation of the logarithm for an arbitrary base x only requires the constant $\log_x(2)$ and an appropriately initialized full-size LUT (comprising all values) for the base x.

The calculation of the first part of the sum $\text{exp} * \log(2)$ requires the floating-point representation for the decimal value of the exponent field. One can use the Xilinx floating-point operator (FPO) [10] to obtain this value. However, we use a faster LUT-based method (this is a separate LUT that is exclusively used for this conversion) to obtain the floating point value which is described in Section 4. In Section 6, we provide a performance comparison between the Floating-Point Operator provided by Xilinx and our approach. Once the floating point value of the exponent is available, the first operand of the final addition is calculated by conducting the multiplication with the constant floating-point value.

The calculation of the second part of the sum, that is, the logarithm of the mantissa, requires the use of an LUT. A naïve LUT will thus need to contain all precomputed values for *log(man)* which requires 32 MB of memory for the SP number range. Vinyals and Friedland found that, the usage of a 32 MB full-size LUT only yields insignificant performance improvements with respect to the GNU implementation [9]. To improve performance and reduce LUT size, they deploy a quantized mantissa that entirely fits into cache memory. In Figure 2, we provide a schematic outline of the Vinyals and Friedland algorithm at the bit level.

The mantissa LUT is indexed by using the $23 - q$ most significant bits of the mantissa under SP and the $52 - q$ most significant bits under DP, respectively. The variable q is the number of least significant bits of the mantissa that will be ignored by the quantization process. Thus, the variable q can be used to appropriately adapt the accuracy and LUT size to the specific requirements of an application. A detailed study of the accuracy loss that is induced by quantizing the mantissa can be found in [9]. The tradeoff between accuracy and embedded memory hardware resources used will be discussed in Section 6.

3. Related Work

A thorough bibliographical search revealed that alternative implementations of fast logarithm algorithms mostly represent special purpose solutions that are tailored to a specific application or hardware platform, that is, there is a lack of a generally applicable solution.

Dedicated software implementations that entail approximation algorithms for the logarithmic function have been developed for accelerating multimedia applications [9, 11]. In 2001, de Soras proposed and made available an algorithm called fast log [11]. This algorithm computes a 3rd order Taylor series approximation of the logarithm for any given IEEE-754 floating-point number. The algorithm is fast but lacks accuracy in certain cases/number ranges [9]. The LUT-based approach of ICSILog, which we implemented in reconfigurable logic in the LAU and VLAU components, is as fast as fast log but provides better accuracy [9].

De Dinechin et al. have developed FloPoCo (floating-point cores), an open-source arithmetic core generator for FPGAs [12]. The logarithmic unit generated by FloPoCo (FPLog) supports SP (SP-FPLog), DP (DP-FPLog), and user-defined number formats. The FPLog units can be configured to yield *exactly* the same results as the respective GNU functions; hence, accuracy comparisons between our LAU and FPLog are identical to comparisons between the LAU and the GNU library. The algorithms and implementation techniques that are deployed in FloPoCo for generating the FPLog unit are described in [13, 14].

Section 6 includes a direct comparison between the LAU and FPLog units (using the most recent version 2.0.0 of FloPoCo) in terms of speed and resource utilization on a Virtex 5 FPGA. Note that the FPLog input format slightly differs from the IEEE-754 standard. Two additional bits in every input number indicate whether the input should be treated as special number (*zero, nan,* (+/−) *inf*) or as normal number. Thus, in order to integrate the FPLog component into a design that complies with the IEEE-754 standard, this dedicated input number format specification requires additional logic (which can also be separately generated by FloPoCo) to appropriately set these bits. Furthermore, a common FPGA design paradigm is event-driven architectures. Unfortunately, the FPLog interface does not provide any additional ports for validity input signals, that is, signals that indicate whether the current values at the input and output ports are valid or not. Consequently, FPLog is harder to integrate with event-driven architectures.

National instruments [15] have designed a high-throughput natural logarithm function for FPGAs. The specific design is only available commercially, and only a limited amount of information is provided regarding unit performance. The implementation only supports fixed-point arithmetics, and the input arguments must be unsigned

(input number range: $[1/e, 1)$). For numbers outside this input range, the unit generates undefined results. The interface of the logarithm component provides all necessary ports for validity input signals for easy integration with and use in event-driven environments. The CORDIC algorithm (COordinate Rotation DIgital Computer [16]) is deployed for the specific implementation, and the user can set the desired accuracy level by defining the number of iterations in the CORDIC algorithm. Because this logarithm implementation is not available (not even for a short evaluation period), we were not able to conduct a respective performance comparison with the LAU/VLAU architectures.

Tropea [17] has also presented an area-optimized FPGA implementation to compute the base-N logarithm function. An important aspect of the specific implementation is that it can be mapped on FPGAs from any vendor. Performance results for Xilinx [18] and Actel [19] FPGAs are provided in [17]. The error analysis section in [17] reveals that highly accurate results can be obtained, while only using a small fraction of the overall hardware resources. The unit utilizes the multiplicative normalization method [20] to calculate the logarithm, and several configurations with various precision levels are evaluated.

Recently, Chrysos et al. [21] presented a general reconfigurable architecture for a Bioinformatics algorithm that uses Interpolated Markov Models (IMMs) for gene finding, known as the Glimmer algorithm. The Glimmer algorithm also requires logarithm calculations. The respective hardware architecture contains 6 logarithm unit instances that operate in parallel. The design of the logarithm component in the Glimmer architecture [21] deploys a similar strategy as the LAU.

In 2008, Raygoza-Panduro et al. [22] presented an automatically generated mathematical unit. The hardware description is automatically generated by a JAVA program and can be synthesized. The framework supports a wide range of complex arithmetic operations. The mathematical unit was used to implement a sliding mode controller for a magnetic levitation system. The sytem provides operators for the natural logarithm and the base-10 logarithm. The automatically generated mathematical unit was mapped to a Virtex II FPGA; the logarithm functions (natural and base-10) only occupy 1% of available FPGA slices and 1% of available LUTs on the device. The resource-efficiency of the unit appears to be mainly induced by the usage of bit-width reduced floating-point arithmetics, that is, only 3 bits are used for the exponent field and only 14 bits for the mantissa field, respectively.

4. The LAU Architecture

In the following, we describe the design of a reconfigurable architecture for the ICSILog algorithm. In Figure 3, we provide the block diagram of the top-level unit.

The leftmost module is the *special_case_detector*. As the name suggests, this module assesses whether the LAU input is valid or not. Special cases are negative numbers, *nan*, $-inf$, and *inf* as defined by the IEEE standard. Since the logarithm is not defined on negative numbers, the result is

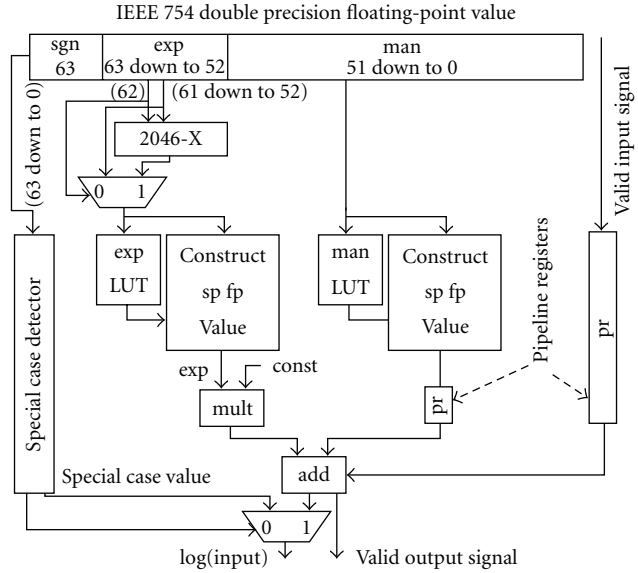


Figure 3: Block diagram of LAU.

nan. For *nan* and $-inf$ inputs, the result is defined as *nan* as well. For an *inf* input, the unit will return *inf* again. The module consists of comparators, logic gates, and pipeline registers that detect the special case inputs and produce the corresponding output. The module also outputs a selection signal for the final 2 to 1 multiplexer (bottom left in Figure 3) that is connected to the output port of the LAU.

To the right of the *special_case_detector* in Figure 3, we have integrated a group of modules that operate on the exponent bits of the input. These modules compute the first operand of the addition that returns the approximation of the logarithm.

Initially, the decimal value of the exponent field needs to be transformed into a floating-point number. The straightforward approach to implement this operation is to use the Xilinx FPO [10] (fixed-to-float) operator. However, we deploy an LUT-based approach to carry out this transformation more efficiently. The *exp_LUT* lookup table in Figure 3 is used for this purpose. Note that this LUT is a special component of our hardware implementation and should not be confused with the mantissa LUT of the ICSILog algorithm (*man_LUT*). Details about the performance and resource tradeoffs between our approach and the alternative design using the Xilinx FPO are provided in Section 6.4.

Internally, all operations are conducted under SP. For the SP-LAU, the *exp_LUT* contains 128 entries (2^{8-1}), while for the DP-LAU, there are 1,024 entries (2^{11-1}), where 8 and 11 are the number of bits that represent the exponent field of an SP and a DP value, respectively. The reason why the size of the *exp_LUT* can be reduced by 50% is explained in the next paragraph. Each entry of the *exp_LUT* contains a total of 9 bits in the SP-LAU and 13 bits in the DP-LAU. The first 3 bits under SP and the first 4 bits under DP are the least significant bits of the exponent field of the floating-point number representation we intend to construct. The remaining 6 (SP) and 9 bits (DP) are the most significant

bits of the mantissa. The remaining bits of the exponent field are always set to 10000 for SP and 1000 for DP. Note that, at this point, an SP value is being constructed for the DP-LAU as well. The remaining bits of the mantissa are all set to zero.

One can observe that there is a correspondence between the decimal values of the exponent field and the exponents themselves. For DP, while the decimal value ranges from 0 to 2,047, the exponent ranges from −1,023 to 1,024. This correspondence can be used to reduce the size of *exp_LUT* by 50%, via only storing the bits required to represent floating-point numbers in the range 0−1,023. To support the full range (0–2,047), we use additional logic. More specifically, the 11-bit mantissa is transformed into a 10-bit index for *exp_LUT* by subtracting the 11-bit value from 2,046. For example, an 11-bit index with a decimal value X in which the most significant bit is set indexes a lookup table entry >1,023. Hence, $X - 1,023$ provides the distance from the last entry of the lookup table with 1,024 entries. Thus, $1,023 - (X - 1,023) = 2,046 - X$ will yield the correct 10-bit index for a *exp_LUT* with half the size. The most significant bit of the exponent field (discarded from the index) becomes the sign of the newly constructed floating-point value. After this transformation, the resulting floating-point number becomes the first operand of the multiplication; the second operand is a constant value. The overall result produced by this part of the architecture is the first operand of the final addition: $\exp * \log(2) + \log(\text{man})$.

The *man_LUT* module in Figure 3 is the standard quantized LUT of the ICSILog algorithm and contains pre-calculated values of logarithms. We therefore used ICSILog to generate the contents of *man_LUT*. As previously described, the most significant bits of the mantissa are used for indexing the *man_LUT*. Each entry of the table (for SP and DP values) consists of an SP floating-point number. As outlined in Section 6, one can increase the accuracy of the LAU by increasing the size of *man_LUT*. For example, in a *man_LUT* of size 4,096, only the 12 most significant bits of the mantissa field of the input value will be used for indexing. Both lookup tables (*exp_LUT* and *man_LUT*) are enhanced by a *construct_sp_fp_value* unit. These units consist of logic gates, registers, and multiplexers which are used to construct the correct floating-point representations from the respective LUT entries. Finally, the sum of the two values generated by *exp_LUT*, *man_LUT*, and the respective *construct_sp_fp_value* units will return an approximation of the logarithm that is identical to the ICSILog software.

As already mentioned, all operations are conducted under SP. Thus, for the SP-LAU, the result is simply the output of the final adder. For DP, the result is transformed into DP by appropriately adapting the bit indices of the SP representation. The least significant bits of the mantissa are set to zero, and a bit extension for the most significant bits of the exponent is conducted while maintaining its sign.

The usage of SP arithmetic, even for the DP-LAU, does not affect the precision of the output because of the approximation strategy that is being used. DP will only be affected if a *man_LUT* with more than 2^{23} entries is used (23 is the number of bits in the mantissa field of SP numbers in the IEEE standard). In this case, the mantissa LUT would require 32 MB of memory. Currently, there is no FPGA available with such a large amount of embedded memory. Clearly, the savings in terms of FPGA resources (embedded memory and DSP slices) by internally using SP for our LAU design are significant. Note that, in our DP-ICSILog software implementation, we transformed the entire algorithm to DP, because the SP algorithm with a type casting operation from `float` to `double` in C was slower than a direct implementation under DP.

5. The VLAU Architecture

An additional optimization can be applied to the LAU architecture (Section 4), when several parallel LAUs shall be placed on an FPGA device. This optimization is based on a special feature of embedded memory blocks in new-generation FPGAs, which can be configured as so-called dual-port memories.

Each memory block provides two fully independent ports that yield access to a shared memory space. An appropriate reconfiguration of the LAU look up tables (*exp_LUT*, *man_LUT*) for usage as dual-port ROMs (Read Only Memories) allows two independent LAUs to use the same memory blocks for lookups.

Figure 4 depicts the optimized VLAU architecture (vector-like LAU). The shared memory area in the middle of Figure 4 (denoted as *shared LUTs*) contains the *exp_LUT* and *man_LUT* look up tables. The two LAU-based pipelines are located to the left and the right of the *shared LUTs* in Figure 4. These two pipelines are exact copies of the LAU architecture (Section 4), but the LUTs have been moved to a *shared* memory area. The individual LAU pipelines are architecturally completely independent from each other, since they only share a read-only memory area. Each LAU pipeline only accesses one of the two ports of the shared LUTs.

The VLAU architecture is well suited for vector-processing, since it can accommodate the computation of two logarithms in one cycle. Because the same pipeline design is used for the LAU and VLAU, a two-unit VLAU is as fast as two independent LAUs. The main advantage of a two-unit VLAU over two independent LAUs is that the VLAU only requires 50% of the memory blocks.

The FPGA-based coprocessor for gene finding by Chrysos et al. [21] represents an example of an architecture that could potentially benefit from the memory-efficient VLAU implementation. The Glimmer architecture is memory intensive and the attained level of parallelism was limited by the number of available embedded memory blocks in the device (personal communication with G. Chrysos; June 14th, 2010). The deployment of VLAUs can thus help to reduce the number of memory blocks required for computing the logarithm and thereby increase the degree of parallelism in the Glimmer architecture.

6. Experimental Results

Initially, we verified the functionality of the LAU/VLAU architectures (Section 6.1). In the following two sections, we

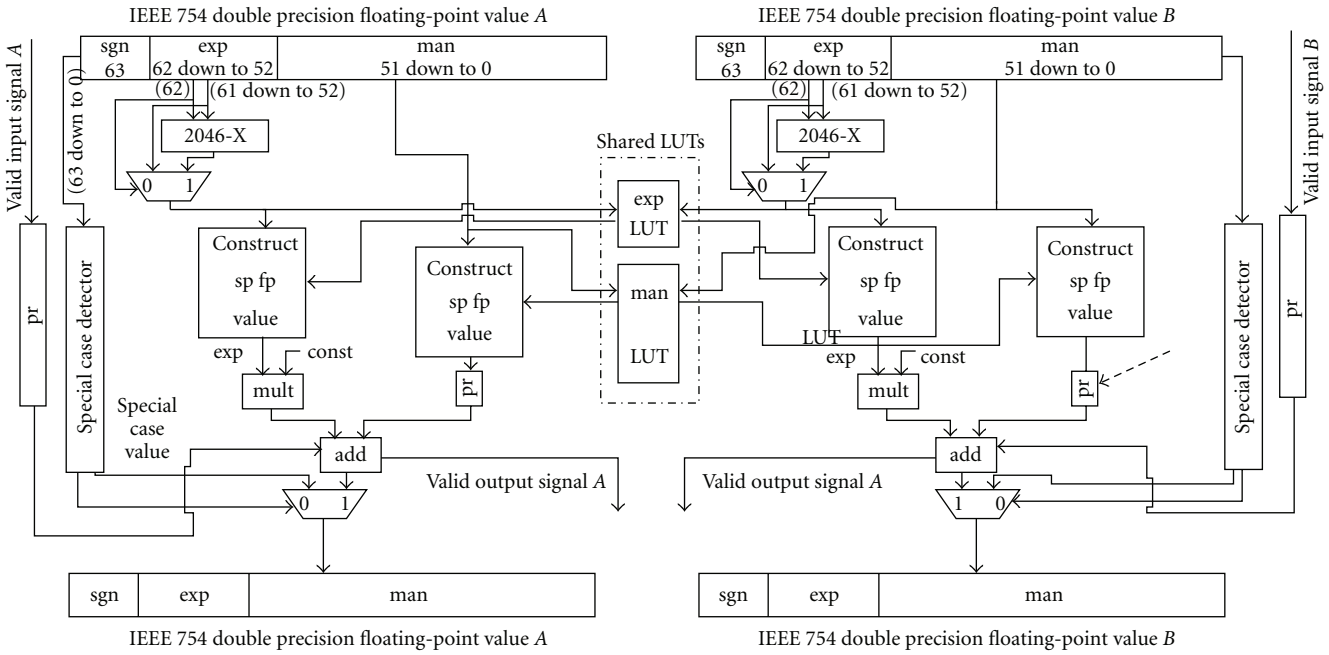


Figure 4: Top-level design of the VLAU architecture.

investigate the behavior of RAxML [8] using DP-ICSILog (Section 6.2) and assess the accuracy of the implementation (Section 6.3). Thereafter, Section 6.4 provides a detailed resource usage and performance evaluation for LAUs with various latencies and also for the VLAU architecture. We also compare performance and resource utilization with the FloPoCo logarithm [12]. A thorough run time comparison between the LAU/VLAU architectures and respective software implementations (GNU and MKL [23]) is presented in Section 6.5. Note that all results in Section 6 refer to Xilinx reports as obtained after the implementation process (post-place and route).

6.1. Verification. In order to verify the correctness of the proposed architectures, we conducted extensive post-place and route simulations as well as tests on an actual FPGA. As a simulation tool, we used Modelsim 6.3f by Mentor Graphics. For hardware verification, we used the HTG-V5-PCIE development platform equipped with a Xilinx Virtex 5 SX95T-1 FPGA.

Initially, the advanced verification tool Chipscope Pro Analyzer was used to monitor the output ports of the SP/DP-LAUs and SP/DP-VLAUs, and the expected signals for given input numbers were tracked. Thereafter, an experimental PC-FPGA platform was set up. We use Gigabit Ethernet to communicate between the PC and the FPGA board based on the optimized unit for direct UDP/IP-based PC-FPGA communication that we recently made available [24]. A JAVA test application was used on the PC side to generate random SP and DP input values (using the standard `java.util.Random` class), organize the numbers into bytes, and transmit them to the FPGA. On the FPGA side, the floating-point representations were reconstructed from the incoming bytes and forwarded to the LAU/VLAU components. The logarithms of the inputs were then sent back to the JAVA test application on the PC from the FPGA and printed to screen.

Table 1: Log-likelihood score deviation with DP-ICSILog.

Dataset	DP-GNU	DP-ICSILog
44 organisms	−11231.35	−11231.29
90 organisms	−54078.01	−54078.18
150 organisms	−39606.31	−39606.60
218 organisms	−134173.86	−134167.56
140 organisms	−124777.22	−124780.10

6.2. DP-ICSILog in a Real-World Application. We integrated DP-ICSILog into RAxML [8], which is a widely used tool for inferring phylogenies (evolutionary trees) from molecular data that has been developed in our group. The vast majority of logarithm invocations is conducted when the log likelihood scores of alternative tree topologies are computed. We found that an LUT with 4,096 entries is sufficient to guarantee numerical stability of RAxML and yield accurate results (see below). Table 1 indicates the respective log likelihood scores for tree searches using the GNU and DP-ICSILog implementations on various DNA datasets with 40, 90, 150, and 218 organisms (sequences) as well as a protein dataset with 140 organisms. Based on standard statistical significance tests for comparing log likelihood scores of phylogenetic trees as implemented in the CONSEL tool suite [25], we found that the score differences among the respective trees are not statistically significant. In other words, the trees computed (under the same starting conditions) using the

GNU implementation and the DP-ICSILog (LUT size: 4,096) cannot be statistically distinguished from each other. Hence, DP-ICSILog with an LUT size of 4,096 provides sufficient application- and domain-specific accuracy for RAxML.

6.3. Accuracy Assessment. Initially, we used the standard C `rand()` function to generate benchmarks with $2 * 10^7$ random numbers in order to measure the average error introduced by the logarithm approximation as a function of LUT size with respect to the GNU function. The results are provided in Table 2. We used the ICSILog software to generate the contents of *man_LUT*, such that it yields *exactly* the same results as ICSILog. From Table 2, we deduce that an LUT with 4,096 entries represents a good tradeoff between accuracy and LUT size for our purposes (developing a hardware architecture for RAxML), since an LUT of this size only requires 3 block rams (36 Kb each). For a medium-size new-generation FPGA, like the Xilinx Virtex 5 SX95T, 3 block rams correspond to only 1% of the total block memory available. As discussed in [9], the size of the LUT increases exponentially for every additional correct bit in the mantissa. Clearly, a specific target application as well as a global view of the entire reconfigurable system that will use the LAU is required to determine the ideal *man_LUT* size. Since the software implementation is available as open-source code, it is easy to assess the required mantissa LUT size a priori, that is, before modifying the reconfigurable architecture. For instance, the overall RAxML hardware architecture requires a huge amount of memory and reconfigurable fabric for other purposes. Therefore, we chose to minimize the hardware resources consumed for the logarithmic function to the largest possible extent.

Finally, for a *man_LUT* with 4,096 entries, we also measured the minimum, maximum, average, and mean squared error between the GNU SP and DP library functions, the respective logarithmic approximation implementations (SP-/DP-ICSILog, DP-LAU), and the SP-/DP-MKL library functions. Table 3 provides these errors for 10^6 random input numbers ranging from 10^{-20} to 10^{20}.

6.4. Performance Assessment versus Hardware. The LAUs and VLAUs were mapped to a Xilinx Virtex 5 SX95T-2 FPGA. In Figure 5, we provide resource usage and performance data for LAU (SP on the left and DP on the right) implementations with different latencies. We tested different latency-specific configuration settings for the Xilinx Floating-Point adders and multipliers that are generated. The variation of these settings allowed us to generate LAU implementations with latencies that range between 5 and 22 clock cycles. Note that, all measurements in this Section refer to LAUs and VLAUs with a *man_LUT* size of 4,096 entries.

The respective clock frequencies of the LAUs were obtained using the Xilinx Tools (ADVANCED 1.53 speed file) and are also provided in Figure 5. The clock frequencies are obtained from the static timing report, and the default Xilinx `Balanced` optimization strategy was selected. All implementations (SP-LAU, DP-LAU, SP-VLAU, and DP-VLAU) are fully pipelined with a throughput of one result per clock cycle and per pipeline datapath. Since the LAU only comprises a single pipeline datapath, a throughput of one result per clock cycle is achieved, while the VLAU (with two independent pipeline datapaths) can compute two results per clock cycle.

Table 2: Average LAU error and *man_LUT* # of block rams.

# Block rams (18 Kb)	LUT entries	Average error
1	512	0.000352
2	1,024	0.000176
3	2,048	0.000088
6	4,096	0.000044
12	8,192	0.000022
24	16,384	0.000011
48	32,768	0.000005

Table 3: Min, max, average, and mean squared error of logarithm implementations with respect to GNU functions.

Program/unit	Min	Max	Avg	MSE
SP-ICSILog	$4.228e-7$	$1.210e-4$	$4.438e-5$	$2.689e-9$
DP-ICSILog	$3.140e-9$	$1.205e-4$	$4.437e-5$	$2.688e-9$
LAU/VLAU	$4.228e-7$	$1.210e-4$	$4.437e-5$	$2.690e-9$
SP-MKL	$0.0e-0$	$3.815e-6$	$5.003e-7$	$1.65e-14$
DP-MKL	$0.0e-0$	$4.44e-16$	$4.52e-22$	$4.93e-38$

In Figure 6, we provide the clock frequencies of the SP-LAU and DP-LAU for *man_LUT* sizes ranging between 512 and 32,768 entries. The frequency reduction with increasing LUT size is due to the additional logic (mostly block rams for the LUT) that is required by the LAU. The number of block rams required increases exponentially for every bit that is added to the mantissa field, which is used as an index for *man_LUT*. The increase of other reconfigurable resources is significantly lower, that is, a LAU with a 32,768 entry *man_LUT* size occupies 700% more 36 Kb block rams than a LAU with a 4,096 entry *man_LUT* size, while only 15% more slices and 9% more slice LUTs are required.

In Table 4, we compare the hardware resources used by our custom LUT-based module and the Xilinx FPO [10] (configured in fixed-to-float mode) for transforming the exponent value into a floating point value. The numbers in parentheses next to the names in the first line of Table 4 represent the latency (number of clock cycles) for alternative configurations. Since the LUT-based approach has a latency of two cycles, we configured the floating point operator to have the same latency and integrated it into the LAU. We also added an 11-bit subtractor, such that the LAU produces correct results. The clock frequency of the LAU using the floating-point operator was 60 MHz lower than for our LUT-based approach. The LUT-based module occupies 1 BRAM (18 Kb) while the FPO solution does not use BRAM memory. For some applications, trading some memory for a substantially higher clock speed is acceptable, since it can yield a higher overall clock frequency and thereby improved overall system performance. When the FPO is configured with the maximum latency of 6 cycles, the LAU is only 5 MHz

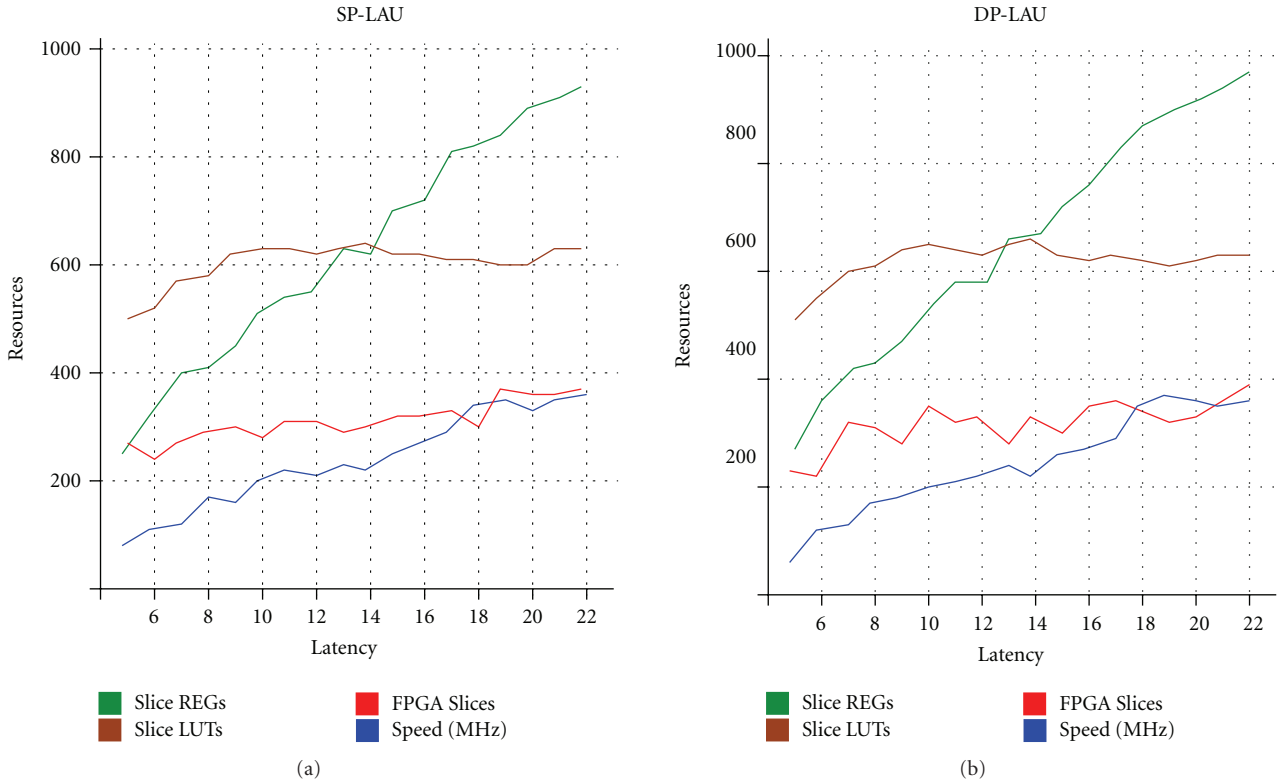


Figure 5: Resources and performance of alternative SP and DP LAU implementations.

Table 4: Resource usage by the LUT-based approach (latency of two cycles) and Xilinx FPO (latency of two and six cycles) for transformation of exponent to SP number.

Resources	LUT(2)	FPO(2)	FPO(6)
Slice registers	32	46	115
Slice LUTs	19	64	88
Occupied slices	20	19	42
# of LUT-flip flop pairs	48	45	126
# of BRAMS (18 Kb)	1	0	0
DP-LAU frequency (MHz)	334	274	339
DP-LAU latency (cycles)	22	22	26

faster than with our LUT-based approach. However, the total latency of the LAU increases by 4 cycles, and the FPO requires a larger amount (see Table 4) of hardware resources.

To the best of our knowledge, the only other open-source logarithm for FPGAs is provided by FloPoCo [12] framework. All FloPoCo operators can be fully parameterized, that is, the user can select the desired precision of the result and define desired performance parameters. To conduct a fair comparison between LAU/VLAU and FPLog units, we used the latest release of FloPoCo (version 2.0.0) and mapped the LAUs/VLAUs and FPLogs to the same FPGA (Virtex 5 SX95T-2). Table 5 provides a performance and resource usage comparison after the implementation process (post-place and route). The reduced FPLog implementations (denoted as (*red.*) in the table) offer the same accuracy as the LAU/VLAU implementations, while the full (precise) FPLog implementations (denoted as (*full*)) yield the same results as the GNU function.

In addition, all available Xilinx optimization strategies were explored to determine the most efficient strategy for each implementation. The available optimization strategies for Virtex 5 devices are `Balanced`, `Area Reduction`, `Minimum`, `Runtime`, `Power Optimization`, and `Timing Performance`. For each implementation/architecture, we only provide the data for the best optimization strategy with respect to clock frequencies in Table 5. The SP-LAU occupies slightly less hardware than the full (high precision) SP-FPLog, while DP-LAU requires significantly less resources than the full DP-FPLog. When the SP/DP FPLogs are configured to yield the same accuracy as the SP/DP LAUs, the FPLog implementations are more resource efficient but exhibit significantly lower maximum clock frequencies. Thus, reduced-precision FPLog units are more likely to lie on the critical path, when embedded into a larger, more complex architecture that needs to calculate logarithms. The VLAUs outperform all other implementations in terms of throughput, since they can produce 2 results per cycle. To achieve performance that is comparable to the VLAU with the FPLog(red.) implementation, two FPLog(red.) instances are required. Therefore, a single VLAU is more resource efficient than two FPLog(red.) units, since the total number of occupied slices and the number of slice LUTs used by the VLAU is smaller than the total amount required for two FPLog(red.) instances. At the same time, the resource

Table 5: Resources, performance, and accuracy of LAUs, VLAUs, and FPLogs.

Resources total	SP				DP			
	LAU	VLAU	FPLog (red.)	FPLog (full)	LAU	VLAU	FPLog (red.)	FPLog (full)
Slice registers 58,800	932	1,864	782	992	970	1,912	809	2,568
Slice LUTs-58,800	551	1,099	712	873	634	1,107	735	1,910
Occupied slices-14,720	298	569	294	330	341	576	307	711
# 36 k block RAM-244	3	3	1	2	3	3	1	2
# 18 k block RAM-488	1	1	1	2	1	1	1	21
# DSP48Es-640	3	6	3	5	3	6	3	14
Frequency (MHz)	370	351	233	240	334	330	233	198
Latency	22	22	18	20	22	22	18	34
Results/cycle	1	2	1	1	1	2	1	1
Error	2^{-17}	2^{-17}	2^{-17}	2^{-23}	2^{-17}	2^{-17}	2^{-17}	2^{-52}

consumption for all other resources, that is, slice registers, brams, and DSPs, is the same. Nevertheless, the single VLAU still outperforms the two FPLog(red.) instances with respect to clock frequency.

As far as the LAU implementation by Chrysos et al. [21] is concerned, two LUTs are deployed, but the LUTs are not initialized as efficiently as in our LAU. Consequently, additional operations, that is, a concatenation, a floating-point multiplication, a float-to-fixed operation, and a fixed-point subtraction, are required for calculating the LUT index. The respective LUT entry is then used to calculate the final output which also represents an approximation of the logarithm function. Since the paper by Chrysos et al. [21] focuses on the overall architecture for Glimmer, only a limited amount of information is provided with respect to implementation and performance of the logarithm unit.

Finally, the configurations presented in [17] by Tropea were mapped to a Virtex 4 LX15-12 FPGA by Tropea. The highest clock frequency reported in [17] is 191 MHz (the architecture has not been made available by the author). Thus, to conduct a fair comparison, we also mapped the LAU to a Virtex4 LX15-12 FPGA. We obtained a clock frequency of 345 MHz for the SP version and of 344 MHz for the DP version.

6.5. Performance Assessment versus Software. We also compared LAU and VLAU performance to a wide range of software implementations: the SP-/DP-GNU logarithms: `logf()/log()`, the SP-/DP-MKL logarithms: `vsLn()/vdLn()`, and the SP-/DP-ICSILog algorithms. As hardware platform, we used a V5SX95T-2 FPGA (speed grade −2) with one arithmetic component, that is, only one LAU and only one VLAU were instantiated, respectively. The software implementations were executed on an Intel Core2 Duo T9600 processor running at 2.8 GHz with 6 MB of L2 Cache. All software (SP-/DP-ICSILog) and hardware implementations (SP-/DP-LAU) we tested used a mantissa LUT with 4,096 entries.

Table 6: Execution times (in ms) of GNU, ICSILog, LAU, and VLAU SP implementations for 10^3 up to 10^8 invocations.

# samples	GNU	ICSILog	LAU	VLAU
10^3	0.03290	0.00620	0.0027	0.0015
10^6	32.40	6.31	2.7	1.42
10^8	3315	595	270.2	142.45

For software tests, we used the GNU `gcc` compiler (version 4.3.2) as well as the Intel `icc` compiler (version 11.1) in order to fully exploit the capabilities of the Intel CPU. We only used `-O1` for optimization with gcc because with more aggressive optimizations (`-O2` and `-O3`) the current SP-ICSILog version yields an average error that is 10^5 times larger than the error obtained by compiling the code with `-O1`. Thus, the aggressive gcc compiler optimizations applied under `-O2` and `-O3` yield numerically unstable code. When `icc` is used, SP-ICSILog produces the expected average error, which is in the range of 10^{-5} for all optimization levels (`-O1`, `-O2`, `-O3`). When `-O2` or `-O3` is used with `icc`, SP-ICSILog is only 1.09 times faster on average than the GNU math library. However, when `-O1` is used, SP-ICSILog is on average 4.5 times faster.

Initially, we used the GNU `gcc` compiler (version 4.3.2, with `-O1`) and measured the execution times for 10^3 up to 10^8 invocations of the GNU library SP function as well as SP-ICSILog. Note that we used the most recent version of the SP-ICSILog algorithm, which is faster than the initial release of the ICSILog software. According to the benchmark that is made available by the authors, the current version is approximately 1.7 times faster than the initial version (when compiled with `gcc` and `-O1`). Table 6 shows the execution

Table 7: Execution times (in ms) of GNU, ICSILog, LAU, and VLAU DP implementations for 10^3 to 10^8 invocations.

# samples	GNU	ICSILog	LAU	VLAU
10^3	0.0722	0.0119	0.003	0.0016
10^6	58.40	9.47	2.99	1.51
10^8	5909	899	299.4	151.5

Table 8: Execution times (in ms) of MKL, ICSILog, LAU, and VLAU SP implementations (The `icc` compiler is used.)

# samples	MKL	ICSILog	LAU	VLAU
10^6	4.7	5.3	2.7	1.42
10^7	46.9	50.2	27.0	14.25
10^8	342.9	486.6	270.2	142.45

Table 9: Execution times (in ms) of MKL, ICSILog, LAU, and VLAU DP implementations (The `icc` compiler is used.)

# samples	MKL	ICSILog	LAU	VLAU
10^6	8.0	8.8	2.99	1.51
10^7	77.2	85.1	29.94	15.15
10^8	668.4	839.7	299.4	151.5

times for the GNU implementation, SP-ICSILog, the SP-LAU, and the SP-VLAU. The SP LAU is 12 times faster than the GNU function and 2.2 times faster than SP-ICSILog, while the SP-VLAU is 23 times faster than the GNU functions and 4.1 times faster than SP-ICSILog.

As already mentioned, the standard release of ICSILog only provides an SP logarithm function. Furthermore, it does not provide built-in error detection/correction for special-case inputs like *nan, inf, −inf* , or negative numbers which is critical for applications like RAxML. In order to conduct a fair performance evaluation of the DP-LAU, we therefore reimplemented the ICSILog algorithm to support DP inputs and invalid input detection. Our new DP version of ICSILog (DP-ICSILog) is only 1.5 times slower than the official SP release by Vinyals and Friedland. DP-ICSILog is also freely available for download together with the LAU/VLAU architectures.

For assessing DP performance, we used `gcc` (`−O1`) and measured execution times for 10^3 up to 10^8 invocations of the GNU, DP-ICSILog, DP-LAU, and DP-VLAU logarithm functions (Table 7). The DP-LAU is 20 times faster than the GNU math library and 3.1 times faster than DP-ICSILog which in turn is up to 6.5 times faster than the GNU implementation. The DP-VLAU is 40 times faster than the GNU math library implementation and 6 times faster than DP-ICSILog.

For our second set of experiments, we used the Intel `icc` compiler (version 11.1, optimization flag `−O1`). We also tested the fast logarithm implementation provided by the Intel Math Kernel Library (MKL [23]) for 106 to 10^8 invocations on random input numbers as in the preceding experiments.

Tables 8 and 9 provide the execution times for the SP and DP MKL, ICSILog, and LAU implementations, respectively.

Table 10: Execution times (in ms) of GNU and ICSILog DP implementations compiled with `gcc` and `−O2`/`−O3`.

# samples	DP-GNU		DP-ICSILog	
	`−O2`	`−O3`	`−O2`	`−O3`
10^3	0.0779	0.0758	0.0119	0.0119
10^6	57.82	57.34	8.50	8.42
10^8	5,692	5,678	799	798

Table 11: Execution times (in ms) of MKL and ICSILog DP implementations compiled with `icc` and `−O2`/`−O3`.

# samples	DP-MKL		DP-ICSILog	
	`−O2`	`−O3`	`−O2`	`−O3`
10^6	8.0	8.0	8.1	8.1
10^7	64.1	60.9	77.9	77.7
10^8	619.6	601.8	769.8	769.6

The SP-LAU is 1.7 times faster than the MKL logarithm and 1.8 times faster than SP-ICSILog, while the respective speedups for the SP-VLAU are 3.3 and 3.5. Unfortunately, a detailed description of the MKL logarithm implementation is currently not available. The DP-LAU is 2.6 times faster than the respective MKL implementation and 2.8 times faster than DP-ICSILog which is almost as fast as the DP-MKL function (speedups vary between 0.8 and 0.9). The DP-VLAU is 5.2 times faster than the MKL implementation and 5.6 times faster than DP-ICSILog.

As already mentioned, SP-ICSILog becomes unstable when optimization flags `−O2` or `−O3` are used with `gcc`. Therefore, we only assessed the performance impact of using `−O2` and `−O3` with `gcc` on DP-ICSILog. We compare DP-ICSILog execution times with all alternative DP implementations: DP-GNU, DP-MKL, and DP-LAU. Table 10 provides the execution times for DP-GNU and DP-ICSILog for 10^3 to 10^8 invocations of the `gcc`-compiled code. The DP-LAU is 19 times faster than the GNU math library and 2.7 times faster than DP-ICSILog, which in turn is up to 7 times faster than the GNU implementation (for `−O2` as well as `−O3`). The DP-VLAU is 37.5 times faster than the GNU math library and 5.3 times faster than DP-ICSILog. Table 11 provides respective execution times under DP for the same experimental setup, but using the Intel icc compiler instead. The DP-LAU is 2.2 times faster than the respective MKL implementation and 2.5 times faster than DP-ICSILog which is as fast as the DP-MKL function. Speedups between DP-ICSILog and the DP-MKL function vary between 0.83 and 0.98 for both optimization levels `−O2` and `−O3`. Finally, the DP-VLAU is 4 times faster than the MKL function and 5 times faster than DP-ICSILog.

7. Conclusion and Future Work

We presented an architecture that efficiently calculates an approximation of the logarithm in reconfigurable logic under SP and DP arithmetics and only uses 2% of the computational resources on medium-size FPGAs. The SP-/DP-LAUs (LUT size: 4,096) as well as the DP software are freely available for download. To the best of our knowledge,

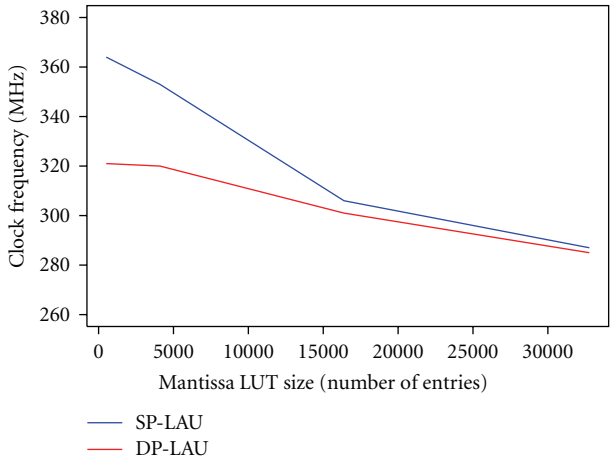


Figure 6: LAU frequencies with respect to LUT size.

this represents the only IEEE-754 compatible open-source implementation of a resource-efficient logarithm approximation unit in reconfigurable logic. Since the accuracy demands of such a basic unit strongly depend on the target application, we also make available several COE files that can be used to initialize LUTs of various sizes and hence easily adapt the LAUs to the desired accuracy level. Except for an increase of block ram usage to hold the mantissa LUT, the proportion of required hardware resources will only slightly increase (see Section 6.4), if the LUT size is increased and the speed will only slightly decrease (see Figure 6).

Finally, we designed a memory-efficient VLAU architecture that exploits the dual-port option of embedded memory blocks. The VLAU utilizes two pipelined LAU datapaths but only requires one instance of the read-only lookup tables. This feature allows a VLAU to calculate two results per cycle while requiring half the LUT memory than two independent parallel LAUs. The VLAU can therefore be used for designing large architectures that require the computation of logarithms on vectors. The SP/DP-VLAUs are freely available for download.

Acknowledgment

Part of this work was funded under the auspices of the Emmy-Noether program by the German Science Foundation (DFG).

References

[1] D. Ververidis and C. Kotropoulos, "Gaussian mixture modeling by exploiting the Mahalanobis distance," *IEEE Transactions on Signal Processing*, vol. 56, no. 7 I, pp. 2797–2811, 2008.

[2] J. Felsenstein, "Evolutionary trees from DNA sequences: a maximum likelihood approach," *Journal of Molecular Evolution*, vol. 17, no. 6, pp. 368–376, 1981.

[3] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "Exploring FPGAs for accelerating the phylogenetic likelihood function," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, pp. 1–8, Rome, Italy, May 2009.

[4] N. Alachiotis, A. Stamatakis, E. Sotiriades, and A. Dollas, "A reconfigurable architecture for the phylogenetic likelihood function," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 674–678, Prague, Czech Republic, September 2009.

[5] V. M. Preciado, *Real-Time Wavelent Transform for Image Processing on the Cellular Neural Network Universal Machine*, vol. 2085 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, 2001.

[6] B. De Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks, "FPGA accelerator for real-time skin segmentation," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTIMEDIA '06)*, pp. 93–97, October 2006.

[7] N. Alachiotis and A. Stamatakis, "Efficient floating-point logarithm unit for FPGAS," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, Atlanta, Ga, USA, April 2010.

[8] A. Stamatakis, "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinformatics*, vol. 22, no. 21, pp. 2688–2690, 2006.

[9] O. Vinyals and G. Friedland, "A hardware-independent fast logarithm approximation with adjustable accuracy," in *Proceedings of the 10th IEEE International Symposium on Multimedia (ISM '08)*, pp. 61–65, December 2008.

[10] Xilinx, "Floating Point Operator v4.0," July 2009, http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.

[11] L. de Soras, "Fast log() Function," July 2009, http://www.flipcode.com/cgi-bin/fcarticles.cgi?show=63828.

[12] F. De Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications(FPL '09)*, pp. 59–64, September 2009.

[13] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithmand exponential functions for FPGAs," *Microprocessors and Microsystems, Special Issue on FPGA-Based Reconfigurable Computing*, vol. 31, no. 8, pp. 537–545, 2007.

[14] J. Detrey and F. De Dinechin, "A parameterizable floating-point logarithm operator for FPGAs," in *Proceedings of the 39th Asilomar Conference on Signals, Systems and Computers*, vol. 2005, pp. 1186–1190, IEEE Signal Processing Society, November 2005.

[15] National Instruments, "High Throughput Natural Logarithm Function," http://www.ni.com/.

[16] J. E. Volder, "The CORDIC trigonometric computing technique," in *Proceedings of IRE Transactions on Electronic Computers*, pp. 330–334, 1959.

[17] S. E. Tropea, "FPGA implementation of base-N logarithm," in *Proceedings of the 3rd Southern Conference on Programmable Logic (SPL '07)*, pp. 27–32, February 2007.

[18] Xilinx, http://www.xilinx.com/.

[19] Actel, January 2009, http://www.actel.com/.

[20] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, 2000.

[21] G. Chrysos, E. Sotiriades, I. Papaefstathiou, and A. Dollas, "A FPGA based coprocessor for gene finding using Interpolated Markov Model (IMM)," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 683–686, August 2009.

[22] J. J. Raygoza-Panduro, S. Ortega-Cisneros, J. Rivera, and A. de la Mora, "Design of a mathematical unit in FPGA for

the implementation of the control of a magnetic levitation system," *International Journal of Reconfigurable Computing*, vol. 2008, Article ID 634306, p. 9, 2008.

[23] Intel, "Intel Math Kernel LibraryReference Manual," http://software.intel.com/en-us/articles/intel-mkl/.

[24] N. Alachiotis, S. A. Berger, and A. Stamatakis, "Efficient PCF-PGA communication over Gigabit Ethernet," in *Proceedings of the International Conferences on Embedded Software and Systems (ICESS '10)*, pp. 1727–1734, Bradford, UK, 2010.

[25] H. Shimodaira and M. Hasegawa, "CONSEL: for assessing the confidence of phylogenetic tree selection," *Bioinformatics*, vol. 17, no. 12, pp. 1246–1247, 2002.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 963539, 12 pages
doi:10.1155/2011/963539

Research Article

A Streaming High-Throughput Linear Sorter System with Contention Buffering

Jorge Ortiz[1] and David Andrews[2]

[1] *Information and Telecommunication Technology Center, The University of Kansas, 2335 Irving Hill Road, Lawrence, KS 66045, USA*
[2] *Computer Science and Computer Engineering, The University of Arkansas, 504 J. B. Hunt Building, Fayetteville, AR 72701, USA*

Correspondence should be addressed to Jorge Ortiz, jorgeo@ku.edu

Received 28 July 2010; Accepted 15 January 2011

Academic Editor: Aravind Dasu

Popular sorting algorithms do not translate well into hardware implementations. Instead, hardware-based solutions like sorting networks, systolic sorters, and linear sorters exploit parallelism to increase sorting efficiency. Linear sorters, built from identical nodes with simple control, have less area and latency than sorting networks, but they are limited in their throughput. We present a system composed of multiple linear sorters acting in parallel to increase overall throughput. Interleaving is used to increase bandwidth and allow sorting of multiple values per clock cycle, and the amount of interleaving and depth of the linear sorters can be adapted to suit specific applications. Contention for available linear sorters in the system is solved through the use of buffers that accumulate conflicting requests, dispatching them in bulk to reduce latency penalties. Implementation of this system into a field programmable gate array (FPGA) results in a speedup of 68 compared to a MicroBlaze processor running quicksort.

1. Introduction

Sorting is an essential function for many scientific and data processing applications. Extensive research has optimized multiple software sorting algorithms for general-purpose computing, thereby increasing application performance. The need for higher performance has also motivated the migration of sorting algorithms into specialized hardware to exploit spatial parallelism. Nonetheless, many of the assumptions made to increase performance on a general-purpose processor do not hold for custom hardware implementations. Thus, reconfigurable applications typically do not enjoy the benefits of software-based sorting algorithms. When directly translated into hardware, software algorithms can quickly degrade into a series of data retrievals, comparisons, swaps, and writes; all problems that can be magnified in systems with low processor speeds, limited storage, disabled caches, and high-latency memory access times.

A conventional sorting system involves data acquisition and collection, processing, dynamic and long-term storage, and sorting and dispatch. Many hardware approaches use linear sorting to keep a sorted list with in-order insertions but fail to optimize throughput, the rate at which data elements are processed. The system's sorting throughput is limited by the weakest link, which in many cases is the sorting stage. Even though parallelism can speed up hardware-based sorting, sorted results are generally attained only one at a time. Thus, a hardware-based linear sorter would only achieve a maximum throughput of one sorted output per clock cycle, regardless of the system's available and exploitable parallelism. Additionally, specialized hardware like priority schedulers might have heterogeneous data processing and arrival times, which need to be considered for the system to work in a pipelined fashion. Pipelined systems are not able to start the next stage before completing the current one, so stalling on a stage potentially halts the pipeline's progress until data becomes ready. This property essentially invalidates other hardware sorting approaches like Bitonic sorting networks [1, 2], which are able to achieve high-throughput only when acting on fully available

data sets. To achieve low-latency pipelining and increase throughput through parallelism in a sorting system, a novel sorter implementation is needed.

In this paper, we present an alternative approach for linear sorters that solves the previously identified problems by

(i) expanding the linear sorter implementation and making it versatile, reconfigurable and better suited for streaming input and output (Section 3),

(ii) parallelizing the linear sorter for increased throughput for scenarios with random and uniform distribution of streaming data (Section 4),

(iii) minimizing latency costs by buffering of contending parallel sorting requests (Section 5),

(iv) implementing the high-throughput linear sorter and outmatching the performance of current linear sorter approaches (Section 6).

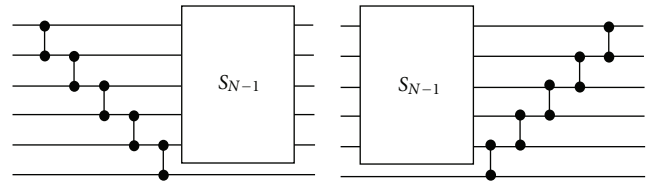
Additional features have been added from our work in [3] to further reduce sorting latency using a contention buffering scheme. Moreover, we provide a detailed explanation of our interleaved linear sorter system implemented as a component of a superscalar reconfigurable processor core. The innovative features of our interleaved approach fill the need for a high-throughput pipelined sorting unit, whose adaptability makes it particularly useful for streaming and reconfigurable systems.

2. Background

Several popular sorting algorithms (e.g., quicksort, mergesort, and heapsort) use divide-and-conquer techniques to achieve efficiency [4]. Intuitively, one would assume they are suitable for a parallel hardware implementation. Regrettably, upon breakdown to a register-transfer level representation, these algorithms are plagued with data movement, synchronization, bookkeeping, and memory access overhead. The sorting speed is highly dependent on a fast and robust computing platform, the type of platform that is inadequate for mobile, embedded, real-time, low-power, or reconfigurable systems.

Hardware sorting makes extensive use of concurrent data comparisons and swaps each clock cycle, rather than relying on the sequential execution of multiple assembly operations like its software counterpart. Due to its parallelism, a hardware implementation can speed up sorting applications, even at lower clock frequencies. Hardware comparisons can occur simultaneously on multiple pairs of elements. A first approach involves making multiple concurrent small operations in comparators that cascade into a well-structured network and is called a sorting network. Inserting serial input to a systolic array of sorter cells provides a second approach to hardware sorting. The third and final approach involves a single large parallel computation over multiple independent nodes and is called a linear sorter.

2.1. Sorting Networks. Sorting networks use parallel wires and multiple levels of swap comparators to shuffle data into a sorted output array of wires. Each swap comparator consists of two data inputs A and B and their two outputs H and L, where $H = \max(A, B)$ and $L = \min(A, B)$. This component can be used recursively to construct a sorting network S_N, capable of sorting N data inputs. Simple implementations for sorting networks use the same principles as well-known sort algorithms. Figures 1(a) and 1(b) show two common implementations, where wires are represented by horizontal and swap comparators by vertical lines. Figure 1(a) resembles a bubble sort: swap-comparators first sink the lowest value to the bottom wire and then operate on the remaining S_{N-1} network recursively. Figure 1(b) acts like an insertion sort, assuming a sorted network S_{N-1} already exists and then using swap comparators to position the remaining input accordingly. While simple, both sorting networks suffer from latency with the worst case scenario having $\mathcal{O}(N)$.



(a) Bubble (b) Insertion

Figure 1: Sorting networks of size N.

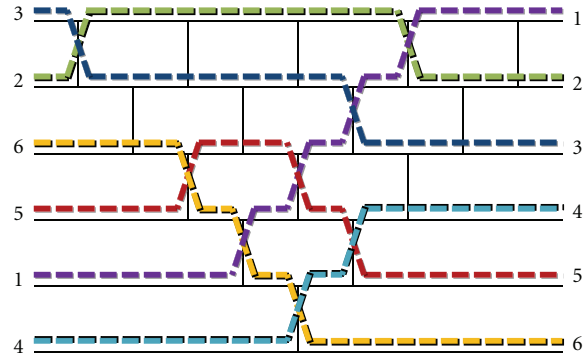


Figure 2: Parallel sorting network of size 6.

The recursive structure of the bubble and insertion networks is identical but inverted. Collapsing them to allow parallel comparisons results in an identical sorting structure. Figure 2 displays this resulting structure sorting a sample set of data of size 6 with 15 comparators. Sorting latency is reduced through parallel comparisons occurring simultaneously on multiple pairs of wires. As the size of the network grows, the number of comparators increases in a quadratic manner. In general, for a network of size N, the amount of comparators required will be $N \times (N - 1)/2$.

Merging networks are more efficient than insertion or bubble networks, which are impractical due to their large depth. These networks keep two lists of ordered data and produce a single ordered list after merging. Batcher was the first to propose sorting networks, by introducing the odd-even mergesort and bitonic sort networks [1]. The odd-even

mergesort network sorts all odd and even data in separate lists before merging, while bitonic networks first sort values in monotonically increasing and decreasing lists and then merge the lists into a single sorted sequence. These types of sorting networks are common in network crossbars for asynchronous transfer mode (ATM) switching, usually in the form of a Batcher-Banyan switch [5–7].

The main advantage of sorting networks is their ability to receive a parallel input block of data, rather than feeding input data serially. However, the detriments include large numbers of processing elements (PE), high latency, and the need to resort a full set of data upon a single new insertion. Martínez et al. reduced sorting network latency by introducing levels of pipelining [8], while Tabrizi and Bagherzadeh introduced a tree-like structure to reduce the area and PE complexity [9]. Nevertheless, sorting networks in general cannot efficiently handle progressive incoming data, and their large area and complexity may hinder implementation [10].

2.2. Systolic Sorters. Systolic arrays, which are a matrix arrangement of processing cells, can be structured to sort serial input. This approach was first proposed by Leiserson [11] in the form of a systolic priority queue. Queue data is fed serially into the systolic system. It traverses the length of the priority queue going forward then traverses it again in reverse direction towards the output. When two data elements, traversing in opposite directions, meet in a queue cell, the one with the maximum value keeps traversing in reverse (towards the output), while the minimum is sent back to the forward path. A special case of the systolic priority queue occurs if all inputs are loaded before starting queue extraction. The priority queue then becomes a systolic sorter [12], but the system requires a systolic array length equal to the number of all input elements to be sorted.

Each cell in the systolic sorter becomes a processing element with two inputs and two outputs. Figure 3 shows this basic cell structure, with `Input_F` and `Output_F` aligned with the forward direction and `Input_R` and `Output_R` with the reverse path. On collisions where the two inputs have valid data, `Output_R` = max(`Input_F`, `Input_R`) and `Output_F` = min(`Input_F`, `Input_R`), which consequently advances the maximum values towards the reverse path culminating on the priority queue's output. The cell's regular structure is an important consideration for VLSI systolic arrays. Without it, this cell is essentially a registered swap comparator.

The structure of the systolic sorter is a linear array of processing cells. Figure 4 demonstrates the regular structure of a priority queue with five cells, with the forward path heading right, the reverse path pointing left, and the rightmost cell reversing the traversal paths. Registered outputs, rather than combinatorial logic, are a key factor of the cell and the priority queue functionality. Serial data is sent to the systolic sorter's input every other clock cycle, allowing comparisons between adjacent forward and reverse queue values, which would otherwise bypass each other.

The extra clock cycle needed per input for a systolic priority queue of length N decrees the sorting latency

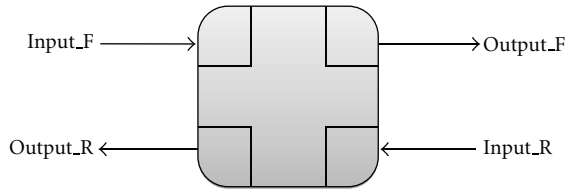


Figure 3: Systolic sorter cell interface.

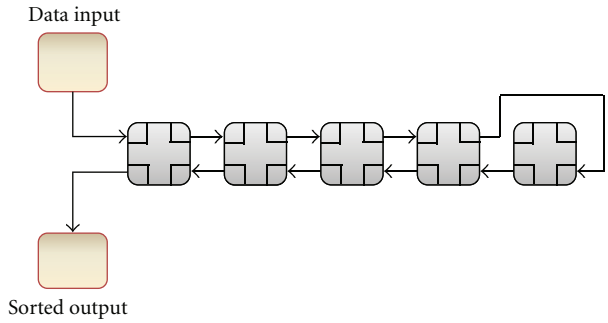


Figure 4: Systolic sorter array structure.

accordingly. $2N - 1$ cycles are needed to feed input data and output the first sorted value, then another $2(N - 1)$ cycles are needed to flush the remaining $N - 1$ sorted values into the output, for a total latency of $4N - 3$ clock cycles. The systolic sorter depth then is just N cells, an improvement over a sorting network's $N \times (N-1)/2$ required swap comparators, at the expense of increased sorting latency.

The size and latency requirements for sorting networks and systolic sorters are an impediment for implementation. Large data sets demand $\mathcal{O}(N^2)$ swap comparators in sorting networks and $\mathcal{O}(N)$ cells in systolic sorters, putting a strain on the system's area requirements. However, systolic sorters can be implemented as components of sorting architectures than can handle large or infinite streaming data sets and provide constant-time sorting to fixed-size sorting systems for arbitrary size data [13]. Furthermore, they can be employed to manipulate cost-performance tradeoffs in hybrid sorting systems that decompose sorting into sequential and parallel parts [14].

Modified systolic sorters that support large data sets nonetheless suffer from large sorting latencies, since incoming values must still traverse the length of the systolic sorter twice. While appropriate for offline sorting, these large latencies create obstacles for streaming data, an encumbrance that is compounded with the extra multicycle input rate of systolic sorters. By contrast, linear sorters, the third approach to hardware sorting, allow for single-clock insertion and single-clock sorting latency.

2.3. Linear Sorters. Linear sorters, which keep a sorted list while inserting new elements in-order, are an alternative approach in hardware-based sorting. The principle is the same as insertion sort in a software-based linked list: new inserted data is positioned at the corresponding place in the sorted list, thus keeping the list sorted. Although in

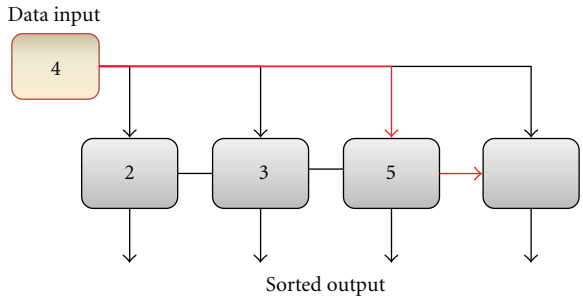


Figure 5: Linear sorter node configuration.

hardware one can iteratively traverse every node in the list when inserting values [15], a more appropriate method is to send incoming values to all nodes in parallel. Near-neighbor interconnections exist among nodes and their left and right neighbors, which allow each sorting node to make an autonomous decision about its new value and positioning in the list. This decision process involves each node acquiring a new value by comparing the incoming value against the node's current value and that of its neighbors. Although the underlying hardware structure of the linear sorter is unchanged, each node and its corresponding value can be thought of as shifting right, shifting left or holding its current value in the context of the sorted list. Figure 5 shows a linear sorter and the interconnections between its nodes. The data input (with a value of 4) is propagated to all nodes, forcing the third one (with a value of 5) to "shift" its value to the right, allowing the reception of the newly inserted value on the sorted list.

Linear sorters have small logic and control footprints, regular structures, and relatively straightforward hardware implementations using shift registers. Additionally, these sorters are particularly appropriate for streaming data, where low sorting latency and continual sorting is crucial. The main drawback of linear sorters is the serial nature of both their inputs and outputs. Even though the output of a linear sorter can be accessed in parallel (as depicted in Figure 5), at most, one value can be erased from the queue during continuous operation of streaming data. The fixed size of a linear sorter adds extra restrictions as it limits the size of growing lists before depleting node availability. When the sorter is full, new inserted values must replace old ones. This can be done using a FIFO that outputs the Top N results of a list [16]. Alternative approaches augments the incoming data with an associated tag that indicates either an insertion or a deletion [17]. Such a linear sorter must additionally allow nodes to shift values left in addition to shifting right, holding, and inserting. Furthermore, it is also possible to sort on tags rather than on the data itself [18]. This approach is useful when implementing priority schedulers, or for preserving the order of data with identical tags. A final approach is to use a linear merge sorter, where two FIFO sorted queues are merged into a single sorted queue through linear sorters [19]. Regardless of the approach, with only a serial output, linear sorters are confined to an output rate of one value per clock cycle, limiting the overall throughput of the system.

Both sorting networks and linear sorters can capitalize on increased throughput to improve their performance. Sorting networks can be pipelined to increase sorting throughput but at a high area cost due to their depth. Regardless, sorting networks still suffer from a latency of $\mathcal{O}(\log^2 N)$ and are still unable to handle data streams [20]. Linear sorters, on the other hand, have a single clock cycle of latency and reduced area but by default are unable to produce increased throughput. We propose an extension to linear sorters which effectively increases their throughput by using parallel linear sorters and interleaving logic.

3. Configurable Linear Sorter

The core of our high-throughput sorting system is based on a linear sorter. Each insertion is broadcast to all nodes, where each node either holds its value, receives the input value, or shifts right. Reconfiguration was a key property to provide versatility and adaptability; hence, the size of the data D, the depth of the linear sorter N, and the sorting direction are configurable parameters of the sorting system. We further extend the linear sorter with tags, so that sorting is performed on the tags rather than the data, as in [18]. Because the tags also have a variable bit length, an application can specify both the size of the data and the tag. The benefits are twofold: the linear sorter minimizes area consumption, while adapting to the application's specific requirements and the logic delay for the comparison operation is reduced for tags smaller than the data width. In case the sorting must depend on the data itself, the value of the tag can be replaced with the data.

3.1. Linear Sorter Functionality. Due to the finite nature of our linear sorter depth N, by default, only the Top N results are kept. If sorting greater than N values is required, the replacement policy can be augmented with external logic that checks for full conditions. When the sorter has only one free node available and an insertion occurs, an output signal bit is set. This enables the external input feeding logic to start buffering rather than discarding new input data.

The output of the linear sorter can be accessed in parallel by retrieving multiple node values at once. This is useful for systems which process data in batches, since the linear sorter depth N can be set to match the batch size. Once an input batch is received and processed, the linear sorter can be reset to start the process again for a new input batch. For continuous operation, the output must be serial, the top value must be erased, and the queue must be informed of this action. To accomplish these properties, an additional external signal makes a request to delete the top value while retrieving it, thus freeing up nodes. This operation is akin to the `pop()` operation used in standard FIFO queues and stacks.

Because the leftmost value is deleted, the node's functionality needs to be enhanced to include a left shift, in addition to the default operations shift right, hold, and insert. Figure 6 shows the added functionality for each node of the linear sorter while sorting nine values, with the vertical

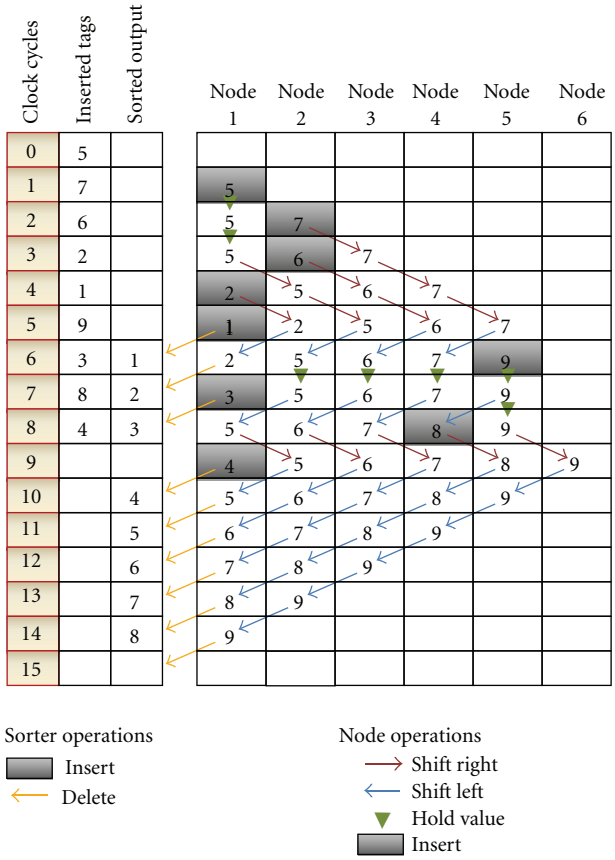


Figure 6: Sample execution of modified linear sorter.

axis indicating clock cycles. During each clock cycle, a subset of the active nodes either shift left, shift right, hold, or insert.

The output logic for this sample linear sorter deletes contiguous tags (and their associated data). Once the first tag is received, the output logic will keep deleting as it retrieves data with incrementally contiguous tags. When it stops deleting, the linear sorter continues to receive input, and deletion resumes when the next contiguous tag is in the top of the queue. The gap in the sorted output column of Figure 6 indicates a pause in delete requests from the output logic. The worst case scenario, where the first tag is inserted last, does not affect the sorting rate, but it does increase the latency of results.

To achieve continuous processing of streaming input, the external delete signal is kept at a constant high after accumulating enough input data. This setup creates a priority queue, where the local maxima is always deleted from the top of the queue. However, an external delete signal adds the advantage of testing a tag or value before deletion. Consequently, the linear sorter waits until certain conditions are met rather than just deleting the current top value (which might have been superseded in subsequent insertions). The example in Figure 6 shows an inactive deletion during clock cycle 9, to ensure a predetermined order. Had the deletion been constant, the next top tag value would have been $\{5\}$ instead of $\{4\}$, producing the output $\{1, 2, 3, 5, 4, 6, 7, 8, 9\}$. This feature is important if either

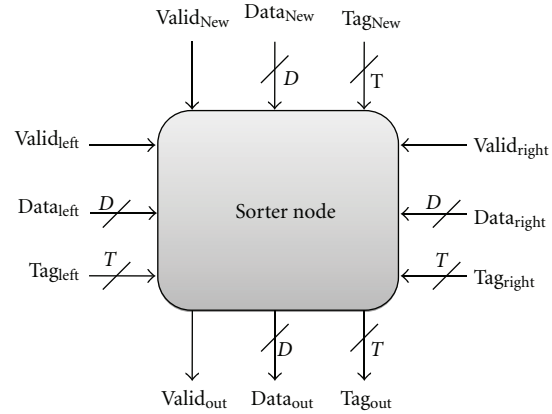


Figure 7: Sorter node interface. D = Data Width, T = Tag Width.

continuity is necessary (like pixels on an image) and/or data values cannot be discarded. A sample situation is processing instructions by integer, floating point, and other functional units with different latencies back into program order using linear sorters. In general, ordered data on any heterogeneous parallel processing system that needs to be reordered must have its values wait until previous sorted data has been dispatched. To ensure this, tags can be initially assigned incremental values before entering out-of-order execution stages.

Careful consideration must be used with the external delete signal to avoid deadlock problems. The linear sorter should be configured with a depth N large enough to allow a sliding window for contiguous values. If, for example, the system can guarantee that no two contiguous data values can arrive more than M clock cycles away, then the minimum value of N should be $M + 1$. This also specifies the maximum latency of sorted results.

3.2. Sorter Node Architecture. To achieve the linear sorter functionality we described, each sorting node implements the appropriate interface and logic. The sorter node interface has three sets of inputs and a set of outputs, each set containing signals for validity, tag, and data, as shown in Figure 7. The widths of the data (D) and tags (T) can be specified as a parameter during synthesis. The three input sets come from the left neighbor, the right neighbor, and the linear sorter insertions being forwarded to all nodes. The node's output set is sent to both its left and right neighbors, so that they too can make sorting decisions autonomously. This architecture coincides with those of single linear sorters [16, 20], which stores (key, data) pairs and compares them against neighbors and incoming values.

The decision logic to shift, insert, or hold a node value is a function of these three sets of inputs, specifically, the tags and validity bits. Shifting decisions are summarized on Table 1. Isolated insertions (row 1) right-shift all tags smaller than the new tag. An isolated deletion (row 2) always left-shifts every node. A left-shift can also occur while both inserting and deleting if the new tag is greater than the node's right neighbor (row 3); otherwise, the node inserts the new tag

if that tag is also greater than its own (row 4). For isolated insertions, nodes will also insert the new tag if it falls between its left neighbor and its own tag (row 5) or if the new tag is added at the end of the queue (row 6). Otherwise, the nodes keep their current tag and data values.

The sorter node has registers of length D, T, and 1 for the data, tag and validity bit, respectively. Three less than comparators test the inserted tag value against both neighbors' and the node's tags. Finally, additional Boolean gates are used to assess the input validity and the operations being requested.

4. Interleaved Linear Sorter System

The throughput of a linear sorting system is limited to one input per clock cycle and can be improved if parallel inputs are used. Multiple linear sorters are configured in parallel to create an interleaved linear sorter (ILS) to overcome this throughput limitation. An ILS system distributes parallel inputs to the linear sorters, trying to minimize stalling from full conditions through even distribution. Assuming random or incrementally contiguous tag numbers, we can use interleaving to equally distribute the values among all linear sorters. Typically, this is done by setting the ILS width, W (the number of linear sorters), to a power of two, which reduces the interleaving modulo arithmetic to simple bit trimming. Despite the average even distribution of this method, there is the distinct possibility that two or more tags will match the same sorter, which requires preemption and buffering. All inputs are then serviced by the hardware in round-robin fashion, without keeping state of which inputs were serviced in the previous clock cycle. When contention for a linear sorter occurs, only the first conflicting request is serviced. The interleaving controller sets an output signal to indicate this input stalling, and the data stream can resume after the buffer, using registers common in pipelined systems.

Figure 8 shows a sample execution of an ILS. The ILS width $W = 4$, indicates the quantity of linear sorters (LS) acting in parallel. The numbering i for the ILS's four linear sorters, LS0, LS1, LS2, and LS3, indicates its interleaving value. Hence, a tag value T will be sent to linear sorter i if $T \bmod W = i$. During the first clock cycle of this execution, each of the parallel inputs tags (and their corresponding data values) are dispatched to different linear sorters. During the second clock cycle, only the first of two clashing tags $\{5, 1\}$ is serviced. The remaining one $\{1\}$ is dispatched in the next clock cycle, while the input is stalled. Even though the same situation happens again during clock cycle 5, all 16 values are sorted after only 6 clock cycles and are then ready to be sent to the outputs at a parallel rate of W. In this particular example, the outputs are ready to start streaming in contiguous order as early as the fifth clock cycle.

The ILS registers the values for multiple conflicting tags and dispatches them over several clock cycles. This solves possible situations in which three or more tags match the same LS. For instance, an input of $\{0, 4, 8, 12\}$ would cause this situation for the example in Figure 8. During these worst case scenarios, the throughput advantages of an ILS degrade gracefully to that of a single linear sorter system. Nevertheless, additional area is required to implement W linear sorters and their input logic for the benefit of the average and best case scenarios. In these cases, the ILS shows a distinct advantage in throughput over a typical linear sorter system.

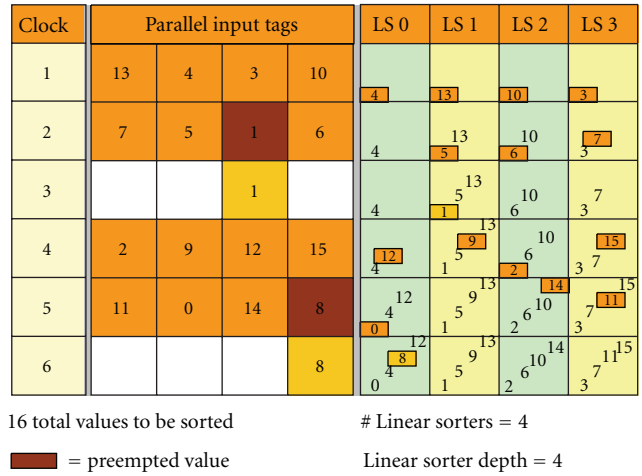


Figure 8: Sample sorting of 16 values by 4 linear sorters.

Table 1: Logic for sorter node's shift operation.

	Operation	Delete	Insert	NewTag less than		
				LeftTag	MyTag	RightTag
1	Right Shift	No	Yes	Yes		
2	Left Shift	Yes	No			
3	Left Shift	Yes	Yes			No
4	New Value	Yes	Yes	No	No	Yes
5	New Value	No	Yes	No	Yes	
6	New Value	No	Yes	Yes	N/A	

4.1. Input Distribution and Latency. The average latency for sorting W values arriving in parallel will vary predictably if we assume a uniformly distributed set of tags. With $W = 1$, there are no conflicts, and the single tag is always processed in one clock cycle. When $W = 2$, there is a 50% chance that the second tag's interleaving value will be the same as the first one. This condition adds an extra clock cycle of latency 50% of the time, therefore giving an overall average of 1.5 clock cycles for $W = 2$. For larger values of W, we used Monte Carlo simulations rather than relying on deterministic algorithms, due to the increasingly complex dependencies with previous tag's interleaving values. Table 2 shows the results of 2^{30} simulations, for $W \in \{1, 2, 4, 8, 16\}$. Because of the added clock cycle latency, throughput results obtained for different ILS widths are normalized over their average latency. This ensures that the calculated ILS throughput is adjusted to account for the additional contention introduced by the parallel linear sorters.

The previous analysis is dependent upon a uniform distribution of tags among the W linear sorters. This is

Table 2: Average latency for interleaved linear sorter of width W.

Number of linear sorters W	Clock cycles
1	1.000
2	1.500
4	2.125
8	2.597
16	3.078

easily accomplished if the tags to be sorted are effectively random (from an interleaving perspective), or if they have been assigned incrementally contiguous or evenly distributed numbers. On the other hand, nonuniform distributions require additional custom input logic. The input logic to interleave multiple values to the linear sorters plays an important role in determining the overall throughput of an ILS. As the ILS width W increases, so will the logic and routing delay, diminishing the overall clock frequency of the system as we will demonstrate in Section 6. In terms of area, the depth of each linear sorter in an ILS can be set independently to accommodate different types of distributions.

4.2. Linear Sorter Depth. One of the key attributes of a linear sorter is its regular structure. Because of this regularity, a linear sorter of depth N can be implemented with little effect over the system's logic and routing delays. Therefore, regardless of the linear sorter depth, its implementation maintains timing performance. This is an asset, since ideally the linear sorter depth N needs to be large enough to prevent full conditions.

In simple cases, full linear sorters stall the input flow until one of its elements is removed. With more complex systems, there is a possibility that the logic controlling the sorted output streaming is input dependent (e.g., the system starts streaming after the tag number 1 is identified as the top sorted value). If the ILS needs to validate a condition before streaming its output, it will accumulate values until the condition is met. It is then possible that one of the queues becomes full and not able to service inputs or outputs, that is, a deadlock condition. Because only a single value can be inserted and deleted in each linear sorter, the depth N must be large enough to allow a sliding window of values. That is, at any given time, every input value received can be dependent only on the inputs received less than N clock cycles before. Fortunately, N can be easily changed to accommodate this restriction.

4.3. Output Logic. The ILS system needs to accumulate values before its sorted output becomes relevant. Therefore, there must be some logic in control of output streaming. A simple choice is to start the output when one of the linear sorters is detected as full or almost full, ensuring a continuous flow of sorted values. Another option would be to test the top sorted tag value before streaming. If we were to ensure the order for predefined tags, then the output logic must test each tag in a round-robin fashion before deletion from each linear sorter (an extension of the system presented in Figure 6). Because W tag values need to be inspected in a single clock cycle, this method's logic and routing generally limits the maximum frequency of the system.

An alternative approach for sorting the ILS's output is to use a pipelined sorting network. This sorting network can resort the top values from each of the W linear sorters. Since W is generally small, the necessary processing elements, latency, and area are also kept small, thus overcoming the hindrances of a sorting network implementation.

4.4. Simulation Results. We used ModelSim to simulate and functionally verify the ILS system. On each rising clock edge, the ILS system receives a new input set of tags, and situations involving contention for the interleaved linear sorters input ports are solved through input buffering. Figure 9 shows the simulation for signals of interest when such events occur. The initial parallel input of `Input Tags` consists of tags $\{0, 1, 6, 7\}$ which are interleaved without incident, sorted immediately, and sent to the `Sorted Output` port in the next rising clock at 50 ns. The following set of inputs $\{4, 5, 2, 3\}$ do not experience any contention for interleaved linear sorters either and are sorted in one clock cycle. At this point, the `Sorted Output` is $\{0, 1, 2, 3\}$, which are the top sorted tags able to be dispensed at a rate of four sorted tags per clock cycle.

Contention for linear sorter 0 occurs in the tag set $\{8, 9, 12, 11\}$, since zero is the interleaving value for tags 8 and 12. Tag 8 is serviced immediately because of the round-robin servicing scheme, and Tag 12 is saved to the `Buffered Input` register. The `Contention` flag now indicates the input logic to start buffering, which invalidates further input to the ILS system until it services the buffered tags. Tag 12, now residing in `Buffered Input`, is serviced in the next clock cycle at 70 ns. Finally, the input tags $\{10, 13, 14, 18\}$ will have three tags in contention for a linear sorter input, and therefore, two extra clock cycles will need to be used to buffer and dispatch them.

5. Delayed Contention Buffering

There is potential for improving the linear sorter contention scheme. The `Buffered Input` signal accumulates all contending values from the current tag set and dispenses them immediately. The average latency cost associated with this procedure was previously shown in Table 2. We instead accumulate contending tags over multiple sets of inputs, dispensing them when the `Buffered Input` signal is full. Table 3 shows the average performance increase when using this delayed buffered contention resolution scheme against the previously described immediate contention resolution scheme.

When the interleaved linear sorter width W is one, there is no benefit from the contention buffer as no conflicts arise from using the single linear sorter in the system. Additionally, when the contention buffer size CW is less than the ILS width W, additional logic must be implemented for worst case contention scenarios. In these cases, it is possible to have

Table 3: Average latency speedup with contention buffering for interleaved linear sorter of width W.

ILS width W	Contention buffer size CW					
	1	2	4	8	16	32
1	0%	0%	0%	0%	0%	0%
2	0%	9%	12%	14%	15%	17%
4	0%		23%	33%	41%	46%
8	0%			30%	45%	57%
16	0%				37%	57%

Table 4: Reduced average latency for interleaved linear sorter of width W.

Number of linear sorters W	Clock cycles	Speedup %
1	1.000	0%
2	1.344	12%
4	1.595	33%
8	1.752	45%
16	2.254	57%

$W - 1$ conflicting requests for a single linear sorter that need to be buffered. Since the buffer size CW cannot hold all these requests, a second buffer would be needed, subsequently making situations in which $CW < W$ quite unappealing due to the extra logic, buffer, and latency requirements.

Table 3 shows that for cases in which $CW \geq W$, there are speedups ranging from 9% to 57%. The greater speedups are achieved for cases in which both linear sorter contention is probable (large W), and the contention buffer can hold multiple requests accumulated over one or various clock cycles (large CW). This is convenient, as it subsidizes the elevated logic and routing costs associated with large interleaved linear sorter widths, thus not only maintaining their throughput but also reducing latency costs. Because the contention buffer prevents tags from being serviced immediately, the sliding window for contiguous values is increased. This requires a complementary increase in the linear sorter depth to augment storage capabilities for unsorted tags. In general, the extra storage capacity should exceed CW extra nodes per linear sorter, for a total of $W \times CW$ sorter nodes in the ILS system. Because additional linear sorter nodes do not greatly affect performance, there is only an area overhead cost associated with this reduced latency benefit.

We chose the cases in which the contention buffer input size CW was twice the interleaved linear sorter system's width W. This buffer size is practical, because it avoids overflows when multiple values get sent to a contention buffer that is almost full. Table 4 presents the revised latency of the ILS system with respect to its width. The additional latency for large values of W is not as dramatic as was presented previously on Table 2 and, therefore, counters the reduced clock frequencies experienced in implemented systems, as Section 6 will describe in detail.

Table 5: ILS FPGA area.

Linear sorters, W	Total slices	Slices/node	Area overhead
1	278	17.4	2.3%
2	641	20.0	17.6%
4	1294	20.2	18.8%
8	2612	20.4	20.0%
16	5250	20.5	20.6%

6. Hardware Implementation

The hardware implementation for the interleaved linear sorter depends on the width W of the ILS, the tag width T, the data width D, and the depth N for each of the W linear sorters. Of those, only three affect the maximum clock frequency. The width W tends to dominate the frequency, the depth N only has a minor effect, and larger values of T create longer tag comparisons. Tag sizes of 32-bits decreased the frequency by 16% compared to an 8-bit tag system. The data and tag sizes directly influence the total area. The number of the linear sorters, sorters depth, data, and tag size are parameterized offline before synthesis.

We used Xilinx ISE 11.1 and EDK 8.2 to synthesize the ILS for a Virtex-5 FPGA. The area for a linear sorter node was fairly static. For an 8-bit data and 8-bit tag linear sorter node, 17 FPGA slices were needed. The total area was generally scaled by the total number of nodes $W \times N$. Table 5 shows ILS systems of different widths W and their respective areas, with each linear sorter being 16 nodes deep.

The area overhead comes from the extra implementation logic for interleaving among multiple linear sorters. Likewise, it also includes the adder/subtractor and counter necessary for detecting full conditions on the linear sorters. Storage for the data to be sorted utilizes the FPGA board's BlockRAMs.

6.1. Throughput. The maximum throughput was calculated as the product of the maximum frequency and the number of linear sorters in the ILS. This assessment includes the logic necessary to interleave the inputs to their corresponding linear sorter, which for large values of W became a performance bottleneck. Table 6 shows the implemented Virtex-5 frequencies for $W \in \{1, 2, 4, 8, 16\}$, with 8-bit tags and 8-bit data, averaged for linear sorters with depth $N \in \{1, 2, 4, 8, 16, 32, 64, 96, 126, 256\}$. Even though at $W = 16$ the ILS throughput per clock cycle of the ILS is high, the clock frequency drops to 40 MHz due to the large logic and routing delays at the input logic stage.

The frequency discrepancies for ILS systems of width $W = 8$ and $W = 16$ are the result of the underlying logic cell in the Virtex-5 board. The lookup table (LUT), is a 6-input component that implements function generators. In the first three cases in Table 6, only one LUT is needed to generate functions with one, two, or four inputs. When the ILS has eight different inputs, two 6-input LUTs have to be cascaded to generate functions. With sixteen inputs, four LUTs are needed. This cascading of LUTs doubles and quadruples the

Table 6: ILS frequency and throughput.

Linear sorters W	Frequency (MHz)	Maximum sorting throughput (millions/s)
1	299	299
2	275	550
4	275	1101
8	132	1058
16	40	645

processing delay of inputs, and decreases the clock frequency accordingly.

Using these frequencies, we can now show the maximum throughput for different ILS configurations. Figure 10 shows the maximum sorting rate for $W \in \{1, 2, 4, 8\}$ for varying linear sorter depths N with 8-bit datas and tags. The 16-wide ILS was omitted due to the poor performance that resulted from its low clock frequency. The frequencies diminish as a function on ILS depth because of fanout. The input signals in each linear sorter have to be forwarded to all sorter nodes, which consequently increases the routing delay of these signals. In turn, the ILS clock frequency is decreased, but the effect is minimal in comparison to the routing delay experienced from interleaving tags to their corresponding linear sorters.

For ILS systems with two, four, and eight linear sorters, the average speedups against a single linear sorter were 1.8, 3.7, and 3.5, respectively. The diminishing returns trend is evident for $W = 8$, which shows a maximum throughput comparable to an ILS with $W = 4$. All of the ILS implementations showed higher throughput when compared to a single linear sorter system.

The maximum throughput results in Figure 10 do not consider two important factors. First, interleaving contention for the same linear sorter results in an average latency that increases with W, as previously shown in Tables 2 and 4. For the average case, the maximum frequency needs to be normalized by one of these factors. Second, the input logic for a small ILS width W will result in fairly simple logic and minimal delay. It is unlikely that this ILS delay would limit a nontrivial system. Instead, it is most likely that logic delays elsewhere in the same system determine the critical path and consequently sets the maximum frequency. As such, we assumed a maximum frequency of 300 MHz, which was the highest frequency obtained for 16-bit comparisons. This eliminates some of the artificially high frequencies an isolated ILS system achieves.

We first evaluate the decreased performance experienced when normalizing our maximum throughput due to interleaving contention for the system's linear sorters. Figure 11 shows the average throughput of an 8-bit tag and 8-bit data interleaved linear sorter, normalized by the corresponding ILS latency in Table 2 using immediate contention resolution with a 300 MHz maximum frequency. The interleaved linear sorters with $W = 2$ show a speedup of 1.3 while the one with $W = 4$ a 1.8 speedup. The $W = 8$ ILS, unfortunately, falls short at a 1.4 speedup due to its lower clock frequencies, contention, and complex input logic.

By using delayed contention resolution with a buffer twice the size of the interleaving width, we subsidize timing penalties of conflicting requests by dispatching them at once. Table 4 specified the ILS latency for our different interleaving values. Implementation of ILS systems with width $W > 4$ was not practical due to extreme routing delays. Contention buffering requires each of the W tag inputs to be routed to any of the $2 \times W$ contention buffer cells. The routing is not only expensive, but it is also magnified for $W > 4$ due to the FPGA's logic cell features, which limit LUT inputs to 6 and require cascading for larger values. The average frequency reduction for an ILS system with width $W = 8$ was more than 45% (the average speedup using contention buffering), making this enhancement perform worse than the original solution.

Figure 12 shows improved results for ILS systems with delayed contention buffering. The difference is more pronounced for $W = 4$, when the system can reap all the benefits of buffering without paying the expensive routing delay price. Its average sorting rate was 633 million values sorted per second, compared to 518 million when using immediate contention resolution.

6.2. Virtex Implementation. The implementation of a 4-way interleaving ILS was compared against quicksort running in a MicroBlaze processor, both in a Virtex-2 Pro ML310 device. The clock frequencies for software and hardware implementations matched the bus frequency of 100 MHz.

Data to be sorted resided in BlockRAMs. To create the tags, a pseudorandom scheme was used. The size of our sliding window for tag generation was set at 64, meaning that two contiguous tags would not be more than 16 address spaces apart within the four BlockRAMs. Unsorted sets of 64 tags and data were written in random order to the BlockRAMs while ensuring the sliding window property. The same data was used for both the MicroBlaze and the interleaved linear sorter tests.

The MicroBlaze version ran a C program for quicksort. The unsorted data resided as an array of values in a single BlockRAM, and values were retrieved and written through the on-chip peripheral bus (OPB) BRAM interface controller. For a small dataset of 64 values, MicroBlaze took 49,982 clock cycles, which include bus arbitration and read and write requests over the OPB. The end result is a sorted set saved in BlockRAM.

Three scenarios were setup with the ILS system doing hardware-based sorting. In each scenario, the unsorted data was saved in four BlockRAMs, with each BlockRAM output connected to the corresponding interleaved linear sorter input. The ILS output logic was set to delete tags and values in strictly contiguous increments, acting as a sorter rather than a priority queue. This means the output will halt until the next tag in the sequence is sorted to the top of the queue, as was shown in Figure 6, ensuring that we will get the same end results as the MicroBlaze test. Setting the depth of the four linear sorters to 16 nodes prevents each individual linear

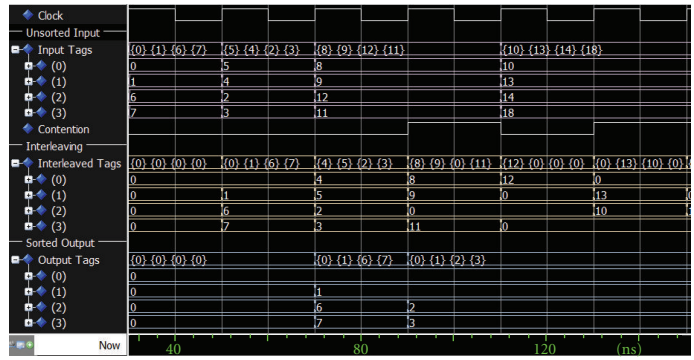


Figure 9: Simulation for interleaved linear sorter, including contention.

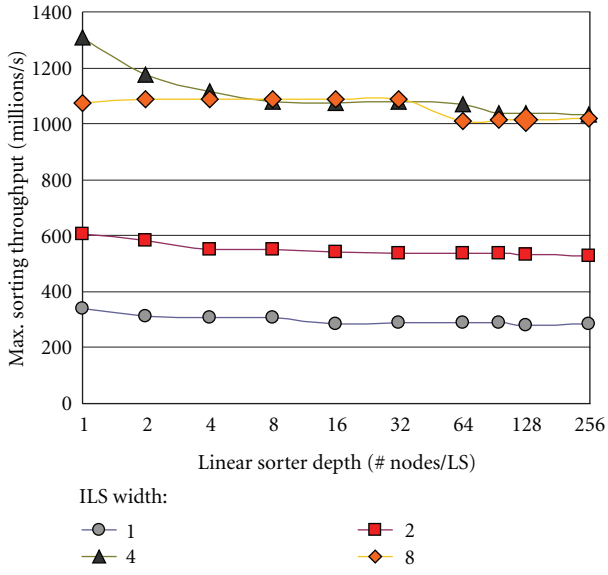


Figure 10: Maximum throughput for interleaved linear sorters.

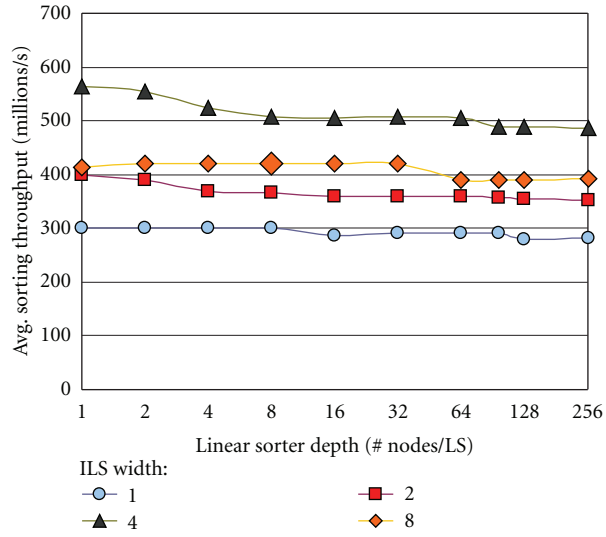


Figure 11: Normalized throughput for interleaved linear sorter system.

sorter from filling up while waiting for the input arrival of contiguous tags, even in the worst case scenario. This depth property further ensures that the input streaming remains not stalled due to lack of space in any of the linear sorters, preventing the system from going into a deadlock situation and allowing continuous output streaming.

In the first scenario, a MicroBlaze processor writes the unsorted data to the four OPB-based BlockRAMs and then sets a signal that starts the input streaming into the ILS. A simple hardware counter was used to drive the addresses of the BRAMs to cycle through all the unsorted values. The ILS output was then connected to the input ports of four secondary OPB BRAMs that hold the sorted values. Finally, the MicroBlaze reads back the sorted values through an on-chip peripheral bus (OPB) BRAM interface controller. Sorting with an ILS takes 2272 clock cycles, achieving a speedup of 22 over the MicroBlaze-only option.

The second scenario is set up in the same fashion as the first, but the results do not need to be read back into the MicroBlaze over the arbitrated bus since final storage happens in the ILS itself, whose linear sorters are deep enough to hold all the data. However, the MicroBlaze checks the ILS output to ensure all values have been retrieved before streaming the sorted results into the four OPB BlockRAMs, which takes a total of 732 clock cycles. Again, the end result is the sorted set saved in BlockRAM, but the speedup is magnified to 68 from the initial setup.

The third and final scenario involves a hardware-only approach with no MicroBlaze involvement. The output from the ILS is consumed by other hardware components as soon as it is ready (but still keeping the contiguous sorted output limitation). Under these circumstances, the interleaved linear sorting system takes only 30 clock cycles, a speedup of 1666 over the MicroBlaze quicksort.

6.3. Superscalar Processor Implementation. The greatest benefit of our interleaved linear sorter system is achieved on a hardware-only computational platform. Even though it is difficult to find applications that can fit this sorting scheme, the speedups attained in these systems make our approach

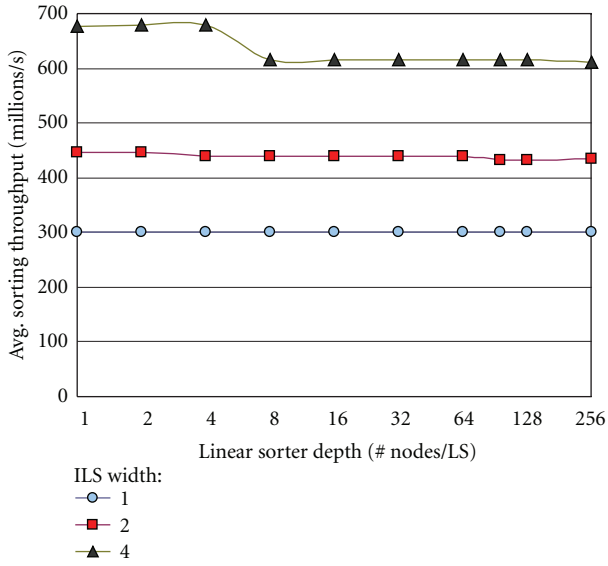


Figure 12: Improved normalized throughput for ILS system with delayed contention buffering.

an attractive option. Therefore an ideal target application is one that receives an unsorted streaming input of values, and transforms them into a streaming output of sorted data for other hardware components. One such application is a reconfigurable superscalar processor.

The configurable interleaved linear sorter was implemented as a component in an FPGA-based configurable superscalar processor [21, 22], which required resorting of processed results to allow precise interrupts. The superscalar processor utilizes register renaming and out-of-order execution to increase performance by exploiting instruction-level parallelism. One of the key components is its heterogeneous collection of functional units, which provide specialized processing for different instruction types at reduced execution latencies. Within the realm of reconfigurable computing, these functional units can also provide custom execution of instructions. The chronological sequence of events for an instruction execution starts with instruction retrieval, decoding, elimination of false data dependencies through register renaming, and storage in the reservation stations. These waiting instructions contain only true data dependencies, and once their operands are ready from the execution of previous instructions, they can be immediately dispatched in out-of-order fashion. Performance is thus increased when executing multiple instructions simultaneously in the parallel functional units.

To service interrupts from external sources like I/O and network, the processor saves its state, services the interrupt, and then restores state to continue execution. The superscalar's in-flight instructions, variable number of functional units, and out-of-order execution make restoration particularly difficult, so a reorder buffer is utilized to track in-order completion of instructions. The reorder buffer was an ideal target application for an interleaved linear sorter, as it had streaming input of instructions, streaming output of executed results, and a sliding window that set a maximum

Table 7: Superscalar and reorder buffer comparison.

	Min. delay	Max. frequency
Interleaving 1		
Superscalar processor	15.3 μs	65 MHz
Interleaved linear sorter system	3.612 μs	277 MHz
Interleaving 2		
Superscalar processor	16.5 μs	61 MHz
Interleaved linear sorter system	4.746 μs	211 MHz
Interleaving 4		
Superscalar processor	17.8 μs	56 MHz
Interleaved linear sorter system	6.941 μs	144 MHz

latency between contiguous tags depending on the number of in-flight instructions. Additionally, the processor's reconfigurable and superscalar nature also demanded increased throughput and performance while requiring flexibility for different number of instruction streams. The interleaved linear sorter system met all these throughput, streaming, and adaptability requirements.

When used as the reorder buffer for the reconfigurable superscalar processor, the interleaved linear sorter system surpassed the throughput requirements of the system. The amount of interleaving was determined by a reconfigurable parameter of the processor, which controlled the memory banks available for parallel instruction retrieval. Table 7 shows the implementation delay and performance for a superscalar processor with an instruction issue width of four and four parallel functional units.

In all the interleaving cases in Table 7, the ILS system experiences less delay than that of the register renaming process in the superscalar processor. On average, the ILS delay was 34% of the register renaming delay, with 26% of that delay due to logic and 74% due to routing.

6.4. Comparison to Other Sorters. To compare our performance with that of a state-of-the-art Batcher odd-even sorting network implementation, we used the results from [20], which were also implemented in a Xilinx Virtex 2 device. The Batcher odd-even method took 95 ns to sort thirty-two 16-bit numbers. We set the ILS system to also sort thirty-two 16-bit tag values, taking 123 ns. While the mergesort performed better for a static data set, it still suffers from the detriments of sorting networks, namely the need to resort the full set of data upon a single new insertion and a large area implementation.

7. Conclusions

Linear sorters overcome the latency disadvantages of sorting networks and systolic sorters. Their regular structure makes them highly configurable and a fitting solution for streaming data. Nevertheless, their single-output nature limits their

throughput. We have presented an implementation using interleaving of linear sorters that alleviates this limitation, using a delayed contention buffering scheme to maintain a low sorting latency. An ILS system of width 4 showed, on average, a 1.8 speedup over a regular linear sorter and a speedup of 68 against an embedded MicroBlaze processor. In an all hardware-implementation without the need of bus requests, like our superscalar processor implementation, this speedup became 1666. The versatility of the ILS system also allows designers to easily configure the number of sorting nodes per linear sorter and the number of linear sorters in the system to best match system bandwidth and area requirements. This configurability makes ILS systems easily adaptable to a variety of applications, particularly those that require high throughput for sorting streaming data.

References

[1] K. Batcher, "Sorting networks and their applications," in *Proceedings of the AFIPS Spring Joint Computer Conference*, vol. 32, pp. 307–314, ACM, 1968.

[2] E. W. Dijkstra, "A heuristic explanation of Batcher's Baffler," *Science of Computer Programming*, vol. 9, no. 3, pp. 213–220, 1987.

[3] J. Ortiz and D. Andrews, "A configurable high-throughput linear sorter system," in *Proceedings of the 7th IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, Atlanta, Ga, USA, 2010.

[4] D. E. Knuth, *The Art of Computer Programming 3. Sorting and Searching*, Addison-Wesley Longman, Amsterdam, The Netherlands, 2nd edition, 1998.

[5] M. de Prycker, *Asynchronous Transfer Mode: Solution for Broadband ISDN*, Ellis Horwood, Upper Saddle River, NJ, USA, 1991.

[6] R. O. Onvural, *Asynchronous Transfer Mode Networks: Performance Issues*, Artech House, Norwood, Ma, USA, 2nd edition, 1995.

[7] R. Kannan, "A pipelined single-bit controlled sorting network with O(Nlog2 N) bit complexity," in *Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '97)*, vol. 1, pp. 253–260, April 1997.

[8] J. Martínez, R. Cumplido, and C. Feregrino, "An FPGA-based parallel sorting architecture for the Burrows wheeler transform," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '05)*, pp. 7–13, September 2005.

[9] N. Tabrizi and N. Bagherzadeh, "An ASIC design of a novel pipelined and parallel sorting accelerator for a multiprocessor-on-a-chip," in *Proceedings of the 6th International Conference on ASIC (ASICON '05)*, vol. 1, pp. 46–49, October 2005.

[10] D. Castells-Rufas, M. Monton, L. Ribas, and J. Carrabina, "High performance parallel linear sorter core design," GSPx, The International Embedded Solutions Event, September 2004.

[11] C. Leiserson, "Systolic priority queues," in *Proceedings of the Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pp. 199–214, 1979.

[12] B. Parhami and D. M. Kwai, "Data-driven control scheme for linear arrays: application to a stable insertion sorter," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 1, pp. 23–28, 1999.

[13] Y. Zhang and S. Q. Zheng, "Design and analysis of a systolic sorting architecture," in *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pp. 652–659, October 1995.

[14] M. Bednara, O. Beyer, J. Teich, and R. Wanka, "Hardware-supported sorting: design and Tradeoff analysis," in *System Design Automation: Fundamentals, Principles, Methods, Examples*, p. 97, 1979.

[15] C.-S. Lin and B.-D. Liu, "Design of a pipelined and expandable sorting architecture with simple control scheme," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '02)*, vol. 4, pp. 217–220, 2002.

[16] R. Perez-Andrade, R. Cumplido, F. M. Del Campo, and C. Feregrino-Uribe, "A versatile linear insertion sorter based on a FIFO scheme," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI: Trends in VLSI Technology and Design (ISVLSI '08)*, pp. 357–362, April 2008.

[17] C. Y. Lee and J. M. Tsai, "A shift register architecture for high-speed data sorting," *Journal of VLSI Signal Processing*, vol. 11, no. 3, pp. 273–280, 1995.

[18] A. A. Colavita, A. Cicuttin, F. Fratnik, and G. Capello, "SORTCHIP: a VLSI implementation of a hardware algorithm for continuous data sorting," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 6, pp. 1076–1079, 2003.

[19] R. Marcelino, H. Neto, and J. Cardoso, "Sorting units for FPGA-based embedded systems," in *Proceedings of the IFIP 20th World Computer Congress, TC10 Working Conference on Distributed and Parallel Embedded Systems (DIPES '08)*, vol. 271, pp. 11–22, Springer, Milano, Italy, September 2008.

[20] L. Ribas, D. Castells, and J. Carrabina, "A linear sorter core based on a programmable register file," in *Proceedings of the 19th Conference on Design of Circuits and Integrated Systems (DCIS '04)*, pp. 635–640, 2004.

[21] J. Ortiz, "A reconfigurable superscalar processor architecture for FPGA-based designs," in *Proceedings of the International Conference on Computer Design (CDES '09)*, pp. 211–217, July 2009.

[22] J. Ortiz, *Synthesis techniques for semi-custom dynamically reconfigurable superscalar processors*, Ph.D. dissertation, University of Kansas, Department of Electrical Engineering and Computer Science, 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 514581, 15 pages
doi:10.1155/2011/514581

Research Article

The Potential for a GPU-Like Overlay Architecture for FPGAs

Jeffrey Kingyens and J. Gregory Steffan

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada M5S 3G4

Correspondence should be addressed to J. Gregory Steffan, steffan@eecg.toronto.edu

Received 3 August 2010; Accepted 27 December 2010

Academic Editor: Aravind Dasu

We propose a soft processor programming model and architecture inspired by graphics processing units (GPUs) that are well-matched to the strengths of FPGAs, namely, highly parallel and pipelinable computation. In particular, our soft processor architecture exploits multithreading, vector operations, and predication to supply a floating-point pipeline of 64 stages via hardware support for up to 256 concurrent thread contexts. The key new contributions of our architecture are mechanisms for managing threads and register files that maximize data-level and instruction-level parallelism while overcoming the challenges of port limitations of FPGA block memories as well as memory and pipeline latency. Through simulation of a system that (i) is programmable via NVIDIA's high-level `Cg` language, (ii) supports AMD's CTM r5xx GPU ISA, and (iii) is realizable on an XtremeData XD1000 FPGA-based accelerator system, we demonstrate the potential for such a system to achieve 100% utilization of a deeply pipelined floating-point datapath.

1. Introduction

As FPGAs become increasingly dense and powerful, with high-speed I/Os, hard multipliers, and plentiful memory blocks, they have consequently become more desirable platforms for computing. Recently there is building interest in using FPGAs as accelerators for high-performance computing, leading to commercial products such as the SGI RASC which integrates FPGAs into a blade server platform, and XtremeData and Nallatech that offer FPGA accelerator modules that can be installed alongside a conventional CPU in a standard dual-socket motherboard.

The challenge for such systems is to provide a programming model that is easily accessible for the programmers in the scientific, financial, and other data-driven arenas that will use them. Developing an accelerator design in a hardware description language such as Verilog is difficult, requiring an expert hardware designer to perform all of the implementation, testing, and debugging required for developing real hardware. Behavioral synthesis techniques—that allow a programmer to write code in a high-level language such as `C` that is then automatically translated into custom hardware circuits—have long-term promise [1–3], but currently have many limitations.

What is needed is a high-level programming model specifically tailored to making the creation of custom FPGA-based accelerators easy. In contrast with the approaches of custom hardware and behavioral synthesis, a more familiar model is to use a standard high-level language and environment to program a processor, or in this case an FPGA-based soft processor. In general, a soft-processor-based system has the advantages of (i) supporting a familiar programming model and environment and (ii) being portable across different FPGA products and families, while (iii) still allowing the flexibility to be customized to the application. Although soft processors themselves can be augmented with accelerators that are in turn created either by hand or via behavioral synthesis, our long-term goal is to develop a soft processor architecture that is more naturally capable of fully utilizing the FPGA.

1.1. A GPU-Inspired System. Another recent trend is the increasing interest in using the Graphics Processing Units (GPUs) in standard PC graphics cards as general-purpose accelerators, including NVIDIA's CUDA, OpenCL, and AMD (ATI)'s Close-to-the-Metal (CTM) [4] programming environments. While the respective strengths of GPUs and FPGAs are different—GPUs excel at floating-point

computation, while FPGAs are better suited to fixed-point and nonstandard bit width computations—they are both very well-suited to highly parallel and pipelinable computation. These programming models are gaining traction which can potentially be leveraged if a similar programming model can be developed for FPGAs.

In addition to the programming model, there are also several main architectural features of GPUs that are very desirable for a high-throughput soft processor. In particular, while some of these features have been implemented previously in isolation and shown to be beneficial for soft processors, our research highlights that when implemented in concert, they are key for the design of a high-throughput soft processor.

Multithreading. Through hardware support for multiple threads, a soft processor can tolerate memory and pipeline latency and avoid the area and potential clock frequency costs of hazard detection logic—as demonstrated in previous work for pipelines of up to seven stages and support for up to eight threads [5–7]. In our high-throughput soft processor we essentially avoid stalls of any kind for very deeply pipelined functional units (64 stages) via hardware support for many concurrent threads (currently up to 256 threads).

Vector Operations. A vector operation specifies an array of memory or register elements on which to perform an operation. Vector operations exploit data-level parallelism as described by software, allowing fewer instructions to command larger amounts of computation, and providing a powerful axis along which to scale the size of a single soft processor to improve performance [8, 9].

Predication. To allow program flexibility it is necessary to support control flow within a thread, although any control flow will make it more challenging to keep the datapath fully utilized—hence we support predicated instructions that execute unconditionally, but have no impact on machine state for control paths that are not taken.

Multiple Processors. While multithreading can allow a single datapath to be fully utilized, instantiating multiple processors can allow a design to be scaled up to use available FPGA resources and memory bandwidth [10]. The GPU programming model specifies an abundance of threads, and is agnostic to whether those threads are executed in the multithreaded contexts of a single processor or across multiple processors. Hence the programming model and architecture are fully capable of supporting multiple processors, although we do not evaluate such systems in this work.

1.2. Research Goals. Together, the above features provide the latency tolerance, parallelism, and architectural simplicity required for a high-throughput soft processor. Rather than inventing a new programming model, ISA, and processor architecture to support these features, as a starting point for this research we have ported an existing GPU programming model and architecture to an FPGA accelerator system. Specifically, we have implemented a `system-C` simulation of a GPU-inspired soft processor that (i) is programmable via NVIDIA's high-level `C`-based language called `Cg` [11], (ii) supports an *application binary interface*-(ABI-) based the AMD CTM r5xx GPU ISA [4], and (iii) is realizable on an XtremeData XD1000 development system composed of a dual-socket motherboard with an AMD Opteron CPU and the FPGA module which communicate via a HyperTransport (HT) link.

Our long-term research goal is to use this system to gain insight on how to best architect a soft processor and programming model for FPGA-based acceleration. In this work, through our implementation of the CTM ISA, we demonstrate that our heavily multithreaded GPU-inspired architecture can overcome several key challenges in the design of a high-throughput soft processor for acceleration—namely, (i) the port limitations of on-chip memories in the design of the main register file, (ii) tolerating potentially long latencies to memory, and (iii) tolerating the potentially long latency of deeply pipelined functional units. Furthermore, we envision that several aspects of this architecture can be extended in future implementations to better capitalize on the strengths of FPGAs: we can scale the soft processor in the vector dimension as in previous work [8, 9]; rather than focusing on floating-point computation, we can instead focus on nonstandard bit width computation or other custom functions; finally, we can scale the number of soft processor accelerators via multiprocessor implementations to fully utilize available memory bandwidth.

In this paper we (i) propose a new GPU-inspired architecture and programming model for FPGA-based acceleration based on soft processors that exploit multithreading, vector instructions, predication, and multiple processors; (ii) describe mechanisms for managing threads and register files that maximize parallelism while overcoming the challenge of port limitations of FPGA block memories and long memory and pipeline latencies; (iii) demonstrate that these features, when implemented in concert, result in a soft processor design that can fully utilize a deeply pipelined datapath. This paper extends our previous workshop publication at RAW 2010 [12] in several ways including a more detailed description of support for predication and handling control flow instructions, a more in-depth treatment of related work, nine additional illustrations and figures, and more detailed experimental results.

2. Related Work

2.1. Behavioral Synthesis-Based Compilers. The main challenge of behavioral synthesis algorithms is to identify parallelism in high-level code and generate a hardware circuit to provide concurrent execution of operations. There are many academic and commercial compilers that are based on synthesis to generate a customized circuit for a given task. Examples of such compilers include Impulse Accelerated Technologies Impulse C [13], Altera's C2H [2], Trident [3], Mitrionics Mitrion-C [1], SRC Computer's SRC Carte, ASC [14], and Celoxica's Handel-C [15]. Typically, these tools will compile C-like code to circuit descriptions in HDL which can then be synthesized by standard FPGA

CAD tools for deployment on accelerator systems such as the Xtremedata XD1000. A synthesis-based compiler will exploit data dependences in high-level code to build local, point-to-point routing at the circuit level. Computations can potentially be wired directly from producer to consumer, bypassing a register store for intermediate computations, a step which is required for general purpose processors. This synthesis-based technique of customized circuits can be especially practical for GPU-like computations, as programs are typically short sequences of code. Where the computation is data-flow dominated, it is also possible to exploit data-level parallelism (DLP) by pipelining independent data through the custom circuit. The downsides to a behavioral synthesis approach are that (i) a small change to the application requires complete/lengthy recompilation of the FPGA design, and (ii) the resulting circuit is not easily understood by the user, which can make debugging difficult.

2.2. Soft Uniprocessors. Soft processors are microprocessors instantiated on an FPGA fabric. Two examples of industrial soft processors are the Altera NIOS and the Xilinx Microblaze. As these processors are deployed on programmable logic, they come in various standard configurations and also provide customizable parameters for application-specific processing. The ISA of NIOS soft processors is based on a MIPS instruction set architecture (ISA), while that of Microblaze is a proprietary reduced instruction set computer (RISC) ISA. SPREE [16] is a development tool for automatically generating custom soft processors from a given specification. These soft processor architectures are fairly simple single-threaded processors that do not exploit parallelism other than pipelining.

2.3. Vector Soft Processors. Yu et al. [17] and Yiannacouras et al. [8] have implemented soft vector processors where the architecture is partitioned into independent vector lanes, each with a local vector register file. This technique maps naturally to the dual port nature of FPGA on-chip RAMs and allows the architecture to scale to a large number of vector lanes, where each lane is provided with its own dual port memory. While the success of this architecture relies on the ability to vectorize code, for largely data-parallel workloads, such as those studied in our work, this is not a challenge. Soft vector processors are interesting with respect to our work because we also rely on the availability of data parallelism to achieve performance improvements. However, while vector processors scale to many independent lanes each with a small local register file, our high throughput soft processors focus on access to a single register file. Hence our techniques are independent and therefore make it possible to use both in combination.

2.4. Multithreaded Soft Processors. Fort et al. [5], Dimond et al. [18], and Labrecque and Steffan [6] use multithreading in soft processor designs, and show that it can improve area efficiency dramatically. While these efforts focused on augmenting an RISC-based processor architecture with multithreading capabilities, we focus on supporting a GPU stream processor ISA. As the GPU ISA is required to support floating point-based multiply-add operations, the pipeline depth is much longer. Therefore, we extend the technique here to match the pipeline depth of our functional units. Although we require many more simultaneous threads, the data-parallel nature of the GPU programming model provides an abundance of such threads.

2.5. SPMD Soft Processors. The GPU programming model is only one instance in the general class of Single-Program Multiple-Data (SPMD) programming models. There has been previous work in soft processor systems supporting SPMD. James-Roxby et al. [19] implement a SPMD soft processor system using a collection of Microblaze soft processors attached to a global shared bus. All soft processors are connected to a unified instruction memory as each are executing instructions from the same program. All soft processors are free to execute independently. When a processor finishes executing the program for one piece of data, it will request more work from a soft processor designated to dispatch workloads. While their work focuses on a multi processor system, little attention is paid to the optimization of a single core. This is primarily because the work is focused on SPMD using soft processors as a rapid prototyping environment. While the GPU programming model is scalable to many processors, we focus on the optimization of a single processor instance. The system-level techniques used by James-Roxby et al. [19] such as instruction memory sharing between processors could be applied in a multiprocessor design of our high-throughput soft processors.

2.6. Register File Access in Soft Processors. As instruction-level parallelism increases in a soft processor design, more read and write ports are required to sustain superscalar instruction issue and commit rates. In trying to support the AMD r5xx GPU ISA, we were confronted with the same problem, as this ISA requires 4 read and 3 write accesses from a single register file, each clock cycle, if we are to fully pipeline the processor datapath. One possibility is to implement a multiported register file using logic elements as opposed to built-in SRAMs. For example, Jones et al. [20] implement such a register file using logic elements. However, they show that using this technique results in a register file with very high area consumption, low clock frequency, and poor scalability to a large number of registers. LaForest and Steffan [21] summarize the conventional techniques for building multiported memories out of FPGA block RAMs (replication, banking, and multipumping) and also describe a new technique called the *live value table*. The solution we propose in this paper is a form of banking that exploits the availability of independent threads. Saghir et al. [22] have also used multiple dual-port memories to implement a banked register file, allowing the write-back of two instructions per cycle in cases where access conflicts do not occur; however, they must rely on the compiler to schedule register accesses within a program such that writes are conflict free. We exploit the execution of multiple threads in lock step, allowing us to build conflict-free accesses to a

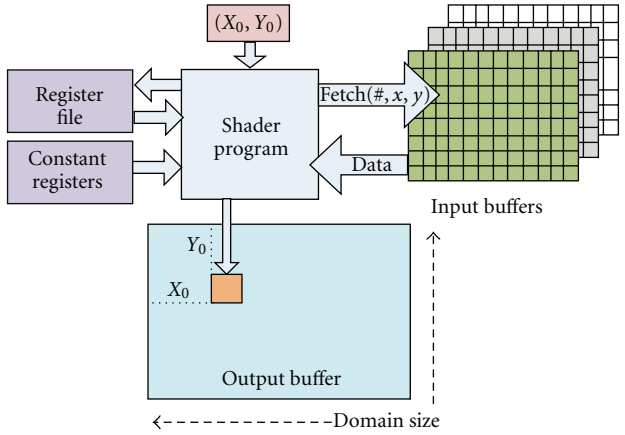


Figure 1: The interaction of a shader program with memories and registers.

banked register file in hardware. While they bank the register file across multiple on-chip memories, we provide banked access both within a single register, through interleaving with a wide memory port, in addition to accessing across multiple memory blocks.

3. System Overview

In this section, we give an overview of our system as well as for GPUs, in particular their shader processors. We also briefly describe the two software interfaces that we use to build our system: (i) NVIDIA's `Cg` language for high-level programming and (ii) the AMD CTM SDK that defines the application binary interface (ABI) that our soft processor implements.

3.1. GPU Shader Processors. While GPUs are composed of many fixed-function and programmable units, the shader processors are the cores of interest for our work. For a graphics workload, shader processors perform a certain computation on every vertex or pixel in an input stream, as described by a shader program. Since the computation across vertices or pixels is normally independent, shader processors are architected to exploit this parallelism: they are heavily multithreaded and pipelined, with an ISA that supports both vector parallelism as well as predication. Furthermore, there are normally multiple shader processors to improve overall throughput.

Figure 1 illustrates how a shader program can interact with memory: input buffers can be randomly accessed while output is limited to a fixed location for each shader program instance, as specified by an input register. Hence the execution of a shader program implies the invocation of parallel instances across all elements of the output buffer. This separation and limitation for writing memory simplifies issues of data coherence and is more conducive to high-bandwidth implementations.

3.2. The NVIDIA `Cg` Language. In our system we exploit NVIDIA's `Cg` [11], a high-level, `C`-based programming language that targets GPUs. To give a taste of the `Cg` language, Figure 2(b) shows a sample program written in `Cg` for elementwise multiplication of two matrices, with the addition of a unit offset. For comparison, ANSI `C` code is provided for the same routine in Figure 2(a). In `Cg`, the `multadd` function defines a computation which is implicitly executed across each element in the output domain, hence there are no explicit `for` loops for iterating over the output buffer dimensions as there would be in regular `C`. The dimensions of the buffers are configured prior to execution of the shader program and hence do not appear in the `Cg` code either.

Looking at the `Cg` code, there are three parameters passed to the `multadd` function. First, 2D floating-point coordinates (`coord`) directly give the position for output in the output buffer and are also used to compute the positions of values in input buffers (i.e., the (X_0, Y_0) input pair shown in Figure 1)—note that a future implementation could remove this limitation to allow many-dimensional coordinates. The second and third parameters (`A` and `B`) define the input buffers, associated with a buffer number (`TEXUNIT0` and `TEXUNIT1`) and a memory addressing mode, `uniform sampler2D`, that in this case tells the compiler to compute addresses same way `C` computes memory addresses for 2D arrays. `Tex2D()` is an intrinsic function to read data from an input buffer, and implements the addressing mode specified by its first parameter on the coordinates specified by its second parameter. For this program the values manipulated including the output value are all of type `float4`, a vector of four 32-bit floating-point values—hence the buffer sizes and addressing modes must account for this.

While `Cg` allows the programmer to abstract-away many of the details of the underlying GPU ISA, it is evident that an ideal high-level language for general-purpose acceleration would eliminate the remaining graphics-centric artifacts in `Cg`.

3.3. AMD's CTM SDK. The AMD CTM SDK is a programming specification and tool set developed by AMD to abstract the GPU's shader processor core as a data-parallel accelerator [4, 23], hiding many graphics-specific aspects of the GPU. As illustrated in Figure 3, we use the `cgc` compiler included in NVIDIA's `Cg` toolkit to compile and optimize shader programs written in `Cg`. `cgc` targets Microsoft pixel shader virtual assembly language (`ps3`), which we then translate via CTM's amucomp compiler into the AMD CTM application binary interface (ABI) based on the r5xx ISA. The resulting CTM shader program binary is then folded into a *host program* that runs on the regular CPU. The host program interfaces with a low-level *CTM driver* that replaces a standard graphics driver, providing a *computer* interface (as opposed to graphics-based interface) for controlling the GPU. Through driver API calls, the host program running on the main CPU configures several parameters prior to shader program execution, including the base address and sizes of input and output buffers as well as constant register data (all illustrated in Figure 1). The host program also uses the CTM driver to load shader program binaries onto the GPU for execution.

```
float   A[WIDTH][HEIGHT];   // input buffer A
float   B[WIDTH][HEIGHT];   // input buffer B
float   C[WIDTH][HEIGHT];   // output buffer

void multadd(void){
  for(int j=0;j<HEIGHT;j++)
    for(int i=0;i<WIDTH;i++)
      C[i][j] = A[i][j]*B[i][j] + 1.0f;
}
```

(a) C code

```
struct data_out {
  float4 sum : COLOR;
};

data_out
multadd(float2 coord : TEXCOORD0,
        uniform sampler2D A: TEXUNIT0,
        uniform sampler2D B: TEXUNIT1){
  data_out r;
  float4 offset = {1.0f, 1.0f, 1.0f, 1.0f};
  r.sum = tex2D(A,coord)*tex2D(B,coord)+offset;
  return r;
}
```

(b) Cg code

```
multadd:
  TEX r1 r0.rg s1
    // r1 = r0.r + (r0.g*s1.width) + s1.base
  TEX r0 r0.rg s0
    // r0 = r0.r + (r0.g*s0.width) + s0.base
  MAD o0 r1 r0 c0
    // o0 = r1 * r0 + c0 (3 left-most elems)
  mad o0 r1 r0 c0
    // o0 = r1 * r0 + c0 (1 right-most elem)
  END
```

(c) CTM code

Figure 2: An example shader program for elementwise matrix multiplication plus an offset, described in (a) C, (b) Cg, and (c) CTM code.

Figure 2(c) shows the resulting CTM code from the example shader program in Figure 2(b). From left to right, the format of an instruction is *opcode*, *destination*, and *sources*. There are several kinds of registers in the CTM ISA: (i) general-purpose vector registers (r0–r127); (ii) "sampler" registers (s0–s15), used to specify the base address and width of an input buffer (i.e., TEXUNIT0–TEXUNIT15 in Cg code); (iii) constant registers (c0–c255), used to specify constant values; (iv) output registers (o0–o3) that are used as the destination for the final output values which are streamed to the output buffer (shown in Figure 1) when the shader program instance completes. All registers are each a vector of four 32-bit elements where the individual elements of the vector are named r, g, b, and a. Both base registers and constant registers are configured during setup by the CTM driver, but are otherwise read-only.

CTM defines both TEX and ALU instructions. A TEX instruction defines a memory load from an input buffer, and essentially implements the Tex2D() call in Cg. The input coordinates (coord in Cg) are made available in register r0 at the start of the shader program instance. The address is computed from both r0 and a sampler register (i.e., s0). For example, the address for the sources given as r0·rgs1 is computed as r0.r+r0.g∗s1.width+s1.base. All ALU instructions are actually VLIW operation pairs that can be issued in parallel: a three-element vector operation specified

in uppercase, followed (on a new line) by a scalar operation specified in lower case. In the example the ALU instruction is a pair of *multiply add*s that specify three-source operands and one destination operand for both the vector (MAD) and scalar (mad) operations. ALU instructions can access any of r0–r127 and c0–c255 as any source operand.

CTM allows many other options that we do not describe here, such as the ability to permute (swizzle) the elements of the vectors after loading from input buffers or before performing ALU operations, and also for selectively masking the elements of destination registers. A complete description of the r5xx ISA and the associated ABI format is available in the CTM specification [23].

In summary, this software flow allows us to support existing shader programs written in `Cg`, and also allows us to avoid inventing our own low-level ISA.

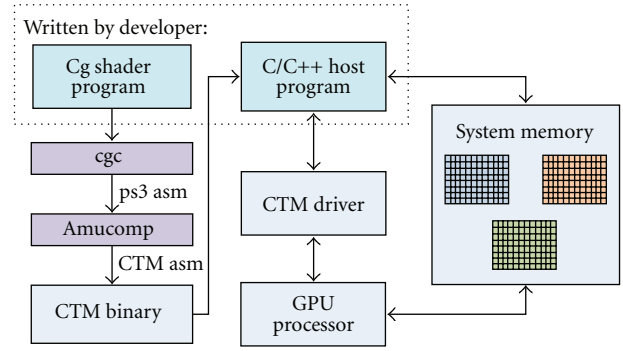


Figure 3: The software flow in our system. A software developer writes a high-level `Cg` shader program and a host program. The `Cg` shader program is translated into a CTM binary via the `cgc` and `amucomp` compilers and is then folded into the host program.

4. A GPU-Inspired Architecture

In this section we describe the architecture of our high-throughput soft-processor accelerator, as inspired by GPU architecture. First we describe an overview of the architecture, and explain in detail the components that are relatively straightforward to map to an FPGA-based design. We then describe three features of the architecture that overcome challenges of an FPGA-based design.

4.1. Overview. Figure 4 illustrates the high-level architecture of the proposed GPU-like accelerator. Our architecture is designed specifically to interface with a HyperTransport (HT) master and slave, although interfacing with other interconnects is possible. The following describes three important components of the accelerator that are relatively straightforward to map to an FPGA-based design.

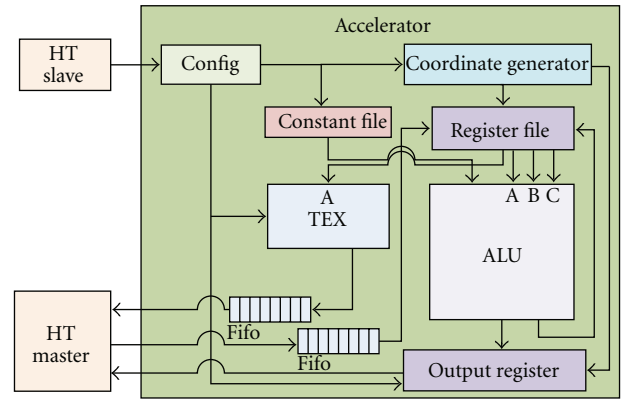


Figure 4: Overview of a GPU-like accelerator, connected to a Hypertransport (HT) master and slave.

Coordinate Generation. As described in Section 3 and by the CTM specification, a shader program instance is normally parameterized entirely by a set of input coordinates which range from the top-left to the bottom-right of the compute domain. The coordinate generator is configured with the definition of the compute domain and generates streams of coordinates which are written into the register file (register `r0`) for shader program instances to read—replacing outer-looping control flow in most program kernels.

TEX and ALU Datapaths. TEX instructions, which are essentially loads from input buffers in memory, are executed by the TEX datapath. Once computed based on the specified general-purpose and sampler registers, the load address is packaged as an HT read request packet and sent to the HT core—unless there are already 32 in-flight previous requests in which case the current request is queued in a FIFO buffer. When a request is satisfied, any permutation operations (as described in Section 3.3) are applied to the returned data and the result is written back to the register file. The CTM ISA also includes a method for specifying that an instruction depends on the result of a previous memory request (via a special bit). Each TEX instruction holds a semaphore that is cleared once its result is written back to the register file—which signals any awaiting instruction to continue. ALU instructions are executed by the ALU datapath, and their results can be written to either the register file or the output register.

Predication and Control Flow. Conditional constructs such as *if* and *else* statements in `Cg` are supported in CTM instructions via predication. There is one predicate bit per vector lane that can be set using the boolean result from one of many comparison operations (e.g., >, <=, ==, ! =). Subsequent ALU instructions can then impose a write mask conditional on the values of these bits. More complex control flow constructs such as *for* loops and subroutines are supported via a control flow instructions that provide control of hardware-level call/return stacks, and branch and loop-depth hierarchies; space limitations prevent us from fully-explaining control flow instructions here, hence we refer the reader to the CTM specification [23] for further details.

Output. Similar to input buffers, the base addresses and widths of the output buffers are preconfigured by the CTM driver in advance (in the registers o0–o3). When a shader program instance completes, the contents of the output

registers are written to the appropriate output buffers in memory: the contents of the output registers are packaged into an HT write request packet, using an address derived from one of the output buffer base addresses and the original input coordinates (from the coordinate generator). Write requests are *posted*, meaning that there is no response packet and hence no limit on the maximum number of outstanding writes.

4.2. Tolerating Limited Memory Ports. In Figure 4 we observe that there are a large number of ports feeding into and out of the central register file (which holds r0–r127). One of the biggest challenges in high-performance soft processor design is the design of the register file: it must tolerate the port limitations of FPGA block memories that are normally limited to only two ports. To fully pipeline the ALU and TEX datapaths, the central register file for our GPU-inspired accelerator requires four read and three write ports. If we attempted a design that reads all of the ALU and TEX source operands (four of them) of a single thread in a single cycle, we would be required to have replicated copies of the register file across multiple block memories to have enough ports. However, this solution does not provide more than one write port, since each replicant would have to use one port for reading operands and the other port for broadcast-writing the latest destination register value (i.e., being kept up-to-date with one write every cycle).

We solve this problem by exploiting the fact that all threads are executing different instances of the same shader program: all threads will execute the exact same sequence of instructions, since even control flow is equalized across threads via predication. This symmetry across threads allows us to group threads into batches and execute the instructions of batched threads in lock-step. This lock-step execution in turn allows us to *transpose* the access of registers to alleviate the ports problem.

Rather than attempting to read all operands of a thread each cycle, we instead read a single operand across many threads per cycle from a given block memory and do this across separate block memories for each component of the vector register. Table 1 illustrates how we schedule register file accesses in this way for batches of four threads each that are decoding only ALU instructions (for simplicity). Since there are three operands to read for ALU instructions this adds a three-cycle decode latency for such instructions. However, in the steady-state we can sustain our goal of the execution of one ALU instruction per cycle, hence this latency is tolerable. This schedule also leaves room for another read of an operand across threads in a batch. Ideally we would be able to issue the register file read for a TEX instruction during this slot, which would allow us to fully utilize the central register file, ALU datapath and TEX datapath: every fourth cycle we would read operands for a batch of four threads for a TEX instruction, then be able to issue a TEX instruction for each of those threads over the next four cycles. We give the name transpose to this technique of scheduling register accesses.

This transposed register file design also eases the implementation of write ports. In fact, the schedule in Table 1 uses only one read port per block memory, leaving the other port free for writes. From the table we see that ALU instructions will generate at most one register write across threads in a batch every four cycles. There are two other events which result in a write to the central register file: (i) a TEX instruction completes, meaning that the result has returned from memory and must be written-back to the appropriate destination register; (ii) a shader program instance completes for a batch of threads and a new batch is configured, so that the input coordinates must be set for that new batch (register `r0`). These two types of register write are performed immediately if the write port is free, otherwise they are queued until a subsequent cycle.

Table 1: The schedule of operand reads from the central register file for batches of four threads (T0–T3, T4–T7, etc.) decoding only ALU instructions. An ALU instruction has up to three vector operands (A, B, C) which are read across threads in a batch over three cycles. In the steady state this schedule can sustain the issue of one ALU instruction from every cycle.

Clock cycle	Inst phase	Register file read	ALU ready
0	ALU_0	ALU:A(T0,T1,T2,T3)	—
1	ALU_1	ALU:B(T0,T1,T2,T3)	—
2	ALU_2	ALU:C(T0,T1,T2,T3)	—
3	—	—	T0
4	ALU_0	ALU:A(T4,T5,T6,T7)	T1
5	ALU_1	ALU:B(T4,T5,T6,T7)	T2
6	ALU_2	ALU:C(T4,T5,T6,T7)	T3
7	—	—	T4
8	ALU_0	ALU:A(T8,T9,T10,T11)	T5
9	ALU_1	ALU:B(T8,T9,T10,T11)	T6
10	ALU_2	ALU:C(T8,T9,T10,T11)	T7
11	...	...	...

4.3. Avoiding Pipeline Bubbles. In the previous section we demonstrated that a transposed register file design can allow the hardware to provide the register reads and writes necessary to sustain the execution of one ALU instruction every cycle across threads. However, there are three reasons why issuing instructions to sustain such full utilization of the datapaths is a further challenge. The first reason is as follows. In the discussion of Table 1 we described that the ideal sequence of instructions for fully utilizing the ALU and TEX datapaths is an instruction stream which alternates between ALU and TEX instructions. This is very unlikely to happen naturally in programs, and the result of other nonideal sequences of instructions will be undesirable bubbles in the two datapaths. The second reason, as shown in Figure 5, is that the datapath for implementing floating-point operations such as multiply add (`MAD`) and dot product (`DOT3`, `DOT4`) instructions is very long and deeply pipelined (64 clock cycles): since ALU instructions within a thread will often have register dependences between them, this can prevent an ALU instruction from issuing until a previous ALU instruction completes. This potentially long stall will also result in unwanted bubbles in the ALU datapath.

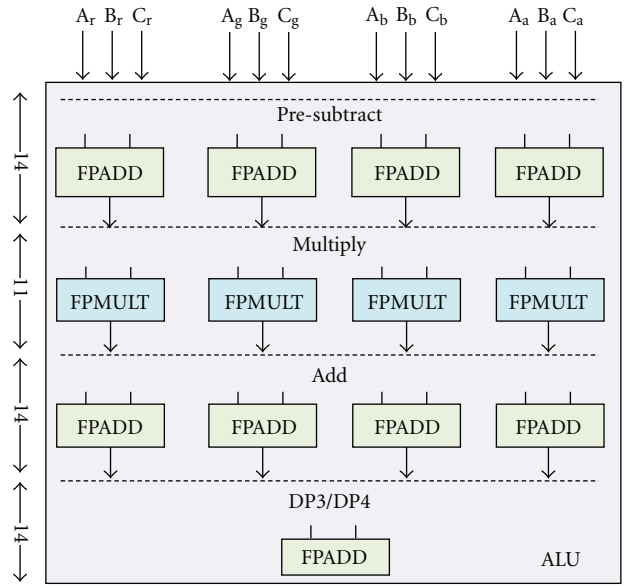


Figure 5: The floating point units in a datapath that supports MADD, DP3, and DP4 ALU instructions. The pipeline latency of each unit is shown on the left (for Altera floating point IP cores), and the total latency of the datapath is 53 cycles without accounting for extra pipeline stages for multiplexing between units.

The third reason is that TEX instructions can incur significant latency since they load from main memory; since an ALU instruction often depends on a previous TEX instruction for a source operand, the ALU instruction would have to stall until the TEX instruction completes.

We address all three of these problems by storing the contexts of multiple batches in hardware, and dynamically switching between batches every cycle. We capitalize on the fact that all threads are independent across batches as well as within batches, switching between batches to (i) choose a batch with an appropriate next instruction to match the available issue phase (TEX or ALU) and (ii) to hide both pipeline and memory latency. This allows us to potentially fully utilize both the ALU and TEX datapaths, provided that ALU and TEX instructions across all batch contexts are ready to be issued when required. Specifically, to sustain this execution pattern we generally require that the ratio of ALU to TEX instructions be 1.0 or greater: for a given shader program if TEX instructions outnumber ALU instructions then in the steady state this alone could result in pipeline bubbles. Storing the contexts (i.e., register file state) of multiple batches is relatively straightforward: it requires only growing the depth of the register file to accommodate the additional registers—although this may require multiple block memories to accomplish. In the next section we describe the implementation of batch issue logic in greater detail.

4.4. Control Flow. The thread batching and scheduling solutions above present problems for the control flow instruction, FLW. The first problem of finding time to schedule the execution of such instructions is easily solved.

Table 2: The schedule of operand reads from the central register file for batches of four threads (T0–T3, T4–T7, etc.) decoding both ALU and TEX instructions. TEX instructions require only one source operand, hence we can read source operands for four threads in a single cycle.

Clock cycle	Inst phase	Register file read	ALU ready	TEX ready
0	ALU_0	ALU:A(T0,T1,T2,T3)	—	—
1	ALU_1	ALU:B(T0,T1,T2,T3)	—	—
2	ALU_2	ALU:C(T0,T1,T2,T3)	—	—
3	TEX	TEX:A(T0,T1,T2,T3)	T0	—
4	ALU_0	ALU:A(T4,T5,T6,T7)	T1	T0
5	ALU_1	ALU:B(T4,T5,T6,T7)	T2	T1
6	ALU_2	ALU:C(T4,T5,T6,T7)	T3	T2
7	TEX	TEX:A(T4,T5,T6,T7)	T4	T3
8	ALU_0	ALU:A(T8,T9,T10,T11)	T5	T4
9	ALU_1	ALU:B(T8,T9,T10,T11)	T6	T5
10	ALU_2	ALU:C(T8,T9,T10,T11)	T7	T6
11	...	...	...	...

Column 2 in Table 2 shows two cycles each period where the ALU is fetching operands B and C. As the batch scheduler and instruction issue logic is idle during this time, the hardware can be used to schedule an FLW instruction. Since FLW requires neither a read or write to the register file, no structural hazards arise.

The second problem is how to resolve diverging control flow. Diverging control flow is when threads within a batch decide to take alternate branch paths. It turns out, the r5xx ISA has encoded support within the FLW instruction to resolve this specific problem. This is because GPUs use the same technique to resolve diverging control path when executing multiple threads using SIMD hardware. The way these instructions are handled is through the hardware management of thread states which are not visible to the programmer. These thread states are manipulated by the control flow instructions, depending on the previous thread states (the state before an FLWinstruction is executed), the evaluation of the branch condition, and a resolution function when threads disagree. Much of this hardware level management is considered on a case-by-case basis. For example, as threads execute over an else instruction the active state of each thread is flipped. This requires all threads to execute serially through all branch paths and mask register writes when they are inactive. Fung et al. [24] have explored optimizations of this technique in more detail. For more details of how the r5xx ISA programs and handles control flow, see the CTM specification [23].

5. Implementation

This section describes our work to implement our GPU-inspired accelerator on the XtremeData platform. After an overview of the XtremeData XD1000 and how we map the CTM system to it, we describe the low-level implementation

of the two key components of our accelerator: the central register file and the batch issue logic.

5.1. The XtremeData XD1000. As illustrated in Figure 6, the XtremeData XD1000 is an accelerator module that contains an Altera Stratix II EPS180 FPGA, and that plugs into a standard CPU socket on a multisocket AMD Opteron motherboard. IP cores are available for the FPGA which allow access to system memory via Hypertransport (HT) that provides a single physical link per direction, each of which is a 16-bit-wide 400 MHz DDR interface and can transfer 1.6 GB/sec. The host CPU treats the XD1000 as an endpoint that is configured by a software driver to respond to a memory-mapped address range using the HT slave interface, similar to other regular peripherals. The FPGA application can also initiate DMA read and write transactions to system memory by constructing and sending HT request packets, providing efficient access to memory without involving the CPU. In our work we extend the XtremeData system to conform to the CTM interface by (i) adding a driver layer on top of the XD1000 driver, and (ii) by memory-mapping our accelerator's hardware configuration state registers and instruction memory to the HT slave interface. Instruction memory resides completely on-chip and stores up to 512 instructions—the limit currently defined by CTM. Each instruction is defined by the ABI to be 192 bits, hence the instruction store requires three M4K RAM blocks. The RAM blocks have two ports: one is configured as a write port that is connected directly to the configuration block so that the CTM driver can write instructions into it; the other is configured as a read port to allow the accelerator to fetch instructions. The CTM driver initializes the accelerator with the addresses of the start and end instruction of the shader program and initiates execution by writing to a predetermined address.

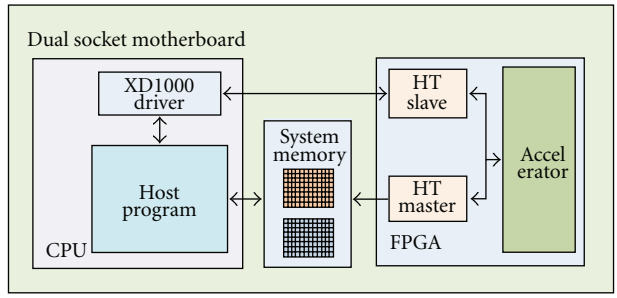


Figure 6: An XtremeData system with a XD1000 module.

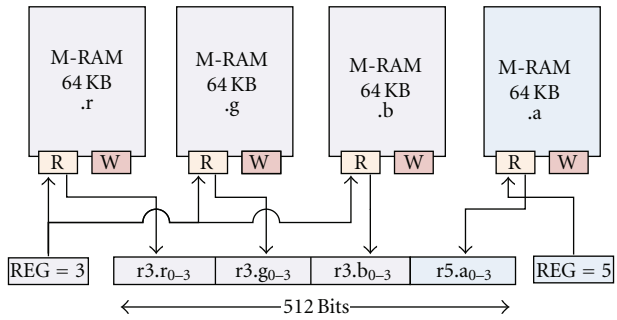


Figure 7: Mapping our register file architecture to four Stratix II's 64 KB M-RAM blocks. The read circuitry shows an example where we are reading operands across threads in a batch for a vector/scalar ALU instruction pair (VLIW): `r3` as an operand for the vector instruction and `r5` as an operand for the scalar instruction. While not shown, register writes are implemented similarly.

5.2. Central Register File. While our transposed design allows us to architect a high-performance register file using only two ports, the implementation has the additional challenges of (i) supporting the vast capacity required, and (ii) performing the actual transposition. Each batch is composed of four threads that each require up to 128 registers, where each register is actually a vector of four 32-bit elements. We therefore require the central register file to support 8 KB of on-chip memory per batch. For example, as illustrated in Figure 7, 32 batches would require 256 KB of on-chip memory, which means that we must use four of the 64 KB M-RAM blocks available in the Stratix II chip in the XD1000 module. Figure 8 shows the circuit we use to transpose the operands read across threads in a batch for ALU instructions so that the three operands for a single instruction are available in the same cycle: a series of registers buffer the operands until they can be properly transposed.

5.3. Batch Issue Logic. As described previously in Section 4.3, to ensure that the ALU datapath is fully utilized our soft processor schedules instructions to issue across batches. For a given cycle, we ideally want to find either an ALU or TEX instruction to issue. Figure 9 shows the circuit that performs this batch scheduling, for an example where we want to find an ALU instruction to issue. We can trivially compare the desired next instruction type (ALU in this case) with the actual next instruction type for each batch as recorded in the batch state register, since this information about the next instruction is encoded in each machine instruction (as defined by the CTM ABI). As shown in the figure, we take the set of boolean signals that indicate which batches have the desired next instruction ready to issue and rotate them, then feed the rotated result into a priority encoder that gives the batch number to issue. The rotation is performed such that the previously selected batch is in the lowest-priority position. In the example, we rotate such that the signal for batch 2 is in the lowest-priority right most position, and the priority encoder hence chooses batch 0 as the first batch with a ready ALU instruction. For a GPU-like programming model where all threads are executing the same sequence of instructions, this is sufficient to ensure fairness and forward progress. The batch issue logic can be pipelined with a total budget of four cycles for the circuit, hence during a second cycle the the batch number is used to index the context memory to read the program counter value for that batch, and during a third cycle the program counter value is used to index the instruction memory for the appropriate instruction.

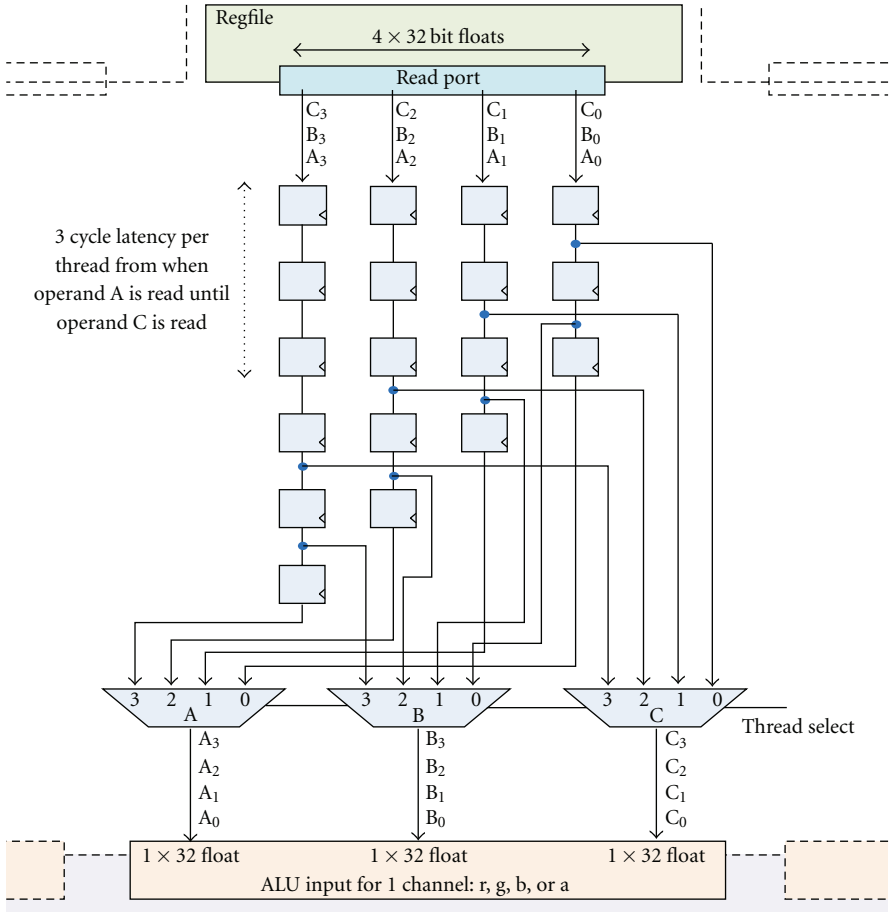


Figure 8: A circuit for transposing the thread-interleaved operands read from the central register file into a correctly ordered sequence of operands for the ALU datapath.

6. Measurement Methodology

In this section we describe the system simulation and benchmark applications that we use to measure the performance of our GPU-inspired soft processor implementation.

6.1. System Simulation. We have developed a complete simulation framework in `SystemC` to measure the ALU utilization and overall performance of workloads on our GPU-like soft processor. Note that we verify the functional accuracy of our simulation by comparing with the outputs of the CTM programs running on the ATI RV570 GPU (on an ATI Radeon x1950 Pro graphics card).

Clock Frequency. Since we do not have a full RTL implementation of our soft processor, we instead assume a system clock frequency of 100 MHz. We choose this frequency to match the 100 MHz HT IP core, which in turn is designed to match a 4x division of the physical link clock frequency (i.e., 400 MHz). We feel that this clock frequency is achievable since (i) other soft processor designs easily do so for Stratix II FPGAs such as the NIOS II/f which executes up to 220 MHz, and (ii) the GPU programming model and abundance of threads allows us to heavily pipeline all components in our design to avoid any long-latency stages.

HyperTransport. Our simulation infrastructure faithfully models the bandwidth and latency of the HT links between the host CPU and the FPGA on the XD1000 platform. We limit the number of outstanding read requests supported by the HT master block to 32, as defined by the HT specification; additional requests are queued. We compute the latency of an HT read request as a sum of the individual latencies of the subcomponents involved. We assume that our soft processor is running at 100 MHz as described above, and that the memory specification is the standard DDR-333 (166 MHz Bus) SDRAM that comes with the XD1000 system. We assume a constant SDRAM access latency of 51 ns, while a constant latency is of course unrealistic; since it contributes only 17% of total latency, we are confident that modeling the small fluctuations of this latency would not significantly impact our results. The latencies of the HT IP core (both input and output paths) were obtained from Slogsnat et al. [25], and the latencies for the the host HT controller, DDR controller, and DDR access were obtained from Holden [26]. Our HyperTransport model is somewhat idealized since we

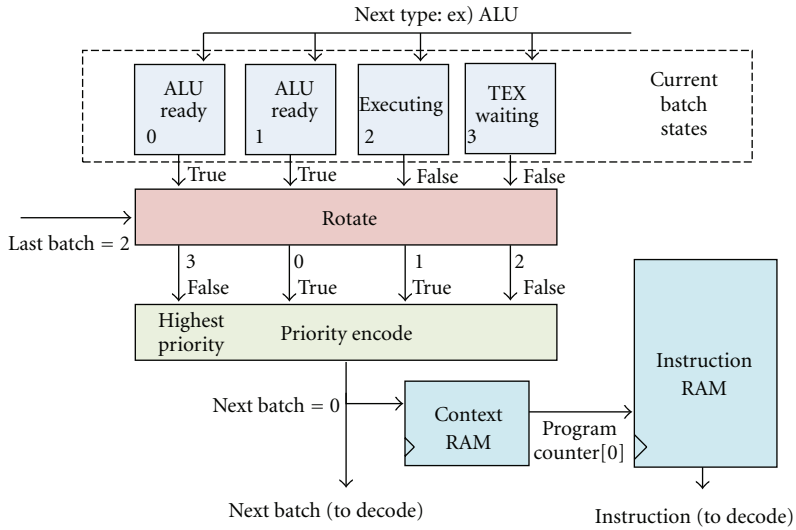


Figure 9: Batch issue logic for hardware managing 4 batch contexts.

Table 3: A breakdown of how each stage of an HT memory request contributes to overall access latency.

Action	Stage	Latency (ns)
Request	(1) FPGA HT IP core	70
	(2) host HT controller	32
SDRAM	(1) access and data fetch	51
Response	(1) host builds response packet	12
	(2) host HT controller	30
	(3) FPGA HT IP core	110
	Total Latency	*305*

do not account for possible HT errors nor contention by the host CPU for memory (Table 3).

Cycle-Accurate Simulation. Our simulator is cycle accurate at the block interfaces shown in Figure 4. For each block we estimate a latency based on the operations and data types present in a behavioral `C` code implementation. We assume that the batch issue logic shown previously in Figure 9 is fully pipelined, allowing us to potentially sustain the instruction issue schedule shown previously in Table 1.

6.2. Benchmarks. Since our system is compatible with the interface specified by CTM we can execute existing CTM applications, including `Cg` applications, by simply relinking the CTM driver to our simulation infrastructure. We evaluate our system using the following three applications that have a variety of instruction mixes and behaviors. Note that in our work so far we have not observed any applications with the potentially problematic instruction mix of more TEX instructions than ALU instructions.

Matmatmult. Matmatmult is included with the CTM SDK as CTM assembly code and performs dense matrix-matrix multiplication based on the work of Fatahalian et al. [27]. We selected this application because of its heavy use of TEX instructions to access row and column vectors of an input matrix: the ratio of ALU to TEX instructions for Matmatmult is 2.25.

Sgemm. Sgemm computes $C_{new} = \alpha(A \cdot B) + \beta C_{old}$ and represents a core routine of the BLAS math library, and was also included with the CTM SDK as CTM assembly code. Sgemm also makes heavy use of TEX instructions to access two input matrices. The ratio of ALU to TEX instructions for Sgemm is 2.56.

Photon. Photon is a kernel from a Monte Carlo radiative heat transfer simulation, included with the open-source Trident [3] FPGA compiler. We ported this application by hand to `Cg` such that each instance of the resulting shader program performs the computation for a single particle, and input buffers store previous particle positions and other physical quantities. We selected this benchmark to be representative of applications with higher ratios of ALU to TEX instructions: for Photon it is exactly 4.00.

7. Utilization and Performance

Our foremost goal is to fully utilize the ALU datapath. In this section we measure the ALU datapath utilization for several configurations of our architecture and also measure the impact on performance of increasing the number of hardware batch contexts. Recall that an increasing number of batches provide a greater opportunity for fully utilizing the pipeline and avoiding bubbles by scheduling instructions to issue across a larger number of threads.

Figure 10 shows ALU utilization assuming the 8-bit HT interface provided with the XD1000 system, for a varying number of hardware batch contexts—from one to 64 batches. Since each batch contains four threads, this means that we support from four to 256 threads. We limit the number of

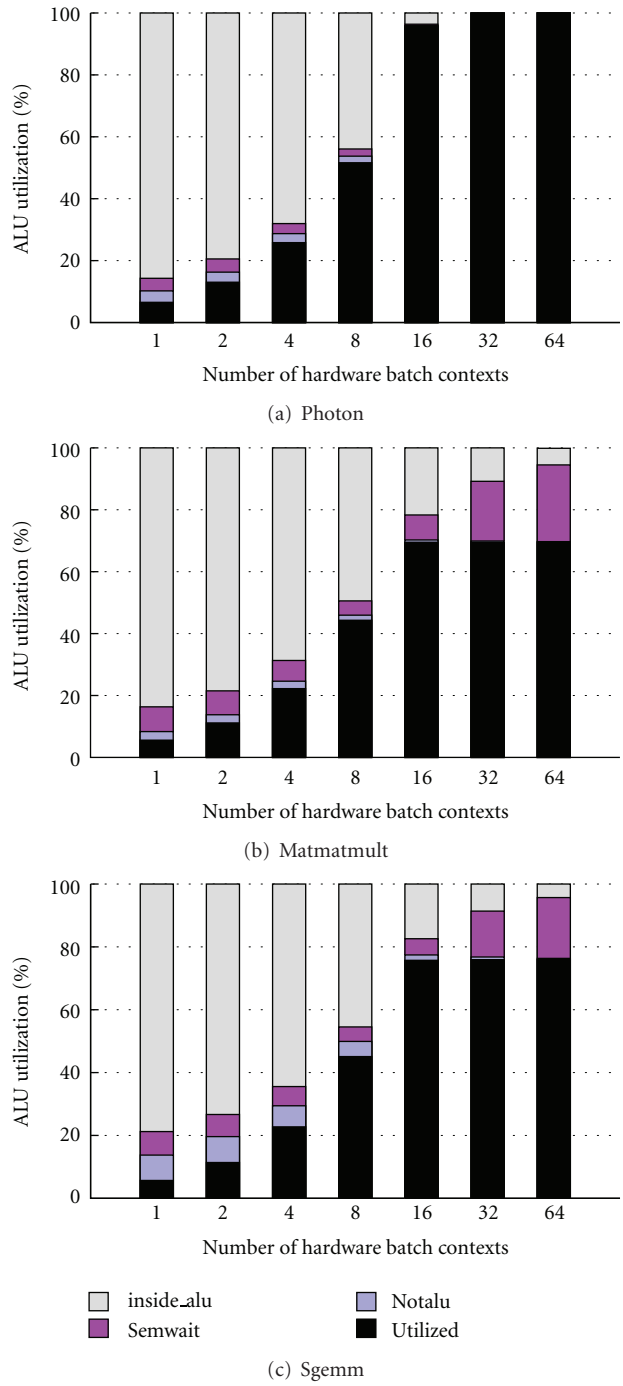


Figure 10: ALU datapath utilization for the 8-bit HT interface provided with the XD1000 system.

batch contexts to 64 because this design includes a central register file that consumes 512 KB of on-chip memory, and thus eight of the nine M-RAMs available in a Stratix II FPGA (64 KB each). In the figure we plot ALU utilization (*utilized*) as the fraction of all clock cycles when an ALU instruction was issued. We also break down the ALU idle cycles into the reasons why no ALU instruction from any batch could be issued (i.e., averaged across all batches contexts). In particular, we may be unable to issue an ALU instruction for a given batch for one of the following three reasons.

Semwait. The next instruction is an ALU instruction, but it is waiting for a memory semaphore because it depends on an already in-flight TEX instruction (memory load).

Inside_ALU. The next instruction is a ready-to-issue ALU instruction, but there is already a previous ALU instruction executing for that batch: since there is no hazard detection logic, a batch must conservatively wait until any previous ALU instruction from that batch completes before issuing a new one, to ensure that any register dependences are satisfied.

NotALU. The next instruction is not an ALU instruction.

From the figure we observe that when only one hardware batch context is supported the ALU datapath is severely underutilized (less than 10% utilization), and the majority of the idle cycles are due to prior ALU instructions in the ALU pipeline (*inside_ALU*). Utilization steadily improves for all three benchmarks as we increase the number of hardware batch contexts up to 16 batches, at which point Matmatmult and Sgemm achieve utilization of 70% and 75%, respectively. However, neither Matmatmult nor Sgemm benefit from increasing further to 32 batches: in both cases waiting for memory is the bottleneck (*Semwait*), indicating that both applications have consumed available memory bandwidth. Similarly, increasing even further to 64 batches yields no improvement, with the memory bottleneck becoming more pronounced. In contrast, for Photon the increase from 16 to 32 batches results in near perfect utilization of the ALU datapath; correspondingly, the increase from 32 to 64 batches cannot provide further benefit. Intuitively, Photon is able to better utilize the ALU datapath because it has a larger ratio of ALU to TEX instructions (four to one).

While the HT IP core provided for the XD1000 is limited to an 8-bit HT interface, the actual physical link connecting the FPGA and CPU is 16 bits. Since memory appears to be a bottleneck limiting ALU utilization, we investigate the impact of a 16-bit HT link such as the one described in [25] as shown in Figure 11. For this improved system we observe that the memory bottleneck is sufficiently reduced to allow full utilization of the ALU datapath for all three benchmarks when 32 hardware batch contexts are supported. In turn, this implies that support for 64 or more hardware batch contexts remains unnecessary. The fact that 32 batches seems sufficient makes intuitive sense since 32 batches comprises 128 threads, while the ALU datapath pipeline is 64 cycles deep and thus requires only that many ALU instructions to be fully utilized—more deeply pipelined ALU functional units would likely continue to benefit from increased contexts.

While maximizing ALU datapath utilization as our overall goal, it is also important to understand the impact of increasing the number of hardware batch contexts on performance. Figure 12. shows speedup relative to a single hardware batch context for both 8-bit and 16-bit HT interfaces. Interestingly, speedup is perfectly linear for between two and eight contexts for all benchmarks and both HT designs, but for 16 and more contexts speedup is sublinear.

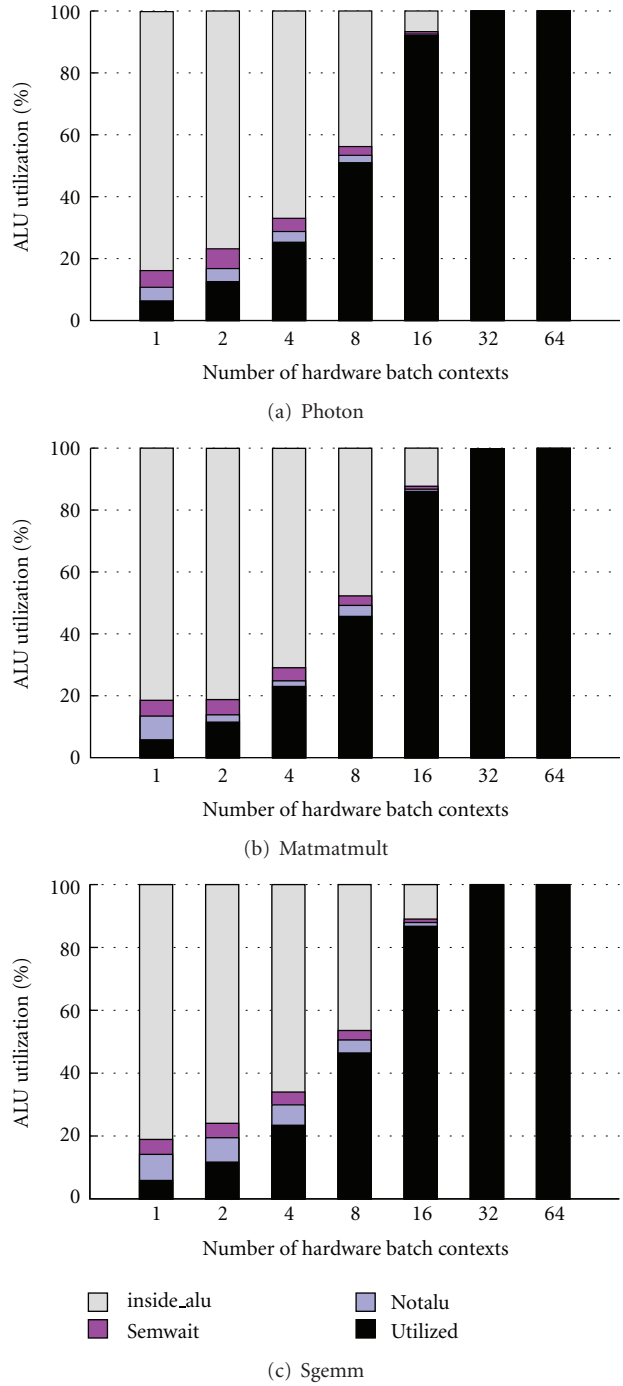


Figure 11: ALU datapath utilization for a 16-bit HT interface.

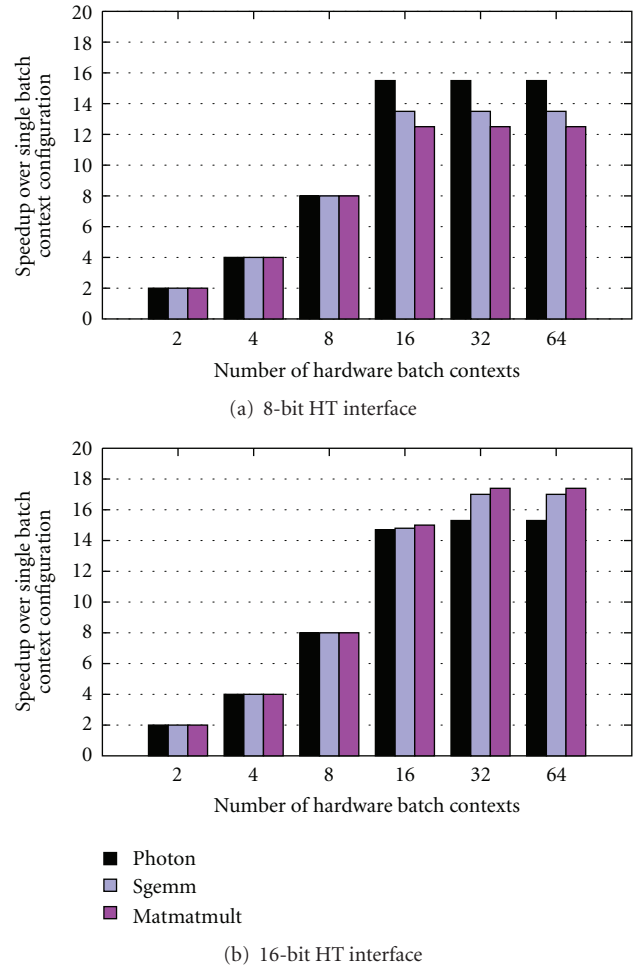


Figure 12: Speedup versus a single hardware batch context for (a) 8-bit and (b) 16-bit HT interfaces.

For the 8-bit HT interface performance does not improve beyond 16 contexts, while for the 16-bit HT interface 32 contexts provide an improvement but 64 contexts do not. Looking at the 16-bit HT interface for 32 contexts, we see that each benchmark speedup is inversely related to the ratio of ALU to TEX instructions: applications with a smaller fraction of TEX instructions benefit less from the latency-tolerance provided by a larger number of contexts. Photon benefits the least and has a ratio of 4.00, followed by Sgemm that has a ratio of 2.56; Matmatmult benefits the most and has a ratio of 2.25.

7.1. Reducing the Register File. While we have demonstrated that 32 hardware batch contexts is sufficient to achieve near perfect ALU datapath utilization, as shown in Section 5.2 this is quite costly, requiring four M-RAMs on the Altera Stratix II. While this may not be an issue if a single accelerator is the only design on the chip, if there are other components or multiple accelerators then storing all of the batch states would be a problem. However, most of the 128 general-purpose vector registers per thread defined by CTM will not be used for many applications. In our architecture it is straightforward to reduce the number of registers supported by a power of two to reduce the total memory requirements for the central register file. For example, Photon, Matmatmult, and Sgemm each use only 4, 15, and 21 general-purpose registers, hence the proposed customization would reduce the size of the central register file by 32x, 8x, and 4x, respectively; for Photon this would instead allow us to build the central register file using only 16 of the much smaller M4 K memories.

7.2. Summary. These results indicate that even for a deeply pipelined ALU of 53 clock cycles we are able to fully utilize this datapath by interleaving the execution of instructions from different batches. This is made possible by the abundance of independent threads provided by the data-parallel GPU programming model.

8. Conclusions

We have presented a GPU-inspired soft processor that allows FPGA-based acceleration systems to be programmed using high-level languages. Similar to a GPU, our design exploits multithreading, vector operations, and predication to enable the full utilization of a deeply pipelined datapath. The GPU programming model provides an abundance of threads that all execute the same instructions, allowing us to group threads into batches and execute the threads within a batch in lock-step. Batched threads allow us to (i) tolerate the limited ports available in FPGA block memories by transposing the operand reads and writes of instructions within a batch and (ii) to avoid pipeline bubbles by issuing instructions across batches. Through faithful simulation of a system that is realizable on an XtremeData XD1000 FPGA-based acceleration platform we demonstrate that our GPU-inspired architecture is indeed capable of fully utilizing a 64-stage ALU datapath when 32 batch contexts are supported in hardware. The long-term goal of this research is to discover new high-level programming models and architectures that allow users to fullyexploit the potential of FPGA-based acceleration platforms; we believe that GPU-inspired programming models and architectures are a step in the right direction.

References

[1] J. Koo, D. Fernandez, A. Haddad, and W. Gross, "Evaluation of a high-level-language methodology for high-performance reconfigurable computers," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '07)*, pp. 30–35, July 2007.

[2] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 45–54, April 2006.

[3] J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale, "Trident: an FPGA compiler framework for floating-point algorithms," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 317–322, August 2005.

[4] J. Hensley, "AMD CTM overview," in *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '07)*, ACM, August 2007.

[5] B. Fort, D. Capalija, Z. G. Vranesic, and S. D. Brown, "A multithreaded soft processor for SoPC area reduction," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 131–140, April 2006.

[6] M. Labrecque and J. G. Steffan, "Improving pipelined soft processors with multithreading," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 210–215, August 2007.

[7] R. Moussali, N. Ghanem, and M. A. R. Saghir, "Supporting multithreading in configurable soft processor cores," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '07)*, pp. 155–159, October 2007.

[8] P. Yiannacouras, J. G. Steffan, and J. Rose, "Vespa: portable, scalable, and flexible fpga-based vector processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '08)*, 2008.

[9] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core CPU accelerator," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '08)*, pp. 222–231, February 2008.

[10] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Scaling soft processor systems," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 195–205, April 2008.

[11] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a c-like language," in *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '03)*, pp. 896–907, ACM, New York, NY, USA, 2003.

[12] J. Kingyens and J. G. Steffan, "A GPU-inspired soft processor for high-throughput acceleration," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, April 2010.

[13] "Developing fpga coprocessors for performance-accelerated spacecraft image processing," *Xcell Journal Second Quarter*, pp. 22–26, 2008.

[14] O. Mencer, "ASC: a stream compiler for computing with FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, Article ID 1673737, pp. 1603–1617, 2006.

[15] I. Page, "Closing the gap between hardware and software: hardware-software cosynthesis at Oxford," in *Proceedings of the IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems*, pp. 201–211, February 1996, Digest no: 1996/036.

[16] P. Yiannacouras, J. Rose, and J. Gregory Steffan, "The microarchitecture of FPGA-based soft processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '05)*, pp. 202–212, New York, NY, USA, 2005.

[17] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core CPU accelerator," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '08)*, pp. 222–231, ACM, New York, NY, USA, 2008.

[18] R. Dimond, O. Mencer, and W. Luk, "Application-specific customisation of multi-threaded soft processors," *IEE Proceedings: Computers and Digital Techniques*, vol. 153, no. 3, pp. 173–180, 2006.

[19] P. James-Roxby, P. Schumacher, and C. Ross, "A single program multiple data parallel processing platform for FPGAs," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04)*, pp. 302–303, April 2004.

[20] A. K. Jones, R. Hoare, I. S. Kourtev et al., "A 64-way VLIW/SIMD FPGA architecture and design flow," in *Proceedings of the 11th IEEE International Conference on Electronics, Circuits and Systems (ICECS '04)*, pp. 499–502, December 2004.

[21] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proceedings of the 18th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '10)*, pp. 41–50, February 2010.

[22] M. A. R. Saghir, M. El-Majzoub, and P. Akl, "Datapath and isa customization for soft vliw processors," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA (ReConFig '06)*, pp. 1–10, September 2006.

[23] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine forgpus," in *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '06)*, p. 184, ACM, New York, NY, USA, 2006.

[24] W. W .L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual International Symposium on Microarchitecture (MICRO '07)*, pp. 407–418, IEEE Computer Society, Washington, DC, USA, 2007.

[25] D. Slogsnat, A. Giese, and U. Brüning, "A versatile, low latency HyperTransport core," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '07)*, pp. 45–52, ACM, New York, NY, USA, 2007.

[26] B. Holden, *Latency Comparison between HyperTransport and PCI-Express In Communications Systems*, World Wide Web Electronic Publication, 2006.

[27] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms formatrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPH-ICS Conference on Graphics Hardware*, pp. 133–137, ACM, New York, NY, USA, 2004.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 546962, 13 pages
doi:10.1155/2011/546962

Research Article

Boosting Parallel Applications Performance on Applying DIM Technique in a Multiprocessing Environment

Mateus B. Rutzig,[1] Antonio C. S. Beck,[1] Felipe Madruga,[1] Marco A. Alves,[1] Henrique C. Freitas,[2] Nicolas Maillard,[1] Philippe O. A. Navaux,[1] and Luigi Carro[1]

[1] *Instituto de Informática, Universidade Federal do Rio Grande do Sul, 91501-970 Porto Alegre, RS, Brazil*
[2] *Instituto de Informática, Pontifícia Universidade Católica de Minas Gerais, 30535-901 Belo Horizonte, MG, Brazil*

Correspondence should be addressed to Mateus B. Rutzig, mbrutzig@inf.ufrgs.br

Received 11 August 2010; Revised 14 January 2011; Accepted 14 February 2011

Academic Editor: Aravind Dasu

Limits of instruction-level parallelism and higher transistor density sustain the increasing need for multiprocessor systems: they are rapidly taking over both general-purpose and embedded processor domains. Current multiprocessing systems are composed either of many homogeneous and simple cores or of complex superscalar, simultaneous multithread processing elements. As parallel applications are becoming increasingly present in embedded and general-purpose domains and multiprocessing systems must handle a wide range of different application classes, there is no consensus over which are the best hardware solutions to better exploit instruction-level parallelism (TLP) and thread-level parallelism (TLP) together. Therefore, in this work, we have expanded the DIM (dynamic instruction merging) technique to be used in a multiprocessing scenario, proving the need for an adaptable ILP exploitation even in TLP architectures. We have successfully coupled a dynamic reconfigurable system to an SPARC-based multiprocessor and obtained performance gains of up to 40%, even for applications that show a great level of parallelism at thread level.

1. Introduction

Industry competition in the current electronics market makes the design of a device increasingly complex. New marketing strategies have been focusing on increasing the product functionalities to attract consumer's interest: they desire the equivalent of a supercomputer at the size of a portable device. However, the convergence of different functions in a single device produces new design challenges by enlarging the range of heterogeneous code that the system must handle. To worsen such scenario, the designers must take into account tighter design constraints as power budget and manufacturing process costs, all mixed up in the difficult task of increasing the processing capability.

Because of that, the instruction-level parallelism (ILP) exploitation strategy is no longer enough to improve the overall performance of general and embedded applications. The newest ILP exploitation techniques do not provide an advantageous tradeoff between the amount of transistors added and the extra speedup obtained [1, 2]. Despite the great advantages shown in the employment of instruction set architecture (ISA) extensions, like the employment of single instruction multiple data (SIMD) instructions, such approaches rely on long design and validation times, which goes against the need for a fast time-to-market for present day systems. On the other hand, application-specific integrated circuits (ASICs) provide high-performance and small chip area. However, such an approach attacks only a very specific application class, failing to deliver the required performance when executing applications in which behaviors were not considered at design time, being not suitable for executing general-purpose applications.

Reconfigurable systems appear as a mid-term between general-purpose processors and ASICs, solving somehow the ILP issues discussed before. They have already shown good performance improvements and energy savings for stand-alone applications in single core environments [3–6]. Adaptable ILP exploitation is the major advantage of

this technique, since the reconfigurable fabric can adapt to fit the required application parallelism degree at a given time, enabling acceleration over a wide range of different application classes.

However, as already discussed, general-purpose and embedded systems are composed of a wide range of applications with different behaviors, in which the parallelism grain available varies from the finest to the coarsest. To accelerate applications that present high level of coarse-grained parallelism (at thread/process level), multiprocessor systems are widely employed, providing high performance and short validation time [7]. However, in contrast to architectures that make use of fine-grained parallelism (at instruction level) exploitation, such as the superscalar processors, the usage of the multiprocessor approach leaves all the responsibility of parallelism detection and allocation to the programmers. They must split and distribute the parallelized code among processing elements, handling all the communication issues. The software partitioning is a key feature in a multiprocessor system: if it is poorly performed or if the application does not provide a minimum parallelism at process/thread levels, even the most computational powerful system will run way below their full potential.

Thus, to cover all possible types of applications, the system must be conceived to provide a good performance at any parallelism level and to be adaptable to the running applications. Nowadays, at one side of the spectrum, there are the multiprocessing systems composed of many homogeneous and simple cores to better explore the coarse-grained parallelism of highly thread-based applications. At the other side, there are multiprocessor chips assembled with few complex superscalar/SMT processing elements, to explore applications where ILP exploration is mandatory. As can be noticed, there is no consensus on the hardware logic distribution to explore the best of ILP and TLP together regarding a wide range of application classes.

In this scenario, we merge different concepts by proposing a novel dynamic reconfigurable multiprocessor system based on the dynamic instruction merging (DIM) technique [8]. This system is capable of transparently exploring (no changes in the binary code are necessary at all) the fine-grained parallelism of the individual threads, adapting to the available ILP degree, while at the same time taking advantage of the available thread/process parallelism. This way, it is possible to have a system that adapts itself to any kind of available parallelism, handling a wide range of application classes.

Therefore, the primary contributions of this work are

(i) to reinforce, by the use of an analytical model, the need for heterogeneous parallelism exploitation in multiprocessor environments,

(ii) to propose a multiprocessor architecture provided with an adaptable reconfigurable system (DIM technique), so it is possible to balance the best of both thread/process and ILP exploitations. This way, any kind of code, those that present high TLP and low ILP, or those that are exactly the opposite, will be accelerated.

2. Related Work

The usage of reconfigurable architectures in a multiprocessor chip is not a novel approach. In [9] the thread warping system is proposed. It is composed of an FPGA coupled to an ARM11-based multiprocessor system. Thread warping uses complex computer-aided detection (CAD) tools to detect, at execution time, critical regions of the running application and to map them to custom accelerators implemented in a simplified FPGA. A greedy knapsack heuristic is used to find the best possible allocation of the custom accelerators onto the FPGA, considering the possibility of partial reconfiguration. In this system, one processor is totally dedicated to run the operating system tasks needed to synchronize threads and to schedule their kernels to be executed in the accelerators. However, this processor may become overloaded if several threads are running on tens or hundreds of processors, affecting system scalability. Another drawback is that, due to the high time overhead imposed by the CAD and greedy knapsack algorithms, only critical code regions are optimized. Consequently, only applications with few and very defined kernels (e.g., filters and image processing algorithms) are accelerated, narrowing the field of application of this approach.

In [10], the Annabelle SoC is presented. It comprises an ARM core and four domain-specific coarse-grain reconfigurable architectures, named Montium cores. Each Montium core is composed of five 16-bit arithmetic and logic units (ALUs), structured to accelerate DSP applications. The ARM926 is responsible for the dynamic reconfiguration processes by executing the run-time mapping algorithm, which is used to determine a near-optimal mapping of the applications to the Montium cores. Although the authors discuss the possibility of heterogeneous parallelism exploitation in a multiprocessor environment, this work focuses only on speeding up DSP applications (e.g., FFT, FIR, and SISO algorithms).

In [11], the authors propose the employment of a shared reconfigurable logic, claiming that area and energy overhead are barriers when reconfigurable fabric is used as a private accelerator for each processing element of a multiprocessor design. Results of area and power reduction are demonstrated when sharing temporally and spatially the reconfigurable fabric. However, such approach relies on compiler support, precluding binary compatibility and affecting time-to-market due to larger design times.

In this work, we address the particular drawbacks of the above approaches by creating an adaptable reconfigurable multiprocessing system that

(i) unlike [9, 10], provides lower reconfiguration time, thus allowing ILP investigation/acceleration of the entire application code, including highly thread-parallel algorithms,

(ii) unlike [11], maintains binary compatibility through the application of a lightweight dynamic detection hardware that, at run-time, recognizes parts of code to be executed on the reconfigurable data path.

3. Analytical Model

In this section, we try to define the design space for multiprocessor-based architectures. First, we model a multiprocessing architecture (*MP—multiprocessor*) composed of many simple and homogeneous cores to elucidate the advantages of thread-level parallelism and compare its execution time (ET) to the modeling of a high-end single processor (*SHE—single high-end*) model with a great instruction-level parallelism exploration capability.

In the software point of view, we have used the amount of fine- (instruction) and coarse- (thread) level parallelism available in the application to investigate the performance potentials of both architectures. Considering a portion of code of a certain application, these software characteristics are denoted as

(i) α—can be executed in parallel in a single core,

(ii) β—cannot be executed in parallel in a single core,

(iii) δ—can be split among the cores of the multiprocessor environment,

(iv) γ—cannot be split among the cores of the multiprocessor environment.

Let us start with the basic equation relating execution time (ET) with instructions,

$$\text{ET} = \text{Instructions} * \text{CPI} * \text{CycleTime}, \tag{1}$$

where CPI is the mean number of cycles necessary to execute an instruction and Cycletime the operating frequency of the processor.

In this model, no information about cache accesses is considered, nor the performance of the disk or I/O is taken into account. Nevertheless, although simple, this model can provide interesting clues on the potential of multiprocessing architectures for a wide range of applications classes.

3.1. Low-End Single Processor. Based on (1), for a low-end single (*SLE—single low-end*) processor, the execution time can be written as

$$\text{ET}_{\text{SLE}} = \text{Instructions}\,(\propto \text{CPI}_{\text{SLE}} + \beta\text{CPI}_{\text{SLE}})\,\text{CycleTime}_{\text{SLE}}. \tag{2}$$

Since the low-end processor is a single-issue processor, it is not able to exploit ILP. Therefore, classifying instructions in α and β as previously stated does not make much sense. In this case, α is always equal to zero and β equal to one, but we will keep the notation and their meaning for comparison purposes.

3.2. High-End Single Processor. In the case of a high-end ILP exploitation architecture, based on (1) and (2), one can state that ET_{SHE} *(execution time of the high-end single processor)* is given by the following equation:

$$\text{ET}_{\text{SHE}} = \text{Instructions}\,(\propto \text{CPI}_{\text{SHE}} + \beta\text{CPI}_{\text{SLE}})\,\text{CycleTime}_{\text{SHE}}, \tag{3}$$

CPI_{SHE}, that also could be written as $\propto \text{CPI}_{\text{SLE}}$/issue (i.e., a high-end single processor would have the same CPI as the CPI of a low-end processor divided by the mean number of instructions issued per cycle), is usually smaller than 1, because a high-end single processor can exploit high levels of ILP, thanks to replication of functional units, branch prediction, speculative execution and mechanisms to handle false data dependencies, and so on. A typical value of CPI_{SHE} for a current high-end single processor is 0.62 [12], showing that more than one instruction can be executed per cycle. Thus, based on (3) one gets

$$\text{ET}_{\text{SHE}} = \text{Instructions}\left(\frac{\propto \text{CPI}_{\text{SLE}}}{\text{issue}} + \beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{SHE}}. \tag{4}$$

Issue represents the maximum number of instructions that can be issued in parallel to the functional units, considering the best-case situation: there are no data or control dependencies in the code. As already explained, coefficients α and β refer to the percentage of instructions that can be executed in parallel or not (this way, $\alpha + \beta = 1$), respectively. Finally, CycleTime_{SHE} represents the clock cycle time of the high-end single processor.

3.3. Homogeneous Multiprocessor Chip. Having stated the equation to calculate the performance of the high-end and low-end single processor, now the potential use of a homogeneous multiprocessing architecture, built by the replication of low-end processors, is studied. Such architecture does not heavily exploit the available ILP, but mostly the thread-level parallelism (TLP). Some works [13] propose an automatic translation of code with enough ILP into TLP, so that more than one core will execute the code. A multiprocessor environment is usually composed of low-end processor units, so that a large number of them can be integrated within the same die. Considering that each application has a certain number of instructions that can be split into several processors, one could write the following equation, based on (1) and (2):

$$\text{ET}_{\text{MP}} = \text{Instructions}\left(\frac{\delta}{P} + \gamma\right) \times (\alpha\text{CPI}_{\text{SLE}} + \beta\text{CPI}_{\text{SLE}})\,\text{CycleTime}_{\text{MP}}, \tag{5}$$

where δ is the amount of code that can be transformed into multithreaded code, while γ is the part of the code that must be executed sequentially (no TLP is available). P is the number of low-end processors that is available in the chip. Hence, the second term of (5) reflects the fact that in a multiprocessor environment one could benefit from thread-level parallelism, since increasing the number of processors will only accelerate parts of the code that can be parallelized at thread level.

3.4. High-End Single Processor versus Homogeneous Multiprocessor Chip. Based on the above reasoning, one can compare the performance of the high-end single processor

to the multiprocessor environment. However, one important aspect is that the several low-end processors that compose the homogeneous multiprocessor design could also run at much higher frequencies than high-end processors, since their simple organizations reflect smaller area and power consumption. However, the total power budget will probably be the limiting performance factor for both designs. For the sake of the model, we will assume that

$$\text{CycleTime}_{\text{MP}} = K * \text{CycleTime}_{\text{SHE}}, \tag{6}$$

where K is the frequency adjustment factor to normalize the power consumption of both homogeneous multiprocessor and the high-end single processor.

Thus, the comparison of both architectures, based on (3) and (5), is given by

$$\frac{\text{ET}_{\text{SHE}}}{\text{ET}_{\text{MP}}} = \frac{\left[\text{Instructions}\left(\propto(\text{CPI}_{\text{SLE}}/\text{issue})+\beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{SHE}}\right]}{\left[\text{Instructions}\left(\delta/P+\gamma\right)\left(\propto\text{CPI}_{\text{SLE}}+\beta\text{CPI}_{\text{SLE}}\right)\text{CycleTime}_{\text{MP}}\right]}. \tag{7}$$

By simplifying and merging (6) and (7), one gets

$$\frac{\text{ET}_{\text{SHE}}}{\text{ET}_{\text{MP}}} = \left[\frac{1}{\delta/P}+\gamma\right]\left[\frac{\propto(\text{CPI}_{\text{SLE}}/\text{issue})+\beta\text{CPI}_{\text{SLE}}}{\propto\text{CPI}_{\text{SLE}}+\beta\text{CPI}_{\text{SLE}}}\right]\left[\frac{1}{K}\right]. \tag{8}$$

From (8) one can notice that the high-end processor is faster than multiprocessor architecture when $(\text{ET}_{\text{SHE}}/\text{ET}_{\text{MP}}) < 1$. In addition, this equation shows that, although the multiprocessor architecture with low-end simple processors could have a faster cycle time (by a factor of K), that factor alone is not enough to define performance. Regarding the second term between brackets in (8), the fact that the high-end processor can execute many instructions in parallel could give a better performance. Since there is no instruction-level parallelism exploration in a low-end single processor, it means that the term $\propto \text{CPI}_{\text{SLE}}$ is always zero.

In the extreme case, let us imagine that issue $= P = \infty$, meaning that we have infinite resources, either in the form of arithmetic operators or in the form of processors. This would reduce (8) to

$$\frac{\text{ET}_{\text{SHE}\infty}}{\text{ET}_{\text{MP}\infty}} = \left[\frac{1}{\gamma}\right]\left[\frac{\beta\text{CPI}_{\text{SLE}}}{\propto\text{CPI}_{\text{SLE}}+\beta\text{CPI}_{\text{SLE}}}\right]\left[\frac{1}{K}\right]. \tag{9}$$

Equation (9) clearly shows that, as long as one has code which carries control or data dependencies, and cannot be parallelized (at the instruction or thread level), a machine based on a high-end single core will always be faster than a multiprocessor-based machine, regardless of the amount of available resources.

Another interesting experiment is to try to equal the performance of the high-end single core and the performance of the multiprocessor core. This way, let us consider that $T_{\text{SHE}} = T_{\text{MP}}$; hence,

$$\left[\left(\propto\text{CPI}_{\text{SHE}}+\beta\text{CPI}_{\text{SLE}}\right)\frac{1}{K}\right] = \left[\left(\frac{\delta}{P}+\gamma\right)\left(\alpha\text{CPI}_{\text{SLE}}+\beta\text{CPI}_{\text{SLE}}\right)\right]. \tag{10}$$

From (10), one can see that one must have enough low-end processors combined to a highly parallel code (greater δ) to overcome the high-end processors advantage. This statement is clarified by the fact that the term $\propto \text{CPI}_{\text{SLE}}$ is always zero, imposing that β is equal to one and CPI_{SLE} is much higher than CPI_{SHE}.

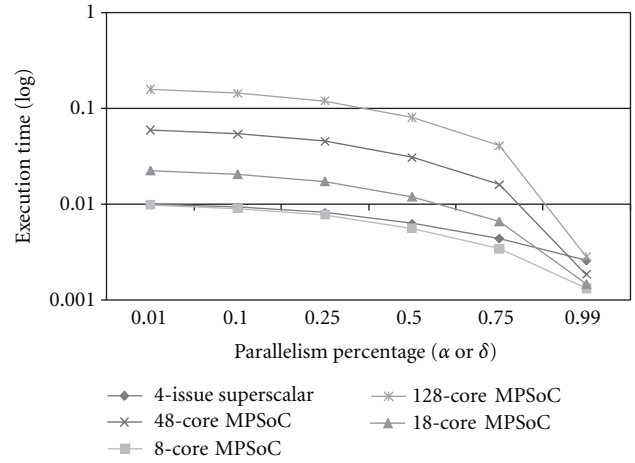


Figure 1: Multiprocessor system and superscalar performance regarding a power budget using different ILP and TLP; $\alpha = \delta$ is assumed.

3.5. Applying the Analytical Model in Real Processors. Given the theoretical model, one can briefly test it with some numbers based on real data. Let us consider a high-end single core: a 4-issue MIPS R10000 superscalar processor, with CPI equal to 0.6 [14] and a multiprocessor design composed of low-end MIPS R3000 processors, with CPI equal to 1.3 each [15]. A comparison between both architectures is done using the equations of the aforementioned analytical model. Figure 1 shows, in a logarithmic scale, the performance of the superscalar processor when parameters α and β change. In addition, in Figure 1 we also show the performance of the multiprocessor design, varying the δ and γ parameters and the number of processors from 8 to 128. To provide a better view of the performance considering both approaches, the x-axis of Figure 1 represents the amount of the instruction- (α) and thread- (δ) level parallelism in the application, where α is only valid for the 4-issue superscalar, while δ is valid for all the MPSoC's setups.

The goal of this comparison is to demonstrate which technique better explores its particular parallelism at different levels, considering six values for both ILP and TLP. For instance, $\delta = 0.01$ means that an application only shows 1% of thread-level parallelism within its code (valid only for

the MPSoC's examples). In the same way, when $\alpha = 0.01$, it is assumed that 1% of instruction-level parallelism (ILP) is available. That is, only 1% of its instructions can be executed in parallel in the 4-issue superscalar processor. Following the same strategy found in current processor designs, for a fair performance comparison, we considered the same power budget for the high-end single core and the multiprocessor approaches. In order to normalize their power budget, we have tuned the frequency adjustment factor K of (5). For that, we fixed the 4-issue superscalar frequency to use it as the power reference, changing the K factor of the remaining approaches to achieve the same consumption as the reference. Thus, the frequency of the 8-core MPSoC must be 3 times higher than 4-issue superscalar processor. For the 18-core, such value must be a quarter higher than the reference value. Since a considerable number of cores employed in the 48-core MPSoC setup, this approach should execute 2 times slower than the 4-issue superscalar processor to achieve the same power consumption. Finally, the frequency of the 128-core MPSoC must be 5.3 times smaller than the 4-issue superscalar to respect the same power budget.

The leftmost side of Figure 1 considers any given application that has a minimum amount of instruction- ($\alpha = 0.01$) and thread- ($\delta = 0.01$) level parallelism available. In this case, the superscalar processor and the 8-core design present almost the same performance. However, considering the same power budget for all approaches by using different operating frequencies shown before, when applications show greater parallelism percentage ($\alpha > 0.25$ and $\delta > 0.25$), the 8-core design achieves better performance with TLP exploitation than the 4-issue superscalar processor with ILP exploitation.

When more cores are added in a multiprocessor design, the overall clock frequency tends to decrease, since the adjustment factor of (5) should be smaller to obey the power budget. In this way, the performance of applications that present low-thread-level parallelism (small δ) worsens when increasing the number of cores. Regarding the applications with $\delta = 0.01$ in Figure 1, performance is significantly decreased as the number of cores increases. Nevertheless, as the application thread-level parallelism increases (i.e., $\delta > 0.01$), the negative impact on performance is softened, since the additional cores will have better use.

Aiming to make a fairer performance comparison among high-end single core and multiprocessor approaches, we have devised an 18-core design composed of low-end processors that, besides presenting the same power consumption due to the power budget assumed, also has the same area of the 4-issue superscalar processor. For that, we considered that the MIPS R3000 takes only 75.000 transistors [16], almost 29 times less than the 2.2 millions of transistors spent on the MIPS R10000 design [17]. Furthermore, for a reasonable comparison, we also considered that the intercommunication mechanism would take nearly 37% of the chip area, as reported in [18]. The performance of both approaches shows the powerful capabilities of the superscalar processor. Regarding the same area and power for both designs, as shown in Figure 1, the multiprocessor approach (18-core MPSoC) only surpasses the superscalar (4-issue superscalar) performance when the TLP level is greater than 85% ($\delta > 0.85$).

Summarizing the comparison with the same power budget, the superscalar machine shows better performance over applications with low-thread-level parallelism. On the other hand, there is an additional tradeoff that must be considered regarding multiprocessor designs, since, when more cores are included in the chip, the multiprocessor performance tends to worsen, since the operating frequency must be decreased to respect the power budget limits. When almost the whole application presents high TLP ($\delta > 0.99$), the 128-core design takes longer execution time than the other multiprocessor designs since its operating frequency is very low.

Considering real applications, thread-level parallelism exploitation is widespread employed to accelerate most multimedia and DSP applications thanks to their data independent iteration loops. However, even applications with high TLP could still obtain some performance improvement by also exploiting ILP. Hence, in a multiprocessor design, ILP techniques also should be investigated to conclude what is the best fit concerning the design requirements. Hence, the analytical model indicates that heterogeneous multiprocessor system is necessary to balance the performance of a wide range of application classes. Section 6 reinforces this trend running real applications over a multiprocessor design coupled to an adaptable ILP exploitation approach named DIM technique.

4. Reconfigurable Multiprocessing System

Section 3 demonstrated that in a heterogeneous application environment, TLP and ILP exploitation are complementary. This way, it is necessary to explore different grains of parallelism to balance performance. Aiming to support this statement, we have built a multiprocessor structure shown in Figure 2(a) to reproduce the analytical model shown in Section 3 by executing well-known applications. The architecture in the example is composed of four cores, so TLP exploitation is guaranteed. However, as ILP exploitation is also mandatory, we have coupled a coarse-grain reconfigurable data path to each one of the cores, since the use of reconfigurable fabric has already shown great speedups with low-energy consumption [6, 8] concerning single thread applications.

Figure 2(b) shows in detail the microarchitecture of the processor, named as reconfigurable core (RC), used as the base processing element of the reconfigurable multiprocessing system. To better explain the RC processor, we divided such architecture in 4 blocks. Block 1 depicts the reconfigurable data path that aggregates the input context, output context, and the functional units. Block 2 presents the basic SparcV8 like five-stage pipelined processor. Block 3 illustrates the pipeline stages of the dynamic instruction merging (DIM) [8] technique that works in parallel to the processor pipeline. It is responsible for transforming instruction blocks into configurations of the reconfigurable data

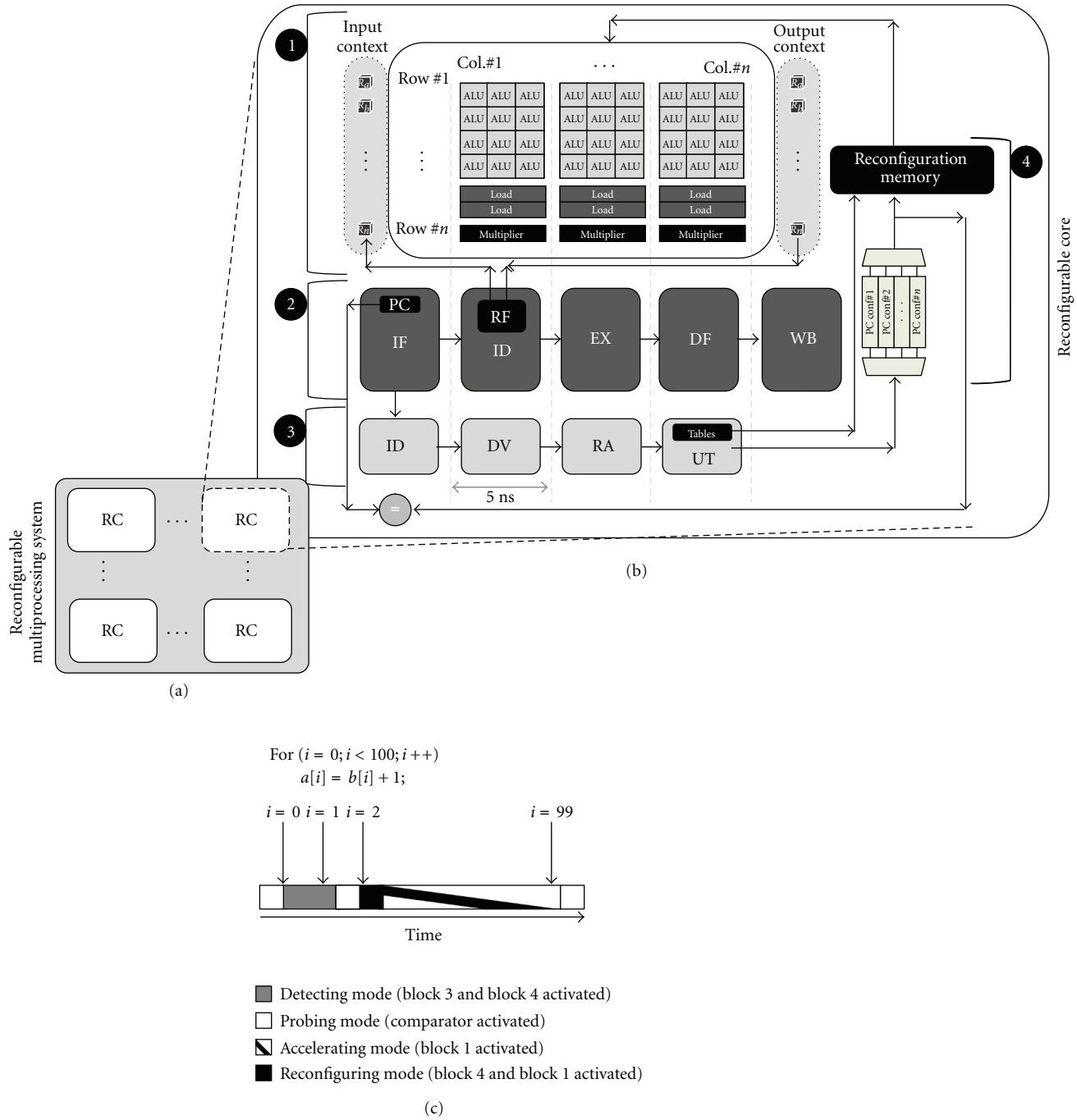


Figure 2: (a) Multiprocessing system. (b) Reconfigurable core. (c) Example of a loop optimization.

path at run time. Block 4 demonstrates the reconfiguration memory and the address cache. The reconfiguration memory holds the configuration bits previously generated by the DIM, so next time when the same translated sequence is found, the configuration bits are reused. The address cache (4-way associative) is responsible for keeping the first PC address of each translated sequence. More details about these components will be presented in the next sections.

Figure 2(c) shows an example of how a loop would be accelerated using the proposed process. The reconfigurable core works in four modes: probing, detecting, reconfiguring, and accelerating. At the beginning of the time bar shown in Figure 2(c), the RC is searching for an already translated configuration to accelerate through execution in the reconfigurable data path.

However, when the first loop iteration appears ($i = 0$), the DIM detects that there is a new code to translate and it changes to detecting mode. In that mode, while the instructions are executed in the processor pipeline, they are also translated to a configuration by the DIM. When the second

loop iteration is found ($i = 1$), the DIM is still finishing building the current configuration (that started when $i = 0$) and storing it into the reconfiguration memory.

Then, when the first instruction of the third loop iteration comes to the fetch stage of the processor pipeline ($i = 2$), the probing mode detects a valid configuration in the reconfiguration memory, since the previously started detection process is now finished and the memory address of the first instruction of the translated sequence was found in the address cache.

Therefore, the RC enters in reconfiguring mode to feed the reconfigurable data path with the operands and the reconfiguration bits. Finally, the accelerating mode is activated and the next loop iterations (until the 99th) are efficiently executed, taking advantage of the reconfigurable logic.

4.1. Reconfigurable Data Path Structure (Block 1). Following the classifications shown in [19, 20], the reconfigurable data path is tightly coupled to the processor pipeline. Such coupling approach avoids external accesses to the memory, saving power and reducing the reconfiguration time. Moreover, its coarse-grained nature decreases the size of the memory necessary to keep each configuration, since the basic processing elements are functional units that work at the word level (arithmetic and logic, memory access, and multiplier). The data path is organized as a matrix of rows and columns, composed of functional units. Three columns of arithmetic and logic units (ALUs) compose a level. A level does not affect the SparcV8 critical path (which, in this case, it is given by the register file). The number of basic rows dictates the maximum instruction-level parallelism that can be exploited, since instructions placed in the same column are executed concurrently (in parallel). The example of the data path shown in Figure 2(b) could execute up to four arithmetic and logic operations, two memory accesses (two memory ports are available), and one multiplication in parallel. The number of rows, in turn, determines the maximum number of dependent instructions placed into one configuration. Both the number of rows and the number of parallel components can be modified according to the application requirements and the design constraints. It is important to notice that simple arithmetic and logic operations can be executed within the same processor cycle without affecting the critical path. Consequently, data-dependent instructions are also accelerated. Memory accesses and multiplications take one equivalent processor cycle to perform their operations.

The entire structure of the reconfigurable data path is purely combinational: there is no temporal barrier between the functional units. The only exception is for the entry and exit points. The entry point is used to keep the input context, which is connected to the processor register file. The fetching of the operands from the register file is the first step to configure the data path before actual execution. After that, results are stored in the output context registers through the exit point of the data path. The values stored in the output context are sent to the processor register file on demand. It means that if a given result is produced at any level and it will not be changed in the subsequent levels, its value is written back at the same level that it was produced. In the current implementation, the reconfigurable system provides two write-backs per level.

We have coupled sleep transistors [18] to switch power on/off of each functional unit in the reconfigurable data path. The dynamic reconfiguration process is responsible for the sleep transistors management. Their states are stored in the reconfiguration memory, together with the reconfiguration data. Thus, for a certain configuration, idle functional units are set to the off state, avoiding leakage or dynamic power dissipation, since the incoming bits do not produce switching activity in the disconnected circuit. Although the sleep transistors are bigger and in series to the regular transistors used in the implemented circuit, they have been designed so that their delays do not significantly impact the critical path or the reconfiguration time.

4.2. Processor Pipeline (Block 2). A SPARC-based architecture is used as the baseline processor to work together with the reconfigurable system. Its five-stage pipeline reflects a traditional RISC architecture (instruction fetch, decode, execution, data fetch, and write-back). The microarchitecture and the performance of such processor are very similar to the MIPS R3000, considered as the low-end processor in the analytical model shown in Section 3.

4.3. Dynamic Instruction Merging (Block 3). As explained before, the dynamic instruction merging (DIM) can work in four modes: detecting, probing, accelerating, and reconfiguring. As can be observed in Figure 2(b), the hardware responsible for the detecting mode contains four pipeline stages.

(i) Instruction decode (ID): the instruction is broken into operation, source operands, and target operand.

(ii) Dependence verification (DV): the source operands are compared to the target operands of previously detected instructions to verify which column the current instruction should be allocated, according to their data dependencies. The placement algorithm is very simple: the DV stage only indicates the leftmost column that the current instruction should be placed.

(iii) Resource allocation (RA): in this stage, the data dependence is already solved and the correct data path column is known. Hence, the RA stage is responsible for verifying the resources availability in that column, linking the instruction operation to the correct type of functional unit. If there is no functional unit available at this column, the next column at the right side will be checked. This process is repeated until finding a free functional unit.

(iv) Update tables (UT): this stage configures the routing to feed that functional unit with the correct source operands from the input context and to write the result in the correct register of the output context. After that, the bitmaps and tables are updated and

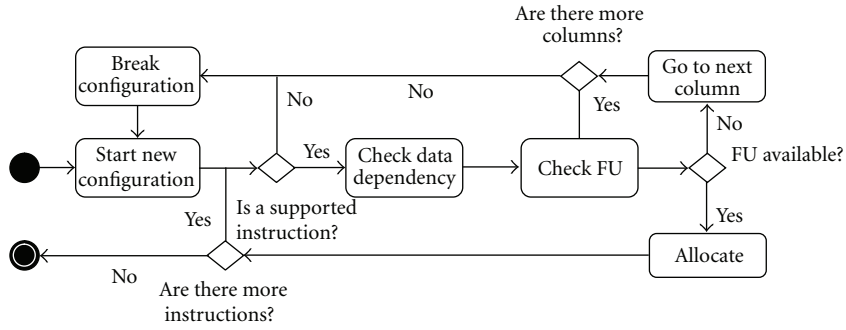


Figure 3: DIM activity diagram.

the configuration is finished: their configuration bits are sent to the reconfiguration memory and the address cache is updated.

Figure 3 illustrates, by an activity diagram, the whole DIM process to create a configuration. The first step is the execution support verification. If there is no compatible functional unit to execute such an operation (e.g., division), the configuration is finished and the next instruction is a candidate to start a new configuration. On the other hand, if there is support, the data dependency among previously allocated instructions is verified (DV stage) and the correct functional unit within that column is defined. Then, the current configuration is sent to the reconfiguration memory.

4.4. Storage Components (Block 4). Two storage components are part of the reconfigurable system: address cache and reconfiguration memory. The configurations are indexed by the address of the first instruction of the translated sequence and kept in the address cache, a 128-entry 4-way associative cache. The Address Cache is only accessed when the DIM is working in the probing mode. An address cache hit indicates that a configuration was found, changing the system to the reconfiguring mode. In this mode, using the pointer given by the address cache, the reconfiguration memory is accessed to feed the data path routing. The reconfiguration memory stores the routing bits and the necessary information (such as the input and output contexts and immediate values) to fire a configuration. Finally, the DIM hardware changes to accelerating mode, beginning the execution process in the reconfigurable data path.

5. Simulation Environment

5.1. Workload. A workload of only high parallel applications with distinct behaviors was chosen, using benchmarks from the well-known SPLASH2 [21] and PARSEC [22] suites. In addition, two numerical applications written in OpenMP were used [23].

The list below briefly describes each of them.

(i) *FFT* [21]. It is a complex 1D version of a six-step FFT algorithm.

(ii) *LU* [21]. It factors a dense matrix in the equivalent lower-upper matrix multiplication.

(iii) *Blackscholes* [22]. It solves the partial differential equation of *Blackscholes* in order to compute prices for a portfolio of European options.

(iv) *Swaptions* [22]. Monte Carlo simulation is used to price a portfolio of *swaptions*.

(v) *Molecular Dynamics* (*MD*) [4]. It implements the velocity Verlet algorithm for molecular dynamics simulation.

(vi) *Jacobi* [23]. It utilizes the Jacobi iterative method to solve a finite difference discretization of Helmholtz.

5.2. TLP and ILP Exploration Opportunities. In this section we show the opportunities for coarse- and fine-grain parallelism exploration in the selected benchmarks shown in Section 5.1. The experiments addressing these applications were done in a SparcV8 architecture varying the number of threads from 1 to 64.

The mean size of the basic blocks (BB) of an application is an important aspect to define its fine grain parallelism level since the room for most ILP exploration techniques relies on this characteristic. The second column of Table 1 presents the mean BBs size of the selected applications. As it can be noticed, even parallel applications provide great room for instruction-level parallelism exploration. The remaining columns in Table 1 show, in percentage, the load balancing between threads of the selected applications. As expected, most applications provide a perfect load balancing up to 64 threads. *FFT* and *LU* do not follow the trend of the other applications, since the load balancing decreases as the number of threads increases.

Therefore, even applications with perfect load balancing (e.g., *swaptions*) provide a great room for instruction-level parallelism since their basic blocks have enough instructions to be parallelized. In the same way, applications with poor load balancing, where probably thread-level parallelism exploration will not be enough to achieve satisfactory performance improvements (e.g., *lu* with a great number of threads), can benefit even more from instruction-level parallelism exploitation. In this way, one can conclude that

Table 1: ILP and TLP opportunities for the selected benchmarks.

Benchmark	Mean BB size (#instr)	Load balancing (%)			
		4 threads	8 threads	16 threads	64 threads
Swaptions	5.92	99.00	99.00	99.00	98.00
Blackscholes	4.83	99.00	99.00	99.00	98.00
MD	6.51	95.04	83.24	88.92	89.87
Jacobi	6.94	97.02	97.02	92.07	93.12
FFT	8.10	69.39	49.50	31.96	24.58
LU	8.32	81.20	56.77	29.35	7.03

Table 2: Number of basic functional units of setups.

	RC#1	RC#2
Columns	48	150
ALU/column	6	8
Load/column	4	6
Mul/column	1	2

Table 3: Average speedup on different number of cores.

	4 cores	8 cores	16 cores	64 cores
TLP	3.74	6.86	12.47	44.3
ILP + TLP	6.46	11.85	21.71	51

the mixed parallelism exploitation is mandatory even for applications where the thread-level parallelism is dominant.

5.3. Methodology. To simulate the reconfigurable multiprocessor system, we have used the scheme presented in [24]. It consists of a functional full system [25] that models the SparcV8 architecture and cycle accurate timing simulators [26] that reproduce the behavior of the individual reconfigurable cores depicted in Figure 2(b). Since the applications are split automatically by the OpenMP and the Posix Threads API, the cycle accurate simulator gives special attention to synchronization mechanisms, such as locks and barriers. Therefore, the elapsed time regarding blocking synchronization and memory transfers are precisely measured.

For all experiments, we have tuned the number of reconfigurable cores based on the number of threads used to run the applications presented in Section 5.1.

To demonstrate the impact of ILP exploitation in the performance, we have used two different configurations for the reconfigurable data path (block 1 of Figure 2(b)), changing its number of basic functional units. The setups, shown in the Table 2, have already presented the best tradeoff considering area and performance executing single-thread applications [8].

6. Results

6.1. Performance Results. This section demonstrates the performance evaluation of the reconfigurable multiprocessing system over three different aspects:

(i) TLP exploitation by changing the number of cores from four up to 64 (in these experiments, stand-alone SPARC cores are used: they are not coupled to the reconfigurable architecture);

(ii) TLP + ILP exploitation, repeating the previous experiment but now using the SPARC cores together with the reconfigurable architecture in the two different versions (RC#1 or RC#2);

(iii) the influence of changing the applications' data set sizes on performance.

Figure 4 explores the first two aspects discussed above: TLP exploitation only, varying the number of stand-alone SPARC cores (solid bars) and TLP + ILP exploitation, by coupling the reconfigurable architecture (RC#1) to each one of the cores (striped bars). Regarding the former, performance scales linearly as the number of cores increases. *FFT* and *LU* do not follow this behavior, since their codes, as shown in Table 1, do not present perfect load balancing as other applications.

As can be observed, the results reinforce the conclusion gathered from the analytical model in Section 3: even for high TLP-based applications, there is a need for finer-grain parallelism exploitation to complement the TLP gains. Table 3 shows the average speedup of both approaches. This table demonstrates that TLP+ILP exploitation, using the RC#1 setup composed of four cores, presents similar performance gains comparing to the eight standalone cores exploiting only TLP parallelism. The same occurs when comparing a system with 8 cores and the RC#1 setup to 16 stand-alone cores.

Figure 5 compares the performance of a system composed of 4 or 8 cores, in which each is coupled to the RC#2 over the system in which the cores are coupled to the RC#1. The improvement is negligible, and not proportional to the additional number of basic functional units. This happens because of the high TLP degree presented in the selected workload. Their threads do not present enough instructions that can be accelerated by the additional basic functional units available in RC#2, so the amount of basic functional units of RC#1 is adequate to satisfactorily explore the ILP available in most applications of the selected workload. *Molecular dynamics* (*MD*) is the one that best takes advantage of the extra units of the RC#2, although it presents only 5% of performance improvements than RC#1. *Jacobi* and *LU* executions show performance loss when using the RC#2

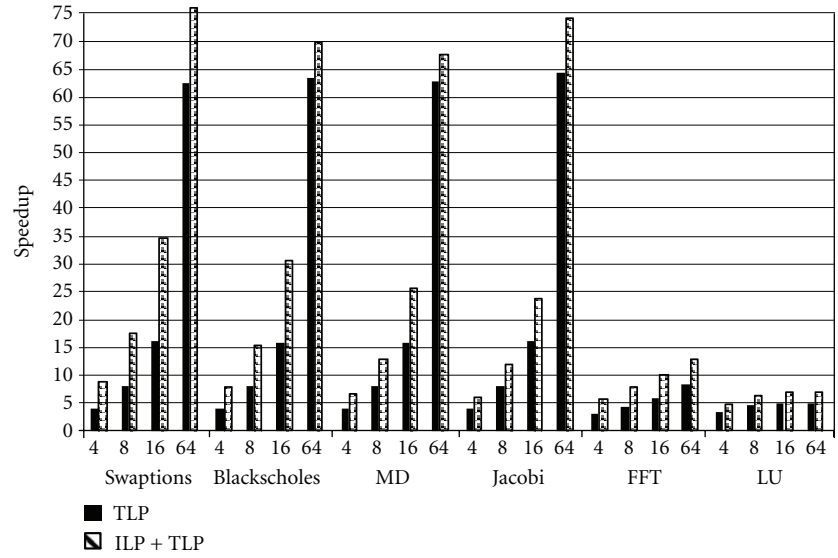


Figure 4: Performance of TLP and ILP + TLP exploration using RC#1 setup.

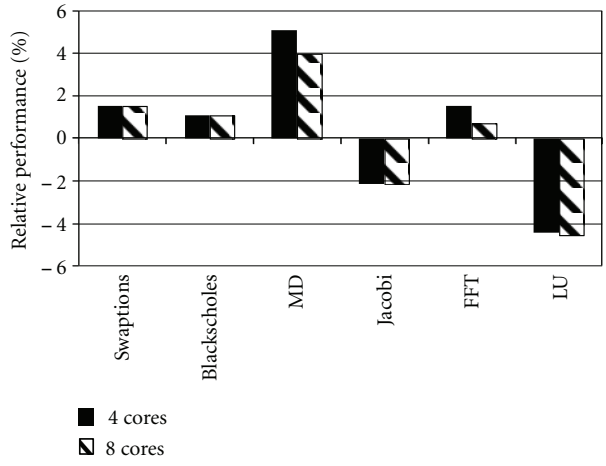


Figure 5: Performance comparison among RC#1 and RC#2 setups.

setup. It happens because the DIM produces a different amount of configurations for both setups. Due to the dynamic behavior of DIM technique, it affects the address cache storage producing more configuration misses in the RC#2 setup. Therefore, since RC#2 does not show any significant advantage over the RC#1, the RC#1 will be used for the remaining experiments.

Figure 6 shows the performance evaluation when running the same application workload with two different data set sizes, so it is possible to demonstrate that changing the data set size does not affect the performance results shown in Figure 4.

However, *FFT* and *LU* present a significant impact when changing the data set size. Figure 7 shows this data in more detail. *FFT* has a significant amount of sequential code responsible for data initialization. Thus, when we increase the data set size, the initialization becomes more significant over the whole application execution time. This behavior is more evident in the multiprocessing system composed of 16 cores.

Regarding *LU*, the larger data set size provides a perfect load balance with a great number of processors [21], as observed in Figure 7. On the other hand, smaller data set sizes increase the imbalance by splitting less blocks per processor in each step of the factorization.

6.2. Energy Results Considering the Same Power Budget. In Section 3.4, we have created a power budget to clarify the advantages/disadvantages of instruction- and thread-level parallelism exploitation. This way, we have also evaluated the energy consumption of the selected benchmarks considering the same power budget for both parallelism exploitations (ILP and TLP+ILP). The power dissipation of the stand-alone SparcV8 is 385.14 mWatts and the reconfigurable core consumes 699.33 mWatts. Therefore, we have compared the 8-core SparcV8 with the reconfigurable multiprocessing system composed of 4 reconfigurable cores, since both reach nearly 3 Watts of power dissipation. In addition, these setups will help us to measure the contribution of the proposed approach in reducing energy consumption, since, as shown in Table 3, both setups provide almost the same performance. Due to the high simulation time, we choose three benchmarks from the application workload to show the energy consumption of both approaches. This application subset contains massive thread-level parallelism applications (MD and Jacobi) as well as application that shows considerable load unbalancing (LU).

Despite the fact that all applications provide massive thread-level parallelism, since their performance scales linearly as the number of cores increases, the proposed approach consumes less Power than the multiprocessing system composed of stand-alone SparcV8 in all benchmarks evaluated. Energy savings are possible because, although the power consumption of reconfigurable core is the same as

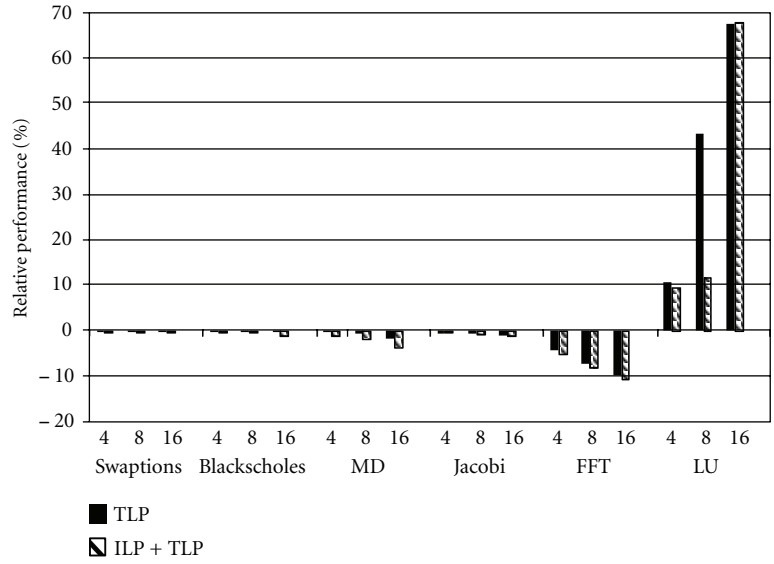


Figure 6: Performance comparison regarding different application data set sizes.

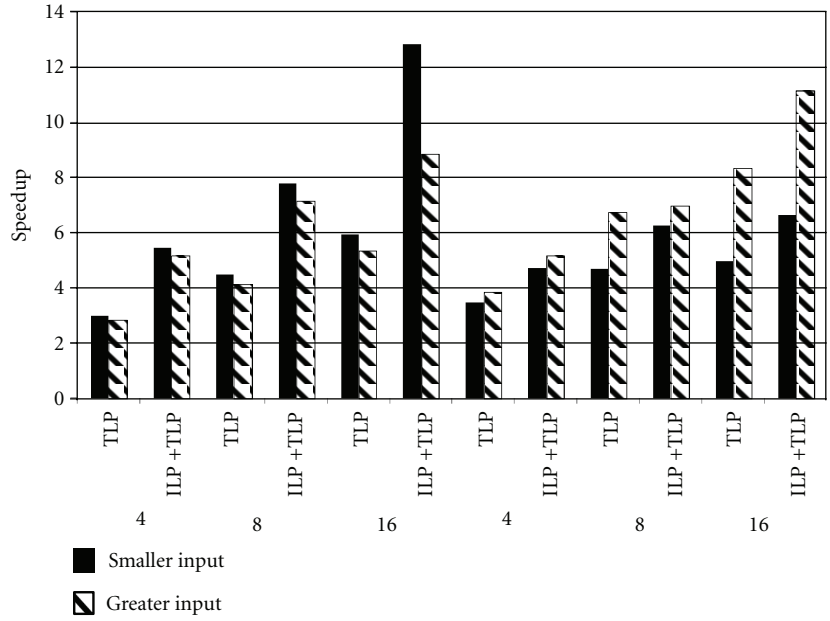


Figure 7: FFT and LU performance regarding different application data set sizes.

the power presented in the multiprocessing system composed of stand-alone SparcV8, the average power is lower, mainly thanks to the use of sleep transistors to turn off idle functional units of the reconfigurable data path. In addition, although more power is spent because of the DIM hardware and reconfigurable data path, total average power is reduced since there are fewer memory accesses for instructions: once they were translated to a data path configuration, they will reside in the reconfiguration memory.

Disregarding the power budget proposed in this section, we can compare the energy consumption of the 8-core SparcV8 with the multiprocessing system composed of eight reconfigurable cores. As can be seen in Table 4, the proposed approach outperforms the 8-core SparcV8, on average, by 72% and still consumes 42% less energy. The main source, besides the already mentioned, is the shorter execution time of the mixed parallelism exploration.

7. Conclusions

This paper demonstrated that, although the instruction-level parallelism (ILP) exploitation is reaching its limits and multiprocessing system appears as a solution to accelerate applications by exploring coarse grains of parallelism, there are significant sequential parts of code that still must be handled by ILP exploitation mechanisms. Therefore,

Table 4: Energy consumption, in mJ, of both multiprocessing system and reconfigurable system on running MD, Jacobi and LU.

	TLP				ILP + TLP			
	4 cores	8 cores	16 cores	64 cores	4 cores	8 cores	16 cores	64 cores
MD	0.282	0.329	0.353	0.376	0.185	0.216	0.232	0.248
Jacobi	99.1	115.7	124.3	132.3	69.6	81.3	87.3	93.1
LU	0.167	0.194	0.232	0.330	0.113	0.132	0.157	0.227
Average	33.2	38.8	41.6	44.3	23.3	27.2	29.2	31.2

there is the need of mixed-grain parallelism exploitation to achieve balanced performance improvements even for applications that present dominant thread-level parallelism. This paper presented such system: an adaptable ILP exploitation mechanism, using reconfigurable logic, coupled to a multiprocessing environment.

References

[1] D. W. Wall, "Limits of instruction-level parallelism," *ACM SIGPLAN Notices*, vol. 26, no. 4, pp. 176–188, 1991.

[2] J. Mak and A. Mycroft, "Limits of instruction data dependence graphs," in *Proceedings of the 7th International Workshop on Dynamic Analysis (WODA '09)*, Chicago, Ill, USA, July 2009.

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

[4] S. J. Patel and S. S. Lumetta, "rePLay: a hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 590–608, 2001.

[5] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pp. 12–21, IEEE Computer Society, Napa Valley, Calif, USA, 1997.

[6] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," in *Proceedings of the 41st Annual Conference on Design Automation (DAC '04)*, pp. 659–681, ACM, New York, NY, USA, 2004.

[7] K. Olukotun, *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, Morgan and Claypool Publishers, 1st edition, 2007.

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

[9] G. Stitt and F. Vahid, "Thread warping: a framework for dynamic synthesis of thread accelerators," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, ACM, Salzburg, Austria, September-October 2007.

[10] G. J. Smit, A. B. Kokkeler, P. T. Wolkotte, and M. D. van de Burgwal, "Multi-core architectures and streaming applications," in *Proceedings of the International Workshop on System Level Interconnect Prediction (SLIP '08)*, pp. 35–42, ACM, Newcastle, UK, April 2008.

[11] M. A. Watkins, M. J. Cianchetti, and D. H. Albonesi, "Shared reconfigurable architectures for CMPS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 299–304, September 2008.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop Workload Characterization (WWC '01)*, pp. 3–14, Washington, DC, USA, December 2001.

[13] Y. Song, S. Kalogeropulos, and P. Tirumalai, "Design and implementation of a compiler framework for helper threading on multi-core processors," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*, pp. 99–109, IEEE Computer Society, Washington, DC, USA, September 2005.

[14] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the MIPS R10000 performance counters," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, p. 16, IEEE Computer Society, Pittsburgh, Pa, USA, January 1996.

[15] T. Roirdan, G. P. Grewal, S. Hsu et al., "System design using the MIPS R3000/3010 RISC chipset," in *Proceedings of the 34th IEEE Computer Society International Conference on Intellectual Leverage, Digest of Papers (COMPCON '89)*, pp. 494–498, San Francisco, Calif, USA, 1989.

[16] C. Rowen, M. Johnson, and P. Ries, "The MIPS R3010 floating-point coprocessor," *IEEE Micro*, vol. 8, no. 3, pp. 53–62, 1988.

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

[18] http://blogs.intel.com/research/2007/07/inside_the_terascale_many_core.php.

[19] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[20] A. C. Beck and L. Carro, *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques*, Springer, New York, NY, USA, 2009.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pp. 24–36, ACM, S. Margherita Ligure, Italy, June 1995.

[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 72–81, ACM, Toronto, Canada, October 2008.

[23] A. J. Dorta, C. Rodriguez, F. D. Sande, and A. Gonzalez-Escribano, "The OpenMP source code repository," in *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '05)*, pp. 244–250, IEEE Computer Society, Washington, DC, USA, February 2005.

[24] M. Monchiero, J. Ahn, A. Falcón, D. Ortega, and P. Faraboschi, "How to simulate 1000 cores," *ACM SIGARCH Computer Architecture*, vol. 37, no. 2, pp. 10–19, 2009.

[25] P. S. Magnusson, M. Christensson, J. Eskilson et al., "Simics: a full system simulation platform," *Computer*, vol. 35, no. 2, pp. 12–58, 2002.

[26] M. B. Rutzig, A. C. Beck, and L. Carro, "Dynamically adapted low power ASIPs," in *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, J. Becker, R. Woods, P. Athanas, and F. Morgan, Eds., vol. 5453 of *Lecture Notes In Computer Science*, pp. 110–122, Springer, Karlsruhe, Germany, March 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 406857, 17 pages
doi:10.1155/2011/406857

Research Article

High-Level Synthesis of In-Circuit Assertions for Verification, Debugging, and Timing Analysis

John Curreri, Greg Stitt, and Alan D. George

NSF Center for High-Performance Reconfigurable Computing (CHREC), ECE Department, University of Florida, Gainesville, FL 32611-6200, USA

Correspondence should be addressed to John Curreri, curreri@hcs.ufl.edu

Received 13 August 2010; Accepted 14 December 2010

Academic Editor: J. M. P. Cardoso

Despite significant performance and power advantages compared to microprocessors, widespread usage of FPGAs has been limited by increased design complexity. High-level synthesis (HLS) tools have reduced design complexity but provide limited support for verification, debugging, and timing analysis. Such tools generally rely on inaccurate software simulation or lengthy register-transfer-level simulations, which are unattractive to software developers. In this paper, we introduce HLS techniques that allow application designers to efficiently synthesize commonly used ANSI-C assertions into FPGA circuits, enabling verification and debugging of circuits generated from HLS tools, while executing in the actual FPGA environment. To verify that HLS-generated circuits meet execution timing constraints, we extend the in-circuit assertion support for testing of elapsed time for arbitrary regions of code. Furthermore, we generalize timing assertions to transparently provide hang detection that back annotates hang occurrences to source code. The presented techniques enable software developers to rapidly verify, debug, and analyze timing for FPGA applications, while reducing frequency by less than 3% and increasing FPGA resource utilization by 0.7% or less for several application case studies on the Altera Stratix-II EP2S180 and Stratix-III EP3SE260 using Impulse-C. The presented techniques reduced area overhead by as much as 3x and improved assertion performance by as much as 100% compared to unoptimized in-circuit assertions.

1. Introduction

Field-programmable gate arrays (FPGAs) show significant power and performance advantages as compared to microprocessors [1], but have not gained widespread acceptance largely due to prohibitive application design complexity. High-level synthesis (HLS) significantly reduces application design complexity by enabling applications written in a high-level language (HLL) such as C to be executed on FPGAs. However, limited HLS support for verification, debugging, and timing analysis has contributed to limited usage of such tools.

For verification, designers using HLS can use assertion-based verification (ABV), a widely used technique in electronic design automation (EDA) tools [2], to verify runtime behavior by executing an application that contains assertions against a testbench. However, assertion-based verification of programs written in C using HLS tools, such as Impulse-C [3] and Carte [4], is often limited to software simulation of the FPGA's portion of the code, which can be problematic due to common inconsistencies between simulated behavior and actual circuit behavior. Such inconsistencies most commonly result from timing differences between the software thread-based simulation of the circuit and the actual FPGA execution [5]. In some cases, these inconsistencies may cause an application that behaves normally in software simulation to never complete (i.e., hang) when executing on the FPGA. Debugging an HLS-generated circuit to identify the cause of such hangs is a significant challenge that currently requires excessive designer effort.

Timing analysis, a procedure which determines if performance constraints are met, is an additional limitation of many HLS tools. Although timing analysis is widely used in physical design tools, in many cases, HLS tools do not consider timing constraints. Even worse, designers are unaware of the performance of different regions of an HLS-generated

circuit, which makes optimization more difficult. Although timing measurements can be taken during high-level simulation, such measurements are based on software simulation and do not reflect actual circuit performance [6].

One potential solution to these verification, debugging, and timing-analysis problems is for designers using HLS to use postsynthesis register-transfer-level (RTL) simulation. However, such an approach requires a designer to manually add assertions to HLS-generated hardware-description-language (HDL) code, which is a cumbersome process (as compared to adding assertions at the source level) and there are numerous situations where such simulations may be infeasible or undesirable. For example, a designer may use HLS to create a custom core that is part of a larger multiprocessor system that may be too complex to model with cycle accuracy. Even if such modeling was realized, slow simulation speeds can make such verification prohibitive to many designers.

Ideally, designers could overcome these limitations by specifying assertions in high-level code, which the HLS tool could integrate into generated circuits to verify behavior and timing, while also assisting with debugging. To achieve this goal, we present HLS techniques to efficiently support in-circuit assertions. These techniques enable a designer to use assertions at the source level while checking the behavior and timing of the application. Furthermore, we leverage such assertions to enable a debugging technique referred to as hang detection that reports the specific high-level regions of code where a hang occurs. To realize these in-circuit assertion-based techniques, this paper addresses several key challenges: scalability, transparency, and portability. Scalability (large numbers of assertions) and transparency (low overhead) are interrelated challenges that are necessary to enable thorough in-circuit assertions while minimizing effects on program behavior. We address these challenges by introducing optimizations to minimize performance and area overhead, which could potentially be integrated into any HLS tool. Portability of in-circuit assertion synthesis, for verification or timing analysis, is critical because HLS tools can target numerous platforms and must, therefore, avoid platform-specific implementations. The presented techniques achieve portability by communicating all assertion failures over the HLS-provided communication channels. Using a semiautomated framework that implements the presented HLS techniques, we show that in-circuit assertions can be used to rapidly identify bugs and violations of timing constraints that do not occur during software simulation, while only introducing a small overhead (e.g., reduction in frequency on the order of less than 3% and increase in FPGA resource utilization of 0.7% or less have been observed with several application case studies on an Altera Stratix-II EP2S180 and Stratix-III EP3SE260). Various case studies with optimized assertions have shown a 3x reduction in resource usage and improved assertion performance by as much as 100% compared to unoptimized assertion synthesis. Such work has the potential to improve designer productivity and to enable the use of FPGAs by nonexperts who may otherwise lack the skills required to verify and optimize HLS-generated circuits.

This paper is presented as follows. Section 2 discusses related work. Assertion-synthesis techniques and optimizations are explained in Section 3. Section 4 discusses timing analysis. Hang detection is described in Section 5. Section 6 describes the experimental setup and framework used to evaluate the presented techniques. Section 7 presents experimental results. Section 8 provides conclusions.

2. Related Research

Many languages and libraries enable assertions in HDLs during simulation, such as VHDL assertion statements, SystemVerilog Assertions (SVA) [7], the Open Verification Library (OVL) [8], and the Property Specification Language (PSL) [9]. Previous work has also introduced in-circuit assertions via hardware assertion checkers for each assertion in a design. Tools targeted at ASIC design provide assertion checkers using SVA [10], PSL [11], and OVL [12]. Academic tools such as Camera's debugging environment [13] and commercial tools such as Temento's DiaLite also provide assertion checkers for HDL. Kakoee et al. show that in-circuit assertions [12] can also improve reliability, with a higher fault coverage than Triple Modular Redundancy (TMR) for a FIR filter and a Discrete Cosine Transform (DCT).

Logic analyzers such as Xilinx's ChipScope [14] and Altera's SignalTap [15] can also be used for in-circuit debugging. These tools can capture the values of HDL signals and extract the data using a JTAG cable. However, the results presented by these tools are not at the source level of HLS tools. A source-level debugger has been built for the Sea Cucumber synthesizing compiler [16] that enables breakpoints and monitoring of variables in FPGAs. Our work is complementary by enabling HLL assertions and can be potentially be used with any HLS tool.

Checking timing constraints of HDL applications can be performed with many of the methods mentioned above. SVA, PSL, and OVL assertions can be used to check the timing relationship between expected values of signals in an HDL application [17]. A timed C-like language, TC (timed C), has been developed for checking OVL assertions inserted as C comments for use during modeling and simulation [18]. In-circuit logic analyzers such as ChipScope [14] and SignalTap [15] can also be used to trace application signals and check timing constraints for signal values. The HLS tool, Carte, provides timing macros [6] which return the value of a 64-bit counter that is set to zero upon FPGA reset. However, most HLS tools (including Impulse C) do not provide this functionality. In-circuit implementation of high-level assertions is a more general approach that potentially supports any HLS tool and enables designers to use ANSI-C assertions.

After a comprehensive literature search, we found no previous work related to hang detection of HLS applications (except for the initial work [19] being extended by this paper). Hang detection for microprocessors has been implemented on FPGAs [20]. Nakka et al. [21] separate hang detection for microprocessors into three categories. First, Instruction-Count Heartbeat (ICH) detects a hung

process not executing any instructions. Second, Infinite-Loop Hang Detector (ILHD) detects a process which never exits a loop. Finally, Sequential-Code Hang Detector (SCHD) detects a process that never exits a loop because the target address for the completion of a loop is corrupted. Although similar detection categories could be used for hardware processes generated by HLS tools, the methods needed for hang detection are different; hardware processes typically use state machines for control flow rather than using instructions. The related work found for microprocessor hang detection is typically used to increase reliability of the system by terminating the hung process rather than to help an application developer find the problematic line of code.

Although HDL assertions could be integrated into HLS-generated HDL, such an approach has several disadvantages. Any changes to the HLL source or a different version of the HLS tool could cause changes to the generated HDL (e.g., reorganization of code or renaming of signals), which requires the developer to manually reinsert the assertions into the new HDL. It is also possible that the developer may not be able to program in HDL or the HLS tool may encrypt or obfuscate generated HDL (e.g., Labview-FPGA). HLL assertions for HLS avoid these problems by adding assertions at the source level. Specifically, ANSI-C [22] assertions were chosen to be synthesized to hardware, since they are a standard assertion widely used by software programmers. Synthesizing ANSI-C assertions would allow existing assertions already written for software programs to be checked while running in circuit.

HLS optimizations for assertions were originally introduced in [19]. In this paper, we extend that work with techniques for timing analysis and hang detection.

3. Assertion Synthesis and Optimizations

ANSI-C assertions, when combined with a testbench, can be used as a verification methodology to define and test the behavior of an application. Each individual assertion is used to check a specific run-time Boolean expression that should evaluate to true for a properly functioning application. If the expression evaluates to false, the assertion prints failure information to the standard error stream including the file name, line number, function name, and expression that failed; after this information is displayed, the program aborts.

The presented HLS optimizations for in-circuit assertions assume a system architecture consisting of at least one microprocessor and FPGA and an application modeled as a task graph. These assumptions are common to existing HLS approaches [3]; therefore, the discussed techniques are potentially widely applicable with minor changes for different languages or tools.

In-circuit assertions are integrated into the application by generating a single *assertion checker* for each assertion and an *assertion notification function*, as shown in the top right hand side of Figure 1. The assertion checker implements the corresponding Boolean assertion condition by fetching all data, computing all intermediate values, and signaling the assertion notification function upon failure. The assertion notification function is responsible for printing information regarding all assertion failures and halting the application.

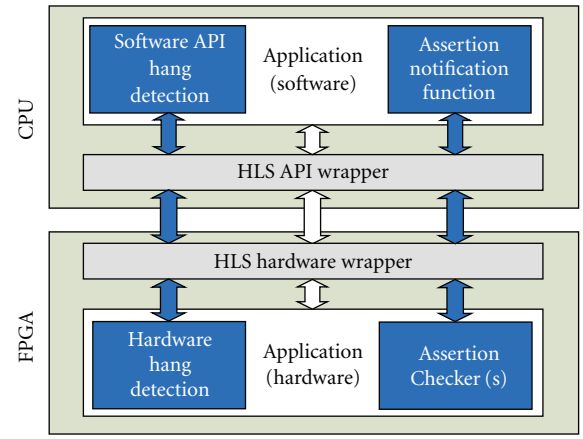


Figure 1: Assertion framework.

The assertion notification function can run simultaneously with the application as a task waiting for failure messages from the assertion checkers. The task is defined essentially as a large *switch* statement per communication channel that implements one case for each hardware-mapped assertion. Although a hardware/software partitioning algorithm could potentially map the assertion notification function task to either hardware or software, typically, the assertion notification function will be implemented in software due to the need to communicate with standard error. Although the added HLS communication channels in the task graph could greatly increase the I/O requirements for hardware/software communication, such a situation is avoided by time multiplexing all communication over a single physical I/O channel (e.g., PCIe bus, single pin). Performance overhead due to this time multiplexing should be minimal or even nonexistent (depending on the HLS tool) since ANSI-C assertions only send messages upon failure and halt the program after the first failed assertion.

One potential method to synthesize assertion checkers into circuits is described as follows. Semantically, an assert is similar to an *if* statement. Thus, assertions could be synthesized by converting each assertion into an *if* statement, where the condition for the *if* statement is the complemented assertion condition, and the body of the *if* statement transfers all failure information to the assertion notification function. Although such a straightforward conversion of *assert* statements may be appropriate for some applications, in general, this conversion will result in significant area and performance overhead. To deal with this overhead, we present three categories of optimizations that improve the scalability and transparency of in-circuit assertions, which are described in the following sections.

3.1. Assertion Parallelization. To maximize transparency of in-circuit assertions, the circuit for the assertion checker should have a minimal effect on the performance of the

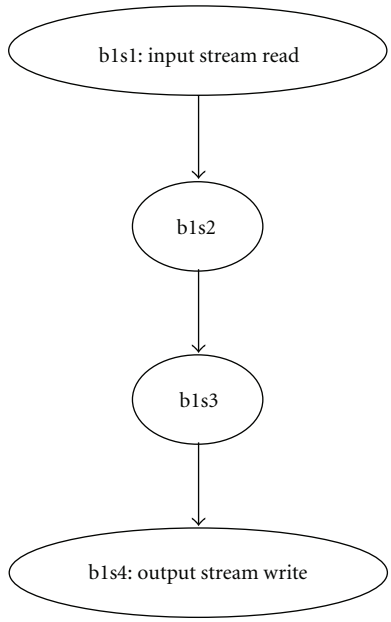


Figure 2: Application's state machine without assertion.

original application. However, by synthesizing assertions via direct conversion to *if* statements, the synthesis tool modifies the application's control-flow graph and resulting state machine, which adds an arbitrarily long delay depending on the complexity of the assertion statement. For Impulse-C, the delay of the assertion $assert((j <= 0 \mid\mid a[0] == i)\&\&(b[0] == 2 \mid\mid i > 0))$ can be shown by comparing the corresponding subset of the application's state machine before (Figure 2) and after (Figure 3) the assertion is added. For this example, the assertion can add up to seven cycles of delay to the original application for each execution of the assertion. While seven cycles may be acceptable for some applications, if this assertion occurred in a performance-critical loop, the assertion could potentially reduce the loop's rate (i.e., the reciprocal of throughput) to 12.5% of its original single-cycle performance, which could significantly affect how application components interact with each other.

HLS tools can minimize the effect of assertions on the application's control-flow graph by executing the assertions in parallel with the original application. To perform this optimization, HLS can convert each assertion statement into a separate task (e.g., a process in Impulse-C) that enables the original application task to continue execution while the assertion is evaluated. Instead of waiting for the assertion, the application simply transfers data needed by the assertion task and then proceeds.

For the previous assertion example, the optimization reduced the overhead from seven cycles to a single cycle as shown in Figure 4. The optimization was unable to completely eliminate overhead due to resource contention for shared block RAMs. Such overhead is incurred when the assertion task and the application task simultaneously require access to a shared resource.

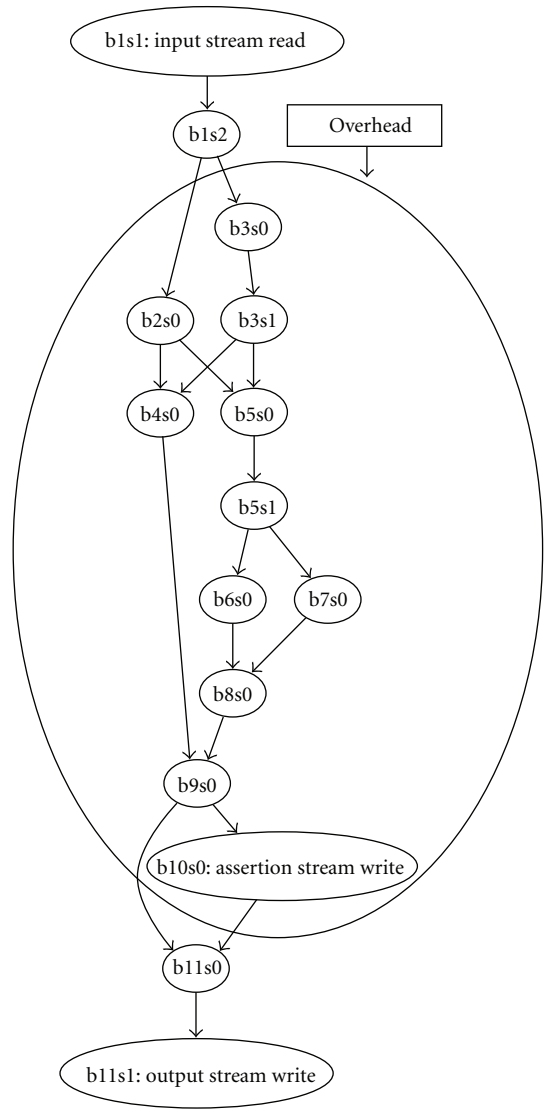


Figure 3: Application's state machine with unoptimized, serial assertion.

3.2. Resource Replication. As mentioned in the previous section, resource contention between assertions and the application can lead to performance overhead even when assertions are executed in parallel. To minimize this overhead, HLS can perform resource replication by duplicating shared resources.

For example, arrays in C can be synthesized into block RAMs. A common source of overhead is due to the limited number of ports on block RAMs that are simultaneously used by both the application tasks and assertion tasks. When accessing different locations of the block RAM, the circuit must time-multiplex the data to appropriate tasks, which causes performance overhead. HLS can effectively increase the number of ports by replicating the shared block RAMs, such that all replicated instances are updated simultaneously by a single task. This optimization ensures that all replicated instances contain the same data, while enabling an arbitrary

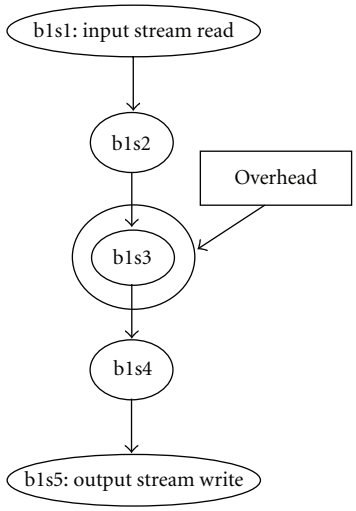


Figure 4: State machine with parallel assertion.

number of tasks to access data from the shared resource without delay.

Resource replication provides the ability to reduce performance overhead at the cost of increased area overhead. Such tradeoffs are common to HLS optimizations and are typically enabled by user-specified optimization strategies (i.e., optimize for performance as opposed to area). One potential limitation of resource replication is that for a large number of replicated resources, the increased area overhead could eventually reduce the clock speed, which may outweigh the reduced cycle delays. However, for the case study in Section 7.2.3, resource replication improved performance by 33% allowing the application's pipeline rate to remain the same.

3.3. Resource Sharing. Whereas the previous two optimizations dealt with performance overhead, in-circuit assertions can also have a large area overhead. Although an assertion checker circuit will generally cause some overhead due to the need to evaluate the assertion condition, HLS can minimize the overhead by sharing resources between assertions. For example, if a particular task has ten assertions with a multiplication in the condition, resource sharing could potentially share a single multiplier among all the assertions.

Although resource sharing is a common HLS optimization [23] for individual tasks, sharing resources across assertions adds several challenges due to the requirement that all statements sharing resources must be guaranteed to not require the resources at the same time. For task-graph-based applications, assertions may occur in different tasks at different times, which prevents a HLS tool from statically detecting mutually exclusive execution of all assertions.

Due to this limitation, HLS can potentially apply existing resource-sharing techniques to assertions within nonpipelined regions of individual tasks, because those assertions are guaranteed to not start at the same time. However, due to the assertion parallelization optimization, different starting times for two assertions do not guarantee that their execution does not overlap. For example, an assertion with a complex condition may not complete execution before a later assertion requires a shared resource. To deal with this situation, HLS can implement all assertions that share resources as a pipeline that can start a new assertion every cycle. Although this pipeline will add latency to all assertions in the same task that require access to the shared resources, such latency does not affect the application and only delays the notification of program failure. This technique of pipeline assertion checking is evaluated in Section 7.2.1.

Resource sharing could potentially be extended to support an arbitrary number of simultaneous assertions in multiple tasks by synthesizing a pipelined assertion checker circuit that implements a group of simultaneous assertions. To prevent simultaneous access to shared resources, the circuit could buffer data from different assertions using FIFOs (e.g., one buffer per assertion) and then process the data from the FIFOs in a round-robin manner. This extension requires additional consideration of appropriate buffer sizes to avoid having to stall the application tasks and an appropriate partitioning of assertions into assertion checker circuits, which we leave as future work.

In some cases, resource sharing may improve performance in addition to reducing area overhead by enabling placement and routing to achieve a faster clock due to fewer resources. However, resource sharing will at some point experience diminishing returns and may eventually increase clock frequency due to a large increase in multiplexers and other steering logic.

4. In-Circuit Timing-Analysis Assertions

For applications with real-time requirements, particularly in embedded systems, verification must guarantee that all timing constraints are met (a process referred to as *timing analysis*) in addition to checking the correctness of application behavior. If an HLS-generated application does not meet timing constraints during execution, then it would be helpful to know the location of the section of code that is violating constraints in order to focus optimization effort. However, determining the performance of an HLS-generated application can be difficult. HLS tools, such as Impulse-C and Carte, provide some compile-time feedback about the rate and latency of a pipelined loop, but it is largely unknown how many cycles a particular line of code will require. While it is possible to determine the number of cycles a line (or lines) of code will take by examining the HDL generated by the tool, delay can be data dependent, as shown in the possible traversals of the state machine generated by the evaluation of the conditional statement $if((j <= 0 \;||\; a[0] == i)\&\&(b[0] == 2 \;||\; i > 0))$ in Figure 3). However, such a process requires significant designer effort and requires the designer to have knowledge of the HLS-generated code. While a delay range for the computation in each line of code could be provided by the HLS tool via static analysis, the delay of communication calls cannot be determined by static analysis. Software

simulation cannot provide accurate timing due to timing differences between thread execution on the microprocessor and execution on the FPGA. In this section, we describe the additional concepts and methods needed to extend in-circuit assertions to perform timing analysis for applications built with HLS tools.

Figure 5 illustrates usage of timing-analysis assertions for an audio filtering application designed with a HLS tool. In this example, the application designer has determined that the filter takes too long to execute on the FPGA by measuring the time to run the application on the FPGA. However, the application designer is unsure of which part of the application in the FPGA is not meeting timing constraints. Using timing-analysis assertions, the application designer can check the timing of different application regions in the FPGA, as shown in the figure in addition to the case study in Section 7.5. Data-dependent delays can be checked to see if they are within bounds for each loop iteration. Although not shown in the figure, the same method can be used to check streaming communication calls for delays caused by buffers becoming full or empty.

In order to enable ANSI-C assertions to check the timing of an application, time must be accessible via a variable. In C, time is typically determined via a function call. In Figure 5, the ANSI-C function, *clock*, is used to return the current time in cycles. To measure the time of a section of code, the *clock* function should be called before and after that section of code, with the difference between the two times providing the execution time (in cycles). To perform timing analysis, an assertion can be used to check a comparison between the expected time and the measured time. For example, in Figure 5, the code in the loop for each filter is expected to take less than 100 cycles.

For timing-analysis assertions, time can potentially be represented in many different formats. However, returning time in terms of cycles will require the least amount of overhead. The ANSI-C library provides the *clock* timing function that returns the number of clock ticks that have elapsed since the program started. However, for C programmers who may want to express time in terms of seconds rather than cycles, the ANSI-C constant expression *CLOCKS_PER_SEC* can be used to convert clock ticks to time in seconds. The clock frequency of the FPGA could be determined by comparison with timestamps sent from the CPU. However, an assertion may need to be checked on the first cycle after an FPGA restart. Since determining the frequency of the FPGA automatically could take too long, a preprocessor constant *FPGA_FREQ* is used to define the FPGA frequency in Hz.

The type defined for representing clock ticks in ANSI-C is *clock_t* that typically corresponds to a long integer. For added flexibility when used in hardware, time can be returned and stored as a 32-bit or 64-bit value. A 64-bit value is used by default. To select a 32-bit value, the preprocessor constant *CLOCK_T_32* must be defined in the code. A 32-bit value can be used to reduce overhead but will overflow after 43 seconds for a clock speed of 100 MHz. During software simulation, the assertions using timing information are ignored, which allows simulation to check correctness of the application while ignoring the timing of the microprocessor.

To enable synthesis of timing assertions, a counter, which is set to zero upon reset, is added in each hardware process that contains a *clock* statement. The value returned by the *clock* statement is generated by latching the counter signal for each transition of the state machine. Use of a latched counter signal ensures that the timer value is consistently taken at the beginning of each state transition for states that execute more than one cycle.

One potential problem with this approach is that HLS tools often reorder statements to maximize parallelism. Therefore, *clock* statements could potentially be reordered leading to incorrect timing results. However, such a problem is easily addressed by making a synthesis tool aware of *clock* statements. In this paper, we alternatively evaluated the techniques using instrumentation due to the inability to modify commercial HLS tools. Although instrumentation could experience reordering problems, for the evaluated examples, reordering of *clock* statements did not occur.

5. Hang-Detection Assertions

A common problem with FPGA applications is a failure to finish execution, which is often referred to as *hanging*. Common causes of hanging include infinite loops, synchronization deadlock, blocking communication calls that wait indefinitely to send or receive data, and so forth. Determining the cause of a hanging application, referred to as *hang detection*, is difficult for HLS-generated FPGA designs. While a debugger could be used to trace down the problem during software simulation, the inaccuracies of software simulation can miss hangs that occur during FPGA execution. To deal with this problem, we extend in-circuit assertions to enable hang detection for HLS-generated application.

One challenge of hang detection using assertions is that it is assumed that the assertion will eventually be checked. If the application waits indefinitely for a line of code to finish (e.g., an infinitely blocking communication call) then a different detection method is needed, since the assertion after the hung line will never be executed as shown in Figure 6(a). Without some mechanism to alert the developer to the current state of the program, it will be difficult to pinpoint the problem. For example, in the filter application (see Figure 7), the source of the problem that is causing the application to hang could be in any of the software or hardware processes.

One potential solution is to use assertions in a counter-intuitive way by adding assertions periodically throughout the code that are designed to fail (i.e., *assert*(0)). By also defining the *NABORT* flag, failed assertions will not cause the application to abort, which allows the developer to manually create an application heartbeat (i.e., a signal sent as a notification that the process is alive) that traces the execution of the application on the FPGA as shown in Figure 6(b). In the filter application example, multiple assertions would need to be placed in strategic locations in each FPGA process to determine the events that take place before the application hangs. The resolution (in terms of lines of code) would be determined by how many assertions are used. Unfortunately,

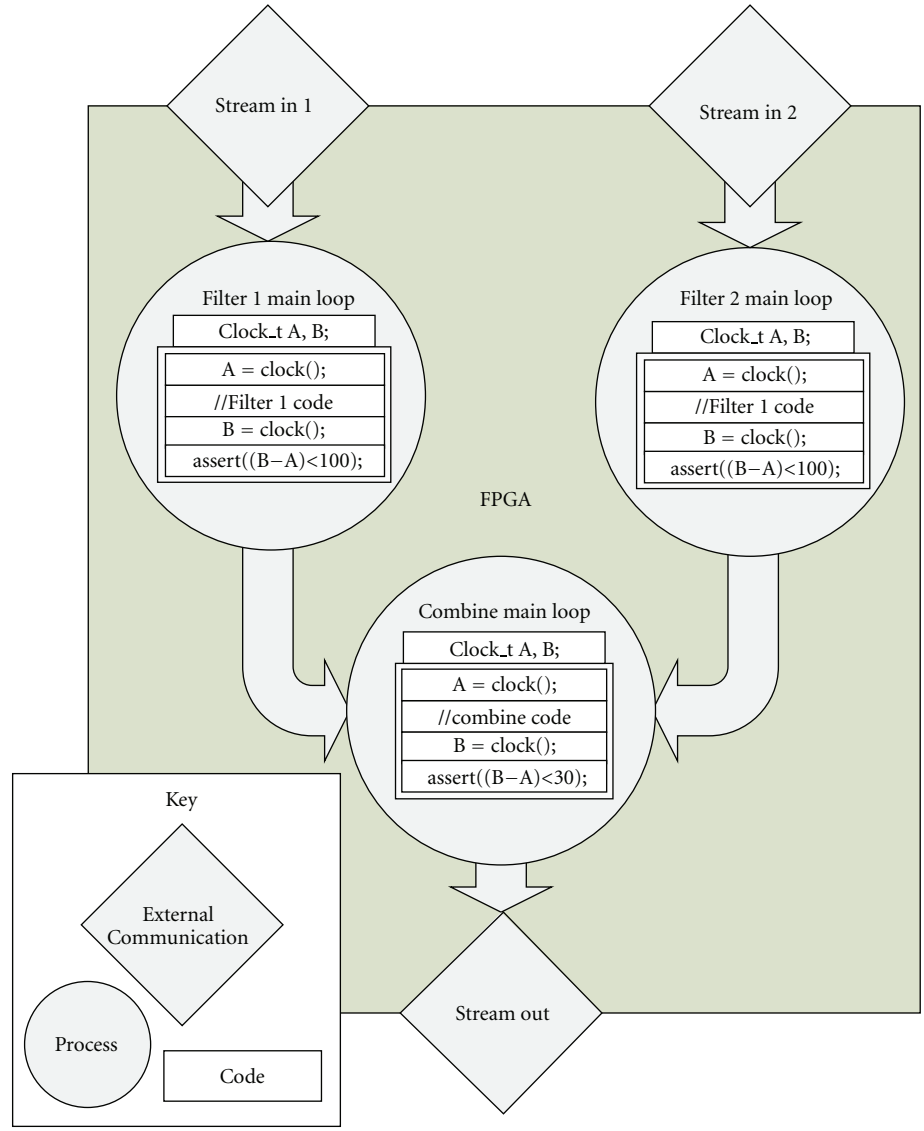
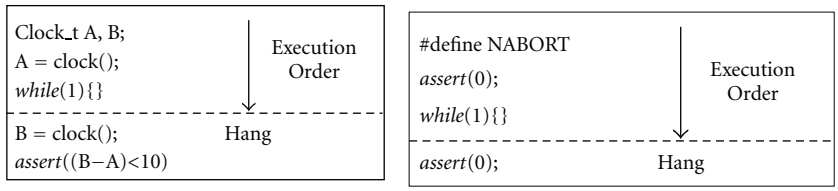


Figure 5: Using timing-analysis assertions with a filter application.



(a) Timing Assertion (b) Assertions set to fail

Figure 6: Manually using ANSI-C assertions for hang detection.

if a large number of assertions are used, then large amounts of communication and FPGA resources could be used by the assertions. Although this approach works, it requires significant designer effort and has large overhead.

To reduce effort and overhead, we present a more automated method of hang detection that does not require user instrumentation and instead uses watchdog timers to monitor the time between changes of the signals that represent the state of the hardware process. The monitoring circuit has software-accessible registers that contain the current state of all hardware process and the state of any hardware process that it has detected as hung. Hang detection is triggered using a watchdog timer for a hardware process that signals when a state takes longer than a user-defined number of cycles;

the assertion pragma, *#pragma assert_FPGA_watch_dog*, sets this timeout period, which is reset anytime a state transition occurs. The watchdog timer is sized to be just large enough to hold the cycle count given in the pragma to reduce FPGA resource and frequency overhead. In software, a separate thread is spawned to monitor the hardware hang detector to check for hung states (i.e., expired watchdog timers). If a hardware process has hung, then the state in the registers is matched to the corresponding line of code via a lookup table generated by parsing an intermediate translation file (both Impulse C and Carte create these files). The state of all other hardware processes are given for reference.

In software, many HLS applications will wait indefinitely at some point in its execution for the FPGA to respond with some form of communication or synchronization. For those applications, hangs caused in the FPGA hardware will also cause the software to hang on the communication or synchronization API call for the FPGA. Although traditional debugging tools can be used to detect these hangs in software, software hang detection is provided to monitor the HLS API calls for convenience. A thread is spawned for all API calls of the HLS tool. The thread will check if the API call finishes within a time period set by the assertion pragma, *#pragma assert_API_watch_dog*. If the API call takes longer than the timeout period, then the current line of code for the API call and all hardware processes will be printed to standard output and the program will abort.

This automated approach simplifies the addition of hang detection to an application, as shown for the filter application in Figure 7 and case study in Section 7.6, compared to manually adding *assert*(0) statements. Two assertion pragmas are added to the application before instrumentation to set the watchdog timeout periods in hardware and software. Although hangs can be caused by the interaction between two or more (hardware or software) processes, providing the state of the hung process along with the current state of all other hardware processes can greatly narrow down the source of problem.

Several improvements can be added to further enhance hang detection of HLS applications. The feedback given to the application developer can be increased by reporting more than the last state of each process in the FPGA. For example, a trace buffer could be added of a user-defined size that would capture the sequence of state that occurred before the hardware process hung. Also, infinite loops in a hardware process will only trigger software API hang detection. Since infinite loops will not stay in a single state to trigger the hang-detection method mentioned above, detection of infinite loops in hardware could also be incorporated by adding a second counter for each process that is dedicated to counting the number of cycles spent in states that are known to be inside one or more loops. The overhead of hang detection could be reduced by allowing the user to select which processes to monitor. The hang detection counters could be removed for some or all processes, while still allowing the current state of the process to be periodically retrieved or retrieved by software API hang detection. This approach would give the user the option to customize hang detection to fit for designs that nearly fill the FPGA.

6. Assertion Framework

To evaluate the assertion-synthesis techniques, we created a prototype tool framework for Impulse-C that implements the techniques via instrumentation of HLL and HDL code. It should be noted that we use instrumentation because we are unable to modify the proprietary Impulse-C tool. All of the techniques are fully automatable and ideally would be directly integrated into an HLS tool.

6.1. Unoptimized Assertion Framework. To implement basic in-circuit assertion functionality, the framework uses HLL instrumentation to convert *assert* statements into HLS-compliant code in three main stages. First, the C code for the FPGA is parsed to find functions containing assertion statements, converting any assertion statements to an equivalent *if* statement. A false evaluation produces a message that will be retrieved from the FPGA by the CPU, uniquely identifying the assertion. Next, communication channels are generated to transfer these messages from the FPGA to the CPU. Finally, the assertion notification function is defined as a software function executing on the CPU to receive, decode, and display failed assertions using the ANSI-C output format. An example of this automated code instrumentation is shown in Figure 8.

To notify the user of an assertion failure, the framework uses an error code that uniquely identifies the failed assertion based on the line number and file name of the assertion. Once the assertion notification function decodes the assertion identifier, the user is notified by printing to the standard error stream by the CPU for the current framework. The framework could be extended to work without a CPU by having the assertion identifier stored to memory, displayed on an LCD, or even flashed as a sequence on an LED by the FPGA. Alternatively, an FPGA could potentially use a softcore processor.

Note that other changes are needed to route the stream to the CPU, such as API calls to create and maintain the stream. The stream must also be added as a parameter to the function. The output of the framework is valid Impulse-C code, allowing further modifications to the source code with no other changes to the Impulse-C tool flow. Once verification of the application is finished, the constant NDEBUG can be used to disable all assertions and reduce the FPGA resource overhead for the final application. An additional nonstandard constant NABORT can be used to allow the application to continue instead of aborting due to an assertion failure.

6.2. Assertion Framework Optimizations. In order to evaluate the optimizations presented in Section 3, a hybrid mix of manual HLL and HDL instrumentation was used. To enable assertion parallelization (Section 3.1), the framework modifies the HLL code to move assertions into a separate Impulse-C process. The framework introduces temporary variables to extract data needed by the assertion. HDL instrumentation then connects the temporary variables and trigger conditions between processes. The results of this optimization can be found in Section 7.2.

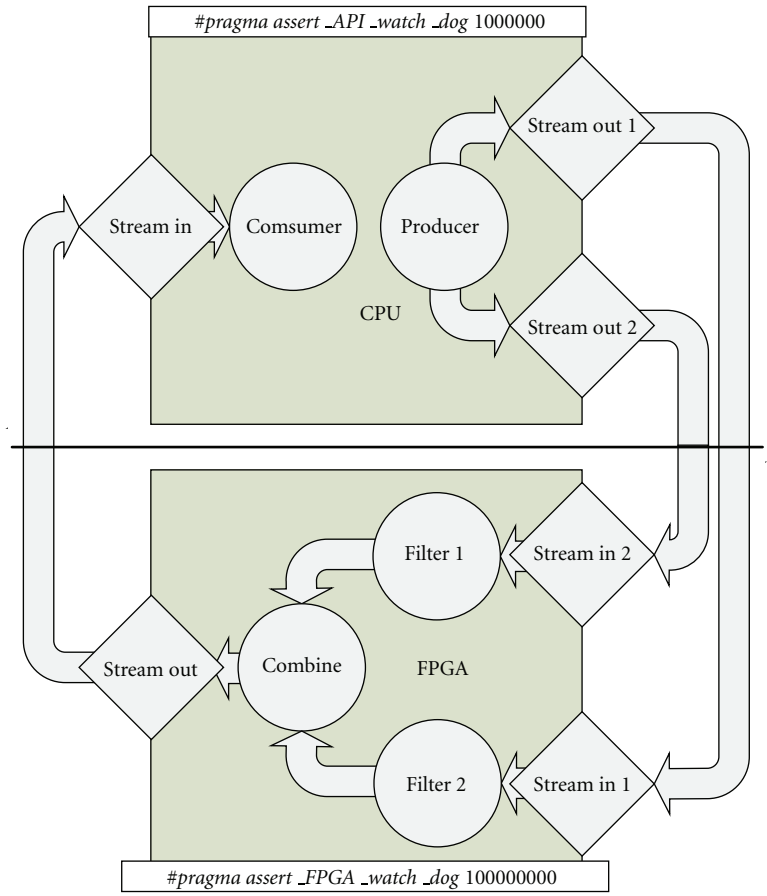


Figure 7: Using hang-detection assertions with a filter application.

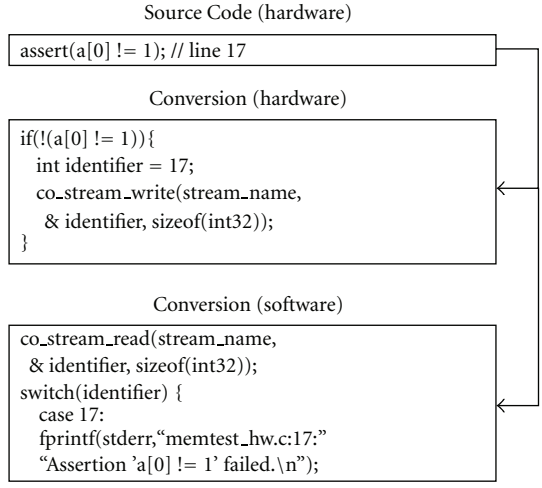


Figure 8: HLL assertion instrumentation.

Resource replication, described in Section 3.2, was performed using manual HLL instrumentation. An extra array was added to the source code that performed the same writes as the original array but reads were only performed by the assertion, as shown in Section 7.3.

The following manual hybrid instrumentation was used to evaluate resource sharing as described in Section 3.3. Although resource sharing could potentially be applied to any shared resource, we evaluate the optimization for shared communication channels, which are common to all

Impulse-C applications. HLL instrumentation creates a streaming communication channel per Impulse-C process and sends the identifier of the assertion upon assertion failure. Creating a streaming communication channel per Impulse-C process can become expensive in terms of resources if a large number of Impulse-C processes contain assertions. To reduce the number of streams created for each process, a single bit of the stream is used per assertion to indicate if an assertion has failed. This technique allows Impulse-C processes to more efficiently utilize the streaming communication channels. When streaming communication resources are shared, a separate process is created via HLL instrumentation that can handle failure signals from up to 32 assertions per process if a 32-bit communication channel is used. For example, if all 32 assertions fail simultaneously, then all 32 bits of the communication channel will simultaneously be asserted. The failure signals are connected to assertions using HDL instrumentation for efficiency. The overhead reduction associated with using this technique is explored in the case study that is presented in Section 7.4.

6.3. Timing-Analysis and Hang-Detection Extensions. Semiautomatic hybrid instrumentation was used to support timing functions presented in Section 4. Impulse-C does not support ANSI-C library calls so the *clock* function calls must be removed. A placeholder variable is declared and used in place of the *clock* statement in the source code. After hardware generation, a Perl script is used to instrument the HDL. A counter is added in each hardware process that contains a *clock* statement, which is set to zero upon reset. A second signal is added to the process that latches the counter signal upon transition of the state machine. The placeholder variable, synthesized into a signal with a similar name in HDL, is replaced with the latched counter signal.

Semiautomatic hybrid instrumentation was used for hang detection in Section 5. For software hang detection, a wrapper was added around each of the Impulse-C library API calls which added the threaded hang detection. The modified software API calls required extra parameters for access to the hardware hang-detection registers. Automatic parsing of the *xhw* file generated by Impulse-C allows states to be converted to line numbers. For hardware hang detection, a hardware process supporting register transfer to software is automatically added to the source code. After Impulse-C generates the HDL, the state machine signals of all other hardware processes are automatically routed into the hang-detection process. The hang-detection circuit is then manually added by overwriting part of the register transfer process.

Although many of the steps for adding timing-analysis and hang-detection instrumentation were manual, all of the steps could be automated via Perl scripts. Ideally, modification to the Impulse-C tool would be made instead of instrumenting source and intermediate code. However, because Impulse-C is proprietary, such modification was not possible for this work.

6.4. HLS Tool and Platform. The framework currently uses Impulse-C. Impulse-C is a high-level synthesis tool to convert a program written in a subset of ANSI-C to hardware in an FPGA. Impulse-C is primarily designed for streaming applications based upon the communicating sequential process model but also supports shared memory communication [5]. Speedups can be achieved in Impulse-C applications by running multiple sequential process in parallel, pipelining loops, and adding custom HDL-coded functions calls.

Quartus 9 was used for synthesis, and implementation of the Impulse-C-generated circuits. The target platforms are the XtremeData XD1000 [24] containing a dual-processor motherboard with an Altera Stratix-II EP2S180 FPGA in one of the Opteron sockets and the Novo-G supercomputer [25] at University of Florida containing 48 GiDEL PROCStar III [26] cards each with four Stratix-III EP3SE260. Impulse C 3.3 is used for the XD1000 while Impulse-C 3.6 with an in-house platform support package is used for Novo-G. Although the XD1000 and Novo-G are high-performance computing platforms, Impulse-C also supports embedded PowerPC and MicroBlaze processors [5]. Furthermore, Novo-G and the XD1000 are representative of FPGA-based embedded systems that combine CPUs with one or more FPGAs. The presented overhead results would likely be similar for other embedded platforms, assuming similar Impulse-C wrapper implementations.

Although we currently evaluate HLS assertions using Impulse-C, the techniques are easily extended to support other languages. For example, in Carte, Impulse-C's streaming transfers would be replaced with DMA transfers. The software-based assertion notification function (see Figure 1) would then need to monitor Carte's FPGA function calls for failed assertions as opposed to monitoring Impulse-C's FPGA processes.

7. Experimental Results

This section presents experimental results that evaluate the utility and overhead of the presented assertion synthesis, timing analysis and hang detection. Section 7.1 motivates the need for in-circuit assertions by illustrating a case study where assertions pass during simulation but fail during FPGA execution. Section 7.2 illustrates the performance and overhead improvements of the assertion parallelization optimization. Section 7.3 evaluates performance benefits of resource replication. Section 7.4 evaluates the scalability of assertions in terms of resource and frequency overhead by applying resource sharing optimizations to the communication channels. Section 7.5 presents the overhead of using assertions for timing analysis. Section 7.6 evaluates two hang-detection methods used on an application that fails to complete.

The designs used in the case studies occupy a relatively small part of the FPGA (24% of logic used in Section 7.5). Designs with higher resource utilization may lead to greater performance degradation and resource overhead of assertions due to increased difficulty in placement and, routing for example. In addition, resource replication might not be applicable for designs that are almost full.

```
1  co_unit64 c2, c1;
2  co_int32 address, array[20], out;
3  c2 = 4294967286; c1 = 4294967296;
4  if (c2 > c1) address = c2 – c1;
5  else address = 0;
6  assert(address >= 0);
7  out = user(address);
8  assert((30 > out) && (out > 20));
9  array[address] = out;
```

Algorithm 1: In-circuit verification example.

7.1. Detecting Simulation Inconsistencies. In this section, we illustrate how assertions can be used for in-circuit verification and debugging to catch inconsistencies between software simulation and FPGA execution of an application. The code in Algorithm 1 shows how assertion statements can be used for in-circuit verification by identifying bugs not found using software simulation. The first assertion is used to detect a translation mistake from source code to hardware (it is possible for a translation mistake to also have an effect on an *assertion*) . The assertion statement (line 6) never fails in simulation but fails when executed on the XD1000 platform. Upon inspection of the generated HDL, it is observed that Impulse-C performs an erroneous 5-bit comparison of c2 and c1 (line 4). The 64-bit comparison of 4294967286 > 4294967296 (which evaluates to false) becomes a 5-bit comparison of 22 > 0 (which evaluates to true), allowing the array address to become negative (line 4). In contrast, the simulator executing the source code on the CPU sets the address to zero (line 5). Impulse C will generate a correct comparison when c1 and c2 are 32-bit variables.

The second assertion (line 8) is used to check the output of an external HDL function (line 7), which is used to gain extra performance over HLS generated HDL. When an external HDL function is used, the developer must provide a C source equivalent for software simulation. However, the behavior and timing of the C source for simulation may differ from the behavior of the external HDL function during hardware execution, again demonstrating a need for in-circuit verification.

For demonstration purposes, this example case is intentionally simplistic and similar conclusions could be drawn using a cycle-accurate HDL simulator. However, in practice, inconsistencies caused by the timing of interaction between the CPU and FPGA would be very difficult to model in a cycle-accurate simulator.

7.2. Assertion Parallelization Optimization. This section provides results for the parallelization optimization of assertions. Section 7.2.1 shows improvements from optimization for Triple-DES encryption. Section 7.2.2 shows optimization improvements for edge-detection. While the applications in the previous sections evaluate frequency overhead, Section 7.2.3 evaluates state machine performance overhead (in terms of additional cycles) and optimization improvements.

Table 1: Triple-DES assertion overhead.

EP2S180	Original	Assert	Difference
Logic used	13677	13851	+174
(out of 143520)	(9.53%)	(9.65%)	(+0.12%)
Comb. ALUT	7929	8025	+96
(out of 143520)	(5.52%)	(5.59%)	(+0.07%)
Registers	10019	10055	+36
(out of 143520)	(6.98%)	(7.01%)	(+0.03%)
Block RAM	222912	223488	+576
(9383040 bits)	(2.37%)	(2.38%)	(+0.01%)
Block interconnect	24657	24878	+221
(out of 536440)	(4.60%)	(4.64%)	(+0.04%)
Frequency (MHz)	145.7	142.0	−3.7 (−2.54%)

7.2.1. DES Case Study. The first application case study shows the area and clock frequency overhead associated with adding performance optimized assertion statements to a Triple-DES [27] application provided by Impulse-C, which sends encrypted text files to the FPGA to be decoded. Two assertion statements were added in a performance critical region of the application to verify that the decrypted characters are within the normal bounds of an ASCII text file. Table 1 shows all sources of overhead, including the streaming communication channels generated by Impulse-C for sending failed assertions back to the CPU. The overhead numbers were found to be quite modest, with resource usage increasing by at most 0.12% of the device and the maximum clock frequency dropping by less than 4 MHz.

For this case study, the optimized assertions were checked in a separate pipeline process to reduce the overhead generated by the assertion comparison. Assertion failures are sent by another process to ensure that assertions can be checked each cycle. The state machine of the application remained unchanged because the optimized assertions were checked in a separate task working in parallel with the application. Since the application's state machine remained the same, the only performance overhead comes from the maximum clock frequency reduction. The resource overhead for optimized assertions actually decreased as compared to unoptimized assertions. The ALUT (Adaptive Look-Up Table) and routing resources needed by Quartus to achieve a maximum frequency of 144.7 MHz for unoptimized assertions was 0.06% greater than the ALUT and routing resources need for optimized assertions that achieved a maximum frequency of 142 MHz.

7.2.2. Edge-Detection Case Study. The following case study integrates performance optimized assertions into an edge-detection application. The edge-detection application, provided by Impulse-C, reads a 16-bit grayscale bitmap file on the microprocessor, processes it with pipelined 5×5 image kernels on the FPGA, and streams the image containing edge-detection information back. Since the FPGA is programmed to process an image of a specific size, two assertions were added to check that the image size (height and width) received by the FPGA matches the hardware configuration.

Table 2: Edge-detection assertion overhead.

EP2S180	Original	Assert	Difference
Logic used	12250	12273	+23
(out of 143520)	(8.54%)	(8.56%)	(+0.02%)
Comb. ALUT	6726	6809	+83
(out of 143520)	(4.69%)	(4.75%)	(+0.06%)
Registers	9371	9417	+46
(out of 143520)	(6.53%)	(6.56%)	(+0.03%)
Block RAM	141120	141696	+576
(9383040 bits)	(1.50%)	(1.51%)	(+0.01%)
Block interconnect	19904	19994	+90
(out of 536440)	(3.71%)	(3.73%)	(+0.02%)
Frequency (MHz)	77.5	79.3	+1.8 (+2.32%)

Table 3: Single-comparison assertion.

	Latency Overhead	
Assertion data structure	Unoptimized	Optimized
Scalar variable	1	0
Array (non-consecutive)	1	0
Array (consecutive)	2	1

Table 4: Pipelined single-comparison assertion.

	Overhead			
	Unoptimized		Optimized	
Assertion data structure	Latency	Rate	Latency	Rate
Scalar variable	1	1	0	0
Array	2	1	1	0

The assertions were added in a region of the application that was not performance critical. As shown in Table 2, the overhead numbers for this case study were also modest, with resource usage increasing by at most 0.06% on the EP2S180.

For the edge-detection case study, the optimized assertions were checked in a separate process to reduce the overhead generated by the assertion comparison. Since the applications state machine remained the same, and maximum clock frequency did not reduce, the application did not incur any performance overhead due to the addition of the assertions. The frequency increase is likely due to randomness in placement and routing results of similar designs. The performance optimization of the assertions increased ALUT resource utilization from 0.03% to 0.06% on the EP2S180.

7.2.3. State Machine Overhead Analysis. This section presents a generalized analysis of performance overhead caused by adding assertions with a single comparison and the performance improvement via optimizations. The results in this section present overhead in terms of cycles and exclude changes to clock frequency, which was discussed in the previous section. We evaluate single-comparison assertions to determine a lower bound on the optimization improvements. To measure the performance overhead of adding assertions, we examine the state machines and pipelines generated by Impulse-C. Impulse-C allows loops (e.g., *for* loops or *while* loops) to be pipelined. Assertions added to a pipeline can modify the pipeline's characteristics. Each pipeline generated by Impulse-C has a latency (time in cycles for one iteration of a loop to complete) and rate (time in cycles needed to finish the next loop iteration). Assertions that are not in a pipelined loop will add latency (i.e., one or more additional states) to the state machine that preserves the control flow of the application. As stated in Section 6.2, assertions can be optimized to reduce or eliminate the overhead of assertions in terms of additional clock cycles required to finish application execution. These optimizations move the comparisons to a separate Impulse-C process so that they can be checked in parallel with the application. Any remaining clock cycle overhead after optimization comes from the data movement needed for assertion checking.

Table 3 shows the latency overhead for nonpipelined, single comparison assertions. In most cases, assertions with these comparisons will increase latency by one cycle. With optimizations, this latency overhead is reduced to zero since extracting data in most cases will not add latency to the application. In the case where an array is consecutively accessed temporally by the application and an assertion, an unoptimized assertion will have a latency overhead of two cycles because of block RAM port limitations. With optimizations, this latency overhead is reduced to one cycle to extract data from the array or block RAM. For more complex assertions, the latency will increase for unoptimized assertions while the latency for optimized assertions will remain the same, as seen when comparing Figures 3 and 4. Even with the multiple array accesses in $assert((j <= 0 \,||\, a[0] == i)\&\&(b[0] == 2 \,||\, i > 0))$, only one cycle is needed to retrieve the array data.

Table 4 shows pipeline latency and rate overhead observed for a single comparison. Adding an unoptimized assertion using a scalar variable to a pipelined loop increased the latency from 2 to 3, resulting in an overhead of one cycle, and degraded the rate from 1 to 2 for the pipeline. Although the rate overhead was a single cycle, this corresponds to a 2x slowdown in performance because the throughput is reduced to half of the original loop. This overhead comes from adding a streaming communication call. For the optimized assertion, the streaming communication call was moved to a separate process that reduced the latency and rate overhead to zero, resulting in a 2x speedup compared to the unoptimized assertions. For assertions using arrays in pipelined loops, adding an assertion caused a 2-cycle latency overhead that increased the latency from 2 to 4. The assertion reduced the rate from 2 to 3, which is a one cycle rate overhead that corresponds to a 50% reduction in performance.

7.3. Resource Replication Optimization. As mentioned in Section 7.2.3, Table 4 shows pipeline latency and rate overhead observed for a single comparison. For assertions used

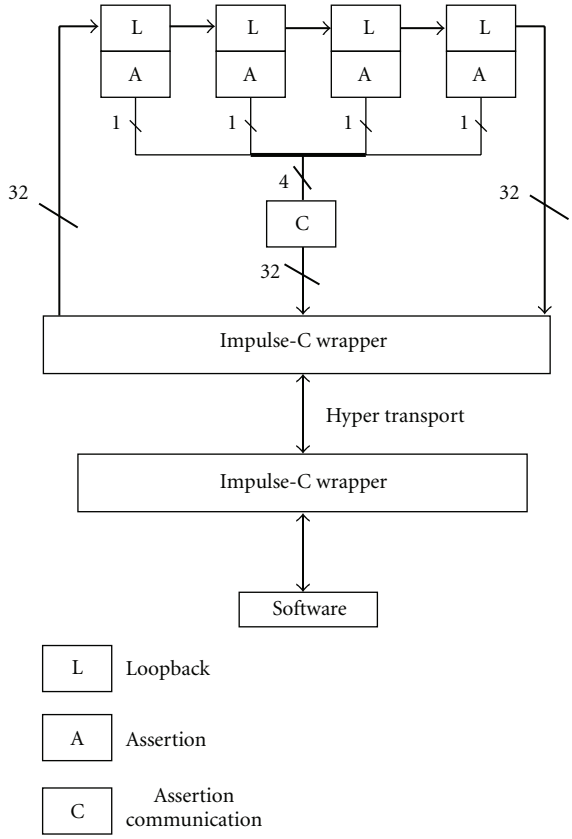


Figure 9: Simple streaming loopback.

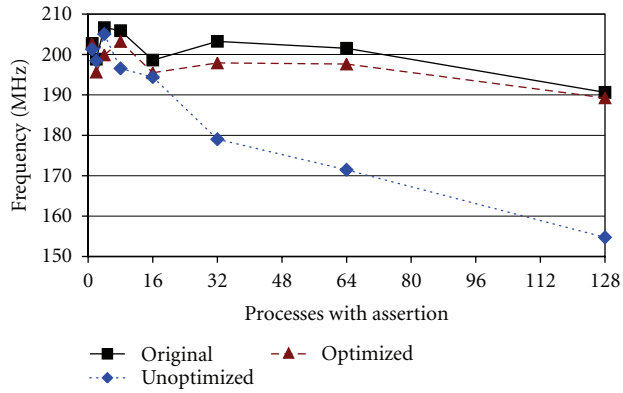


Figure 10: Assertion frequency scalability.

in pipelined loops checking an array data structure, the assertion overhead was reduced via resource replication by adding an additional array to the process dedicated read access to the assertion as described in Section 6.2. With a duplicate array, only the latency increased from 2 to 3 and the rate remained the same which corresponds to a 33% rate improvement over the nonoptimized version. A similar improvement could be gained for a nonpipelined assertion that checks multiple indexes to the same array.

7.4. Resource Sharing Optimization. This section demonstrates the improvement in scalability from resource sharing optimization techniques. We evaluate scalability by measuring the resource and clock frequency overhead incurred by adding assertions to a large number of Impulse-C processes, providing an extremely pessimistic scenario in terms of overhead. A single assertion is added per process which results in a separate streaming communication channel for each process. A single greater than comparison is made per process, generally requiring only minor changes to the process state machine. In this study, the application consists of a simple streaming loopback as shown in Figure 9. The loopback also stores the value and retrieves the value at each stage. Each process added to the application adds an extra stage in the loopback (e.g., for 4 FPGA processes shown as L in Figure 9, incoming data would be passed from the input to the FPGA, passing through each of the processes before being returned to the CPU). The assertion in each process ensures the number being passed is greater than zero. Each process adds overhead in terms of an assertion shown as A in Figure 9 and an extra Impulse-C streaming communication channel shown as C in Figure 9 to notify the CPU of failed assertions. For a 32-bit stream, up to 32 assertions can be connected to the streaming communication channel before a new streaming communication channel is needed.

Using the previously discussed straightforward conversion of *assert* statements to *if* statements, the unoptimized assertions with 128 processes (128 assertions) had a resource overhead on the EP2S180 of 4.07% ALUTs (the highest resource percentage overhead). However, the maximum frequency decreased from 190 MHz for the 128-process original application to 154 MHz or an 18.8% overhead as shown in Figure 10 for the 128-process application with unoptimized assertions.

By applying the resource sharing optimization only to the communication channels so that only a single bit of the stream is used per assertion as described in Section 6.2 (and not the assertion resources), the resource overhead was decreased. The resource overhead on the EP2S180, as shown in Figure 11, was reduced to 1.34% of ALUTs or over a 3x improvement for the 128-process application with assertions. Assertion optimizations increased the maximum frequency for the 128-process application to 189 MHz, as shown in Figure 10, which represents over an 18% improvement. The frequency of the application with assertion optimizations (189.3 MHz) was very close to the original application's frequency of 190.6 MHz. While the resource usage increased consistently for all three tests (original, unoptimized, and optimized) from 1 to 128 processes, the maximum frequencies reported by Quartus did not consistently decrease as the number processes increased until 32 processes were added. The frequency overhead decreased from 32 to 128 processes with optimized assertions because the application added one stream per process, while the assertions only added one stream per 32 processes since 32-bit streaming communication was used. This demonstrates the benefits of the resource sharing optimization for streaming communication channels.

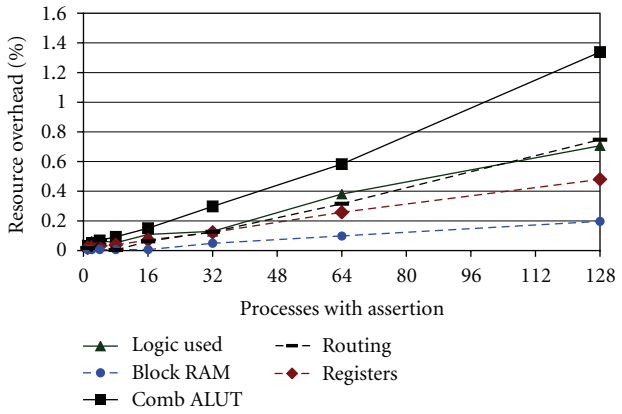


Figure 11: Optimized assertion resource scalability.

7.5. In-Circuit Timing Analysis. This section provides a case study showing the utility and overhead of adding assertions with timing statements to a backprojection application. Backprojection is a DSP algorithm for tomographic reconstruction of data via image transformation. For the backprojection application, instrumentation was added into a nested loop (see Algorithm 2). Two 32-bit timing calls were added around the inner pipelined loop to measure the time required for the pipelined loop to finish generating 512 pixels. After the timing calls, ten assertions were added to find the maximum time required for the pipelined loop to finish for all outer-loop iterations. Since the inner loop has 512 iterations, a minimum of 512 cycles should be needed to complete the loop; however, more cycles could be required for stalls and flushing of the pipeline. To test these assumptions, ten assertions were added to check the timing of the loop with exponentially increasing maximum times and *NABORT* was defined to stop the application from aborting. After execution, only the first assertion passed evaluation, which means that the maximum time for the inner loop is between 640 and 1023 cycles.

This technique allows the application designer to quickly check timing in multiple regions of the application with minimal disturbance to the application in terms of resource and communication overhead. After evaluating the feedback from the assertions, the application designer can modify the application to stream back the exact timing values for problematic regions of code. In addition, the assertion feedback provided before modifying the application can be used to make sure that the timing values streamed back are valid. It is possible that the addition of large data transfers could change the timing of the application.

The backprojection application runs on all four Stratix-III EP3SE260 FPGAs on the GiDEL PROCStar III [26] card. Overhead is only given for one FPGA since the image is split between all four FPGAs. Ideally, a single assertion could check an array of values in a loop for more compact code (see Algorithm 3). However, that approach increases overhead when synthesized with Impulse-C as shown in Table 6 as compared to using individual assertions as shown in Table 5.

```
for(y=0;y<512;y++)
{
   time1 =c1ock();
   for(x=0;x<512;x++)
   {//compute pixel
     . . .
   }
   time2=clock();
   assert((time2-time1)<1024));
   assert((time2-time1)<640));
   assert((time2-time1)<576));
   assert((time2-time1)<544));
   assert((time2-time1)<528));
   assert((time2-time1)<520));
   assert((time2-time1)<516));
   assert((time2-time1)<514));
   assert((time2-time1)<513));
   assert((time2-time1)<512));
   . . .
}
```

Algorithm 2: Adding timing assertions individually to backprojection.

```
int32 constraint[]={1024,640,576,544,528,520,516,514,
513,512};
. . .
for(y=0;y<512;y++)
{
   time1=clock();
   for(x=0;x<512;x++)
   {//compute pixel
     . . .
   }
   time2=clock();
   for (i=0; i<10; i++){
     assert(time2-time1< constraint[i]);
   }
   . . .
}
```

Algorithm 3: Adding timing assertions in a loop to backprojection.

For individual assertions, no additional block RAM was used since assertion failures were transferred via registers rather than using streaming communication on the PROCStar III. The logic overhead of 0.7% is the highest of all the application case studies but is reasonable given that timing calls and multiple assertions were used. The maximum FPGA frequency stayed about the same with an insignificant increase of 0.6 MHz. For a single assertion in a loop, the overhead increased in all categories except for routing. The additional overhead is likely caused by additional complexity of the state machine and the usage of block RAM. The lower routing overhead is probably due to only having to make connections to a single assertion.

Table 5: Individual backprojection timing assertion overhead.

EP3SE260	Original	Assert	Difference
Logic used	48285	49702	+1417
(out of 203520)	(23.72%)	(24.42%)	(+0.70%)
Comb. ALUT	32962	33132	+170
(out of 203520)	(16.20%)	(16.28%)	(+0.08%)
Registers	44098	44595	+497
(out of 203520)	(21.67%)	(21.91%)	(+0.24%)
Block RAM	7114752	7114752	0
(15040512 bits)	(47.30%)	(47.30%)	(0%)
Block interconnect	101317	102740	+1423
(out of 694728)	(14.58%)	(14.79%)	(+0.20%)
Frequency (MHz)	131.9	132.5	+0.6 (+0.45%)

Table 6: Looped backprojection timing assertion overhead.

EP3SE260	Original	Assert	Difference
Logic used	48285	50169	+1884
(out of 203520)	(23.72%)	(24.65%)	(+0.93%)
Comb. ALUT	32962	33459	+497
(out of 203520)	(16.20%)	(16.44%)	(+0.24%)
Registers	44098	44657	+559
(out of 203520)	(21.67%)	(21.94%)	(+0.27%)
Block RAM	7114752	7123968	9216
(15040512 bits)	(47.30%)	(47.37%)	(0.07%)
Block interconnect	101317	102621	+1304
(out of 694728)	(14.58%)	(14.77%)	(+0.19%)
Frequency (MHz)	131.9	131.3	−0.6 (−0.45%)

7.6. Hang Detection. This section shows how in-circuit assertions can be used to detect when an application fails to complete (i.e., hangs), even when software simulation runs to completion. In an effort to speed up a decoder and encoder version of the DES application described in Section 7.2.1, modifications were made that caused the application to complete in software simulation and yet hang on the XD1000. Since Impulse-C does not support *printf* in hardware, assertions were used to provide a heartbeat and "trace" the execution of process on the FPGA. Although this is not a common use of assertions in software, it can be useful to use assertions as a positive indicator rather than a negative indicator when an application is known to crash or hang. *Assert*(0) statements were placed at important points in the code for each FPGA process and *NABORT* was defined to stop the application from aborting. The new code with assertions added was executed via both software simulation and execution on the target platform. After comparing the line numbers of the failed assertions of both runs, it was found that the hang occurred at a memory read, which was causing the process to hang instead of exiting a loop. By identifying the problematic line of code using in-circuit assertions, we were able to debug the application and determined that the memory read should have been a memory write. This correction allowed the process to complete execution.

Table 7: DES hang-detection overhead.

EP2S180	Original	Assertion	Difference
Logic used	21051	21739	+688
(out of 143520)	(14.67%)	(15.15%)	(+0.48%)
Comb. ALUT	12986	13440	+454
(out of 143520)	(9.05%)	(9.36%)	(+0.32%)
Registers	13884	14015	+121
(out of 143520)	(9.67%)	(9.77%)	(+0.09%)
Block RAM	149184	149184	0
(9383040 bits)	(1.59%)	(1.59%)	(0%)
Block interconnect	38924	40241	+1317
(out of 536440)	(7.26%)	(7.50%)	(+0.25%)
Frequency (MHz)	78.8	77.0	−1.80 (−2.28%)

Next, automated hang detection was used on the same problematic DES application. The software hang detector was triggered by the timeout of a communication call. The line number of the software API call was reported back along with the line number (taken before the API call was made) that the hardware process was currently executing. Although hardware hang detection was working correctly in the FPGA, the hardware hang detector was not able to notify the application designer of the problematic line of code since the software API call in conjunction with the erroneous line in the hardware process caused all communication between the CPU and FPGA to stop. To solve this problem, a *sleep* of one second was place above the software API call that was notified as being hung in previous run. The addition of the *sleep* allowed the hardware hang detector to report back the exact line number for the memory read that should have been a memory write. The resource overhead of using automatic hang detection on the Triple-DES application is shown in Table 7. Hang detection had the highest, but still reasonable, percentage of ALUT (0.32%) and routing (0.25%) overhead because of the comparisons and connections made to the state machine of the encoder and decoder hardware process. The assertion pragma, *#pragma assert_FPGA_watch_dog*, was set to a timeout of a hundred million cycles which needed a 30-bit timing register. When using a 64-bit register, the frequency overhead increased to 5.7%. However, such overhead is very pessimistic because even with a 10 GHz clock speed, a 64-bit register supports a maximum timeout of about 58 years. For more typical cases, the frequency overhead should be less than 5.7%.

7.7. Assertion Limitations. The main limitation of in-circuit assertions is that overhead is dependent on the complexity of the assertion statements. For example, a designer could potentially verify a signal processing filter using an assertion statement that performs an FFT and then checks to see if a particular frequency is below a predefined value. In this case, the synthesized assertion would contain a circuit for an FFT, which could have a large overhead. Note that such overhead is not a limitation of the presented synthesis techniques, but rather a fundamental limitation of in-circuit assertions.

To minimize this overhead, we suggest certain coding practices. Whenever possible, designers should use assertion statements that compare precomputed values. Designers should try to avoid consolidating assertions in loops with comparison values stored in arrays because the unnecessary usage of arrays and loops with assertions can increase overhead as shown in Section 7.5. Designers should try to avoid using many logical operators because these operators can cause the HLS tool to create a large state machine to check all combination possibilities of the assertion as shown in Figure 3. By following these guidelines, the assertions will require a minimum amount of resources. Assertion parallelization optimization and resource replication optimization can increase the resource overhead to reduce the performance overhead. Accessing the same array multiple times in an assertion (e.g., $assert(a[i] > a[i-1])$) can be costly either in terms of performance or resource depending if resource replication optimization is used. Even accessing an array only once in an assertion could be costly if the application would normally be using the same array element in the same clock cycle.

8. Conclusions

High-level synthesis tools often rely upon software simulation for verification and debugging executing FPGA processes as threads on the CPU. However, FPGA programming bugs not exposed by software simulation become difficult to remedy once the application is executing on the target platform. Similarly, HLS tools often lack detailed timing-analysis capabilities, making it difficult for an application designer to determine which regions of an application do not meet timing constraints during FPGA execution. The assertion-based verification techniques presented in this paper provide ANSI-C-style verification both for the FPGA and CPU while in simulation and when executing on the target platform. This approach allows assertions to be seamlessly transferred from simulation to execution on the FPGA without requiring the designer to understand HDL or cycle-accurate simulators. The ability of assertions to verify a portion of the application's functionality and debug errors not found during software simulation was demonstrated. ANSI-C timing functions allowed assertions to check application time constraints during execution. Automated hang detection provided source information indicating where software or hardware processes failed to complete in a timely manner. Techniques were shown to enable debugging of errors not found during software simulation that incurred a small area overhead of 0.7% or less and a maximum clock frequency overhead of less than 3% for several application case studies on an EP2S180 and EP3SE260. The presented techniques were shown to be highly scalable, reducing resource overhead of 128 assertions by over 3x, requiring only 1.34% ALUT resources and improving clock frequency by over 18%. The performance overhead of optimized assertions was also demonstrated to be low, with no performance impact observed in the edge-detection case study in terms of frequency degradation or increased cycle usage. A general analysis of performance for single comparison assertions showed that the presented optimizations resulted in a throughput increase ranging from 33% to 100%, when compared to unoptimized assertions, potentially eliminating all throughput overhead. Future work includes further exploration and automation of hang detection.

Acknowledgments

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant no. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Aldec, Altera, GiDEL, Impulse Accelerated Technologies, SRC, and XtremeData, Inc. Special thanks are due to University of Washington ACME Lab for an XD1000 version of the backprojection application that was ported to Novo-G.

References

[1] J. Williams, A. George, J. Richardson, K. Gosrani, and S. Suresh, "Fixedand reconfigurable multi-core device characterization for HPEC," in *Proceedings of High-Performance Embedded Computing (HPEC) Workshop*, Lexington, Mass, USA, September 2008.

[2] Deepchip, "Mindshare vs. marketshare," March 2008, http://www.deepchip.com/items/snug07-01.html.

[3] D. Pellerin and Thibault, *Practical FPGA Programming in C.*, Prentice Hall, Upper Saddle River, NJ, USA, 2005.

[4] D. S. Poznanovic, "Application development on the SRC Computers, Inc. systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, p. 78, April 2005.

[5] Impulse Accelerated Technologies, "Codeveloper's users guide," 2008.

[6] SRC Computers, Inc., "SRC-7 Carte v3.2 C programming environment guide," 2009.

[7] Accellera, "SystemVerilog 3.1a language reference manual," May 2004, http://www.eda.org/sv/SystemVerilog_3.1a.pdf.

[8] Accellera, "OVL open verification library manual, ver. 2.4," March 2009, http://www.accellera.org/activities/ovl.

[9] Accellera, "PSL language reference manual, ver. 1.1," June 2004,http://www.eda.org/vfv/docs/PSL-v1.1.pdf.

[10] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil, "Synthesis of synchronousassertions with guarded atomic actions," in *Proceedings of the 3rd ACM and IEEE International Conference on Formal Methods and Modelsfor Co-Design (MEMOCODE '05)*, pp. 15–24, July 2005.

[11] M. Boulé, J. S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proceedings of the 8th International Symposium on Quality Electronic Design (ISQED '07)*, pp. 613–618, March 2007.

[12] M. R. Kakoee, M. Riazati, and S. Mohammadi, "Enhancing the testability of RTL designs using efficiently synthesized assertions," in *Proceedings of the 9th International Symposium on Quality Electronic Design (ISQED '08)*, pp. 230–235, March 2008.

[13] K. Camera and R. W. Brodersen, "An integrated debugging environment for FPGA computing platforms," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 311–316, September 2008.

[14] Xilinx, "ChipScope pro 10.1 software and cores user guide," March 2008, http://www.xilinx.com/ise/verification/chipscope_pro_sw_cores_10_1_ug029.pdf.
[15] Altera, "Design debugging using the SignalTap ii embedded logic analyzer," March 2009, http://www.altera.com/literature/hb/qts/qts_qii53009.pdf.
[16] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the sea cucumber synthesizing compiler," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 228–237, April 2003.
[17] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-Based Design*, Springer, Berlin, Germany, 2004.
[18] F. Wang and F. Yu, "Assertion-checking of embedded software with dense-time semantics," in *Real-Time and Embedded Computing Systems and Applications*, pp. 254–278, Springer, Berlin, Germany, 2004.
[19] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis techniquesfor in-circuit assertion-based verification," in *Proceedings of the 17th Reconfigurable Architectures Workshop (RAW '10)*, April 2010.
[20] P. Klemperer, R. Farivar, G. P. Saggese, N. Nakka, Z. Kalbarczyk, and R. Iyer, "FPGA implementation of the illinois reliability andsecurity engine," in *Proceedings of the International Conference on Dependable Systemsand Networks (DSN '06)*, pp. 220–221, June 2006.
[21] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer, "An Architectural Framework for Detecting Process Hangs/Crashes," in *Dependable Computing - EDCC 2005*, pp. 103–121, Springer, Berlin, Germany, 2005.
[22] GNU, "The GNU C library reference manual," March 2009, http://www.gnu.org/software/libc/manual/.
[23] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, USA, 1994.
[24] XtremeData Inc., "XD1000 FPGA coprocessor module for socket 940," http://www.xtremedatainc.com/pdf/XD1000_Brief.pdf.
[25] CHREC, "CHREC facilities," http://www.chrec.org/facilities.html.
[26] GiDEL, "PROCStar III PCIe x8 computation accelerator," http://www.gidel.com/pdf/PROCStarIII%20Product%20Brief.pdf.
[27] NIST, "Data encryption standard (DES)," October 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 483681, 12 pages
doi:10.1155/2011/483681

Research Article

Floorplacement for Partial Reconfigurable FPGA-Based Systems

A. Montone,[1] M. D. Santambrogio,[1,2] F. Redaelli,[1] and D. Sciuto[1]

[1] *Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano, Italy*
[2] *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

Correspondence should be addressed to M. D. Santambrogio, marco.santambrogio@polimi.it

Received 20 August 2010; Accepted 20 December 2010

Academic Editor: Aravind Dasu

We presented a resource- and configuration-aware floorplacement framework, tailored for Xilinx Virtex 4 and 5 FPGAs, using an objective function based on *external wirelength*. Our work aims at identifying groups of *Reconfigurable Functional Units* that are likely to be configured in the same chip area, identifying these areas based on resource requirements, device capabilities, and wirelength. Task graphs with few externally connected RRs lead to the biggest decrease, while external wirelength in task graphs with many externally connected RRs show lower improvement. The proposed approach results, as also demonstrated in the experimental results section, in a shorter external wirelength (an average reduction of 50%) with respect to purely area-driven approaches and a highly increased probability of reuse of existing links (90% reduction can be obtained in the best case).

1. Introduction

Nowadays one of the most important design styles in VLSI is represented by Field Programmable Gate Arrays (FPGAs). The standard FPGA design flow starts from an RT-level description of the circuit (e.g., provided by a HDL language) and ends with a configuration file (bitstream) configuring the desired circuit on the device. Much effort has been already placed towards improving the different stages of the design flow, from logic synthesis to placement and routing. On one hand, these problems have been addressed directly by the FPGA vendors [1] as much as by academic works [2]. On the other hand, the floorplanning automation for FPGAs is a current research topic, to face the new challenges provided by FPGAs.

FPGAs particularly present two unique aspects with respect to traditional VLSI designs: resource heterogeneity and reconfigurability. FPGA devices are generally defined using several kinds of resources (e.g., programmable logic cells, memories, multipliers, DSPs, and so on). This requires to take into consideration the resource heterogeneity, to allow each architectural module to be placed in an area region containing all the needed resources. On the other hand, reconfiguration allows the possibility to change the architecture or the application implemented on the FPGA without requiring any physical action on the device. In particular, in partial dynamic reconfigurability, an architecture may change at runtime (i.e., without having a disruption of the provided functionality) a subset of its modules in order to modify its own behavior. Reconfigurability introduces time as a new variable within the floorplanning formulation.

A floorplanner taking into account both of these aspects has to find, for each module (in this paper we use the term *module* to address an architectural component after technology mapping), an area assignment according to target device's capabilities considering that the modules can be replaced later on due to the system needs using the reconfiguration capabilities of the target device. The term floorplacer has been first introduced in the context of traditional VLSI design [3, 4] to emphasize how recently developed algorithms for VLSI design automation face placement and floorplanning tasks concurrently. In a similar spirit, resource management for partially and dynamically reconfigurable FPGAs can benefit from this paradigm. In this paper, which is an extended version of the work presented in [5], we propose to develop a floorplacer for such a device, which will identify groups of modules that are likely to be configured in the same rectangular area and, consequently, identify these chip areas according to the modules' requirements, device capabilities and design objectives. Particularly, in this work, we focus

on optimizing the wirelength since the base floorplacement framework.

The paper is structured as follows. Section 2 introduces the problem and definitions of relevant concepts. Section 3 describes the related work. The target FPGA architecture is described in Section 5, and the experimental results are presented in Section 6. Finally, conclusions and future work are outlined in Section 7.

2. Related Works

The problem of resource-aware floorplanning tailored for FPGAs has been addressed in literature [6, 7]. Montone et al. [7] proposed a floorplanning method that is only aware of resource requirements for the reconfigurable modules. Feng and Mehta [6] proposed an approach, where each architectural module has a list of required resources, hence each module has to be placed in an area region containing all the needed resources. This approach is based on a two steps algorithm: first the execution of Parquet [8] floorplanner with the following resource-aware cost function consisting of a linear combination of Parquet's objective function and the amount of satisfied resource requirements. The result is then refined solving a flow maximization with cost minimization on a purposely built graph. While this approach is currently the state of the art in resource-aware floorplanning for FPGAs, in the majority of cases, the result is not compatible with the FPGA reconfigurability process.

One of the earliest works dealing with reconfiguration-aware floorplanning is [9]. An offline floorplanner is used to decide whether a required functionality should be implemented in hardware on a reconfigurable device or has to be executed via software on a general purpose processor. Different heuristics are proposed to minimize execution time, while reducing implemented modules' fragmentation across the device area.

The first definition of *temporal floorplanning* has been introduced in [10]. According to the author a temporal floorplanning consists of two phases: (i) partitioning and sequencing design modules into design configurations (also called temporal partitioning or scheduling), and spatial positioning of design modules and wiring within the reconfigurable area, in the following named as reconfigurable region, of each configuration.

One of the most important contributions in three-dimensional floorplanning is presented in [11, 12]. They introduce a full 3D floorplanner based on simulated annealing using a three-dimensional transitive closure graph (TCG) and T-trees. They just evaluate the time required by each module to communicate with RAM chips outside the FPGA in order to store results and read input, but they do not evaluate if a found floorplan, particularly its communication infrastructure, is feasible on a given device or not. Furthermore, they do not consider other resources than logic blocks.

A more recent work [13] faces the FPGA floorplanning problem considering partial dynamic reconfiguration, in terms of module reusability. They introduce a variation to the sequence pair representation [14] in order to represent the floorplan in different time frames. The aim if this work is to reduce the quantity of device area reconfigured between different time frames. Given two designs in two different time instants, the authors present a simulated annealing-based floorplanning able to reduce the reconfigured area between the two instants by exploiting reuse of already configured modules.

A brief comparison between previous works is provided in Table 1. The most common limitations of prior work is in their lack of control over the feasibility of the resulting communication infrastructure. Furthermore, most of the related work do not treat resource awareness as a primary goal. However, this is of utmost importance for making module allocation decisions for modern dynamically reconfigurable systems, which contain a variety of heterogenous resources. Our proposed floorplacement method aims to address both of these two important problems, that is, enabling an interconnect and resource-aware design framework.

To the best of our knowledge, there are no prior works dealing with the minimization of wirelength between RFUs and IOBs. The problem of minimizing internal wirelength of RFUs is considered in the past [12], but this approach is based on an estimation of the internal wirelength related to RFUs' aspect ratio and cannot be easily extended in order to support external wirelength. Similarly, by applying the approach introduced in [13], a reuse of links can be obtained, but this approach aims only at re-using RFUs implementing the same functionality, while our approach aims at re-using links between different RFUs connected to the same set of IOBs.

In the following section, we will introduce the basic concepts and definitions related to our proposed floorplacer.

3. Problem Description and Basic Definitions

Given a set of *Reconfigurable Functional Units* (RFUs) the resource- and configuration-aware floorplacement problem consists of finding, for each RFU, a chip area where it can be placed and routed offline and configured at runtime. The entire process is subject to the constraint that each RFU must be associated with an area containing all the required types of resources for its functionality.

Let a RFU be a technologically mapped netlist implementing a required functionality, and a *Reconfigurable Region* (RR) be a rectangular FPGA area where two or more RFUs are going to be placed and routed (at design time) and configured (at runtime) according to the application implemented in the reconfigurable system. The goals of the floorplacer are to

(1) define the number of RRs and associate each RFU with one and only one RR (partitioning task),

(2) find a position of RFUs inside the corresponding RRs (Temporal Floorplacement inside Reconfigurable Region),

(3) find an area constraint for all the RRs inside the FPGA area (RR Floorplacement).

Table 1: Comparison among previous works.

Authors	Comm Infrastructure	Resources Aware	Reconfigurability Aware	Reusability Aware	Algorithm
Feng and Mehta [6]	No	Yes (high res. usage)	No	No	Sim. Annealing over seq. pairs + Flow Max over Flow Graph
Montone et al. [7]	No	Yes	Yes	For logic only	Sim. Annealing
Bazargan et al. [9]	No	No	Yes	No	Sim. Annealing over cubic modules (2d spatial, 1d temporal)
Yuh et al. [11, 12]	Limited, w/High Overhead	No	Yes	No	Sim. Annealing over cubic modules using T-trees and TCG
Singhal and Bozorgzadeh [13]	No	No	Yes	Yes	Sim. Annealing over seq. pairs

The set of input RFUs is represented by a *scheduled task graph*. Such a task graph is divided into time intervals named *static snapshot*. Each static snapshot is characterized by having the same set of configured and running RFUs, during its time interval. Given a RFU m, the function TIME(m) returns the set of static snapshots containing m, that is, static snapshots requiring that m is configured and running. According to these definitions, the scheduled task graph can be considered as a finite state automaton having static snapshots as states and the reconfiguration process as transitions.

Our framework is tailored for Xilinx Virtex 4 and 5 FPGAs [15] that provide access to the off-chip logic through a set of FPGA pins dedicated for communication. The connections between the FPGA's internal logic and external pins are managed through three-state buffers named *Input Output Blocks* (IOBs).

Our proposed approach specifically deals with the interconnection optimization. For FPGAs routing resources are often the limiting resources, hence, achieving better routability in a design may in fact determine the overall feasibility of a design.

Our floorplacement framework manages an objective function based on *external wirelength*, that is, the estimated length of the nets connecting each RFU to the corresponding IOBs. This framework can be used in two different scenarios.

(i) The designer is implementing an application on an FPGA belonging to an existing board with previously assigned IOBs. In this case the framework can help the designer to define area constraints in order to reduce external wirelength.

(ii) The designer has to assign the IOBs and build the board from scratch. In this case the framework can provide feedback to the designer in order to identify or approximate the best IOBs assignment.

In order to formulate this objective function the distance between RFUs and IOBs need to be estimated. Experimental results proved that using the Manhattan distance between the center of the RFU and the position of the IOB provides a good approximation. This definition can be generalized as the distance between one RFU and any location on the chip area as follows:

$$d(\mathrm{RFU}, P) := \left| P_x - \left(\frac{1}{2} \cdot \mathrm{RFU}_w + \mathrm{RFU}_x \right) \right| + \left| P_y - \left(\frac{1}{2} \cdot \mathrm{RFU}_h + \mathrm{RFU}_y \right) \right|, \quad (1)$$

where subscripts x, y, w, and h for RFUs stand for the x coordinate, y coordinate, width and height, of the RFU. RFUs have x and y coordinates corresponding to the bottom-left most corner. P_x and P_y denote the x and y coordinates of an arbitrary location on the FPGA. Similarly the distance between one Reconfigurable Region and a point can be defined. The rationale of the wirelength-driven floorplacement is to constrain a group of RFUs characterized by being connected to a set of IOBs within the same area, thereby, creating a neighborhood. Hence, RFUs communicating with IOBs that are near each other, are kept together and constrained within the same area. This approach results in a shorter external wirelength and a highly increased probability of reuse of existing links. While the first outcome is intuitive, the second requires some elaboration. In a dynamically reconfigurable device a common approach is to allocate two main partitions on the device: the static part and the dynamically reconfigured part, where RFUs will be allocated. The communication infrastructure serving all dynamically inserted functionality is managed by the static part of the design and all the communication, both among RFUs and between RFUs and the static part, are performed by the communication infrastructure exposed through hardware macros. Also the communication between RFUs and IOB has to be managed by the static part of the design and similarly has to be exposed through hardware macros. Let us consider N RFUs accessing an external device through a set of M IOBs, in the worst case the static part has to provide one set of M hardware macros for each RFU, hence, $M \cdot N$ hardware macros. Instead, if the floorplacement is aware of IOB positions, RFUs accessing to the same set of IOBs could be constrained within the same area and hardware macros may be reused by the different RFUs. Consequently, the number of hardware macros that need to be provided by the static part of the design can be drastically lowered toward the theoretical limit of just one set of M hardware macros, that is, just one for each IOB.

Each RFU of the input scheduled task graph can be annotated with information on the position of the IOBs. In order to simplify the management of the IOBs, for each RFU n connected to M IOBs we define a point C, named *centroid*, whose coordinates are given by the arithmetic average of the coordinates of the IOBs:

$$C_x(n) = \frac{1}{M(n)} \sum_{m=1}^{M(n)} \mathrm{IOB}_x(n, m), \tag{2}$$

where $C_x(n)$ is the x coordinate of RFU n, $\mathrm{IOB}_x(m, n)$ is the x coordinate of the mth IOB of RFU n, and $M(n)$ is the number of IOBs of RFU n. The same holds for y coordinates. The centroid represents the ideal position where an RFU should be positioned in order to minimize the external wirelength, that is, floorplacing the RFU such that its own geometrical center coincides with the centroid will result in minimizing the external wirelength. Only RFUs connected to the IOBs have an associated centroid.

In the following section we will first give an overview of the dynamic reconfiguration mechanism of our target FPGA architecture. Next, in Section 5 we will describe our floorplacer which can effectively manage the resources of this reconfigurable architecture.

4. FPGA Reconfiguration and Target Architecture

This section introduces the reconfiguration process from the FPGA's physical point of view. Section 4.1 presents the physical limits of the reconfiguration process, while Section 4.2 shows the design flow proposed by Xilinx for reconfigurable architectures. Finally, Section 4.3 defines the target architecture considered by this work.

4.1. Smallest Reconfigurable Region. The reconfiguration process described here relates to the latest generation of Xilinx FPGAs, that is, Virtex 5 devices. According to their datasheets [16] all the devices of this family share a common structure. The entire FPGA is made of programmable logic (commonly referred to as CLBs for this FPGA family) and periodically distributed Block RAMs (BRAMs), while all the other resources (such as DSP blocks) are placed along the vertical edges of the FPGA. User logic is implemented combining all these resources and connecting them using channels and switchboxes. The information about device configuration is described in a binary configuration file named *bitstream*: logical functions implemented by CLBs (i.e., the content of the lookup tables implementing a logical function with 6 inputs and 1 output), BRAMs content, routing information (i.e., logical status of switches managing the interconnections), and so on. The smallest reconfigurable element is 1 row high and 1 CLB wide and is referred to as *frame*, each frame is addressed by a row number (from 0 to 3) and column number (expressed in CLBs). The bitstream follows the device topology, hence inside a configuration bitstream one can easily identify data configuring a specific frame.

As previously mentioned, the frame is the smallest FPGA area that can be configured independently of the others. Given a generic module h_{module} rows high and w_{module} CLBs wide, the smallest area that can be involved in the reconfiguration process is a rectangle with $h_{\text{small-area}}$ rows height and $w_{\text{small-area}}$ CLBs width such that the following.

(i) The height in rows is the smallest integer greater than the module's height in rows

$$h_{\text{small-area}} = \lceil h_{\text{module}} \rceil \qquad [\text{rows}]. \tag{3}$$

(ii) The width in CLBs is the smallest integer greater than the module's width in CLBs

$$w_{\text{small-area}} = \lceil w_{\text{module}} \rceil \qquad [\text{CLBs}]. \tag{4}$$

For example, given a module requiring for its placement 1.5 rows height and 30 CLBs width, the smallest area that can be assigned to this module, considering the reconfiguration constraints, is a rectangle of height and width of 2 rows and 30 CLBs, respectively. Configuration and reconfiguration processes take place by writing the bitstream inside the FPGA's *configuration memory*. In the case of partial reconfiguration the bitstream carries information addressing the frames that are going to be replaced by the carried data.

4.2. Xilinx Partial Reconfiguration Design Flow. Due to the fact that Xilinx FPGA Virtex families are the target devices of this work, the Xilinx Partial Reconfiguration (PR) design flow [17] will be briefly introduced here. This flow is compatible with dynamic reconfiguration. Let a *Reconfigurable Functional Unit* (RFU) be a technologically mapped netlist implementing a required functionality and a *Reconfigurable Region* (RR) be a rectangular FPGA area where two or more RFUs are going to be placed and routed (at design time) and configured (at runtime) according to the application implemented in the reconfigurable system. PR allows the definition of a set of nonoverlapping RRs. All the *static logic* (i.e., all logic that will always remain configured, including the glue logic) is placed outside the reconfigurable regions, while they are allowed to use routing resources intersecting and even crossing RRs (the use of routing resources crossing RRs is the most relevant case). Figure 1(a) provides an example of a reconfigurable architecture developed with the PR design flow. Note how the reconfigurable regions are aligned to the grid defined by rows and CLBs according to PR requirements.

According to the PR flow, hardware macros can be placed on the boundaries of RRs in order to define *pins* where RFUs can hook themselves. Such macros are made with pairs of CLBs, one side of the CLB pair is connected to an RR signal, while the other is connected to a static logic signal. Previous design flows [18] required static logic being *placed* and also *routed* outside RRs, while the PR relaxes this constraint.

4.3. Target Architecture. The target architecture considered in this work is based on the PR design flow and has a static part implementing the communication infrastructure providing

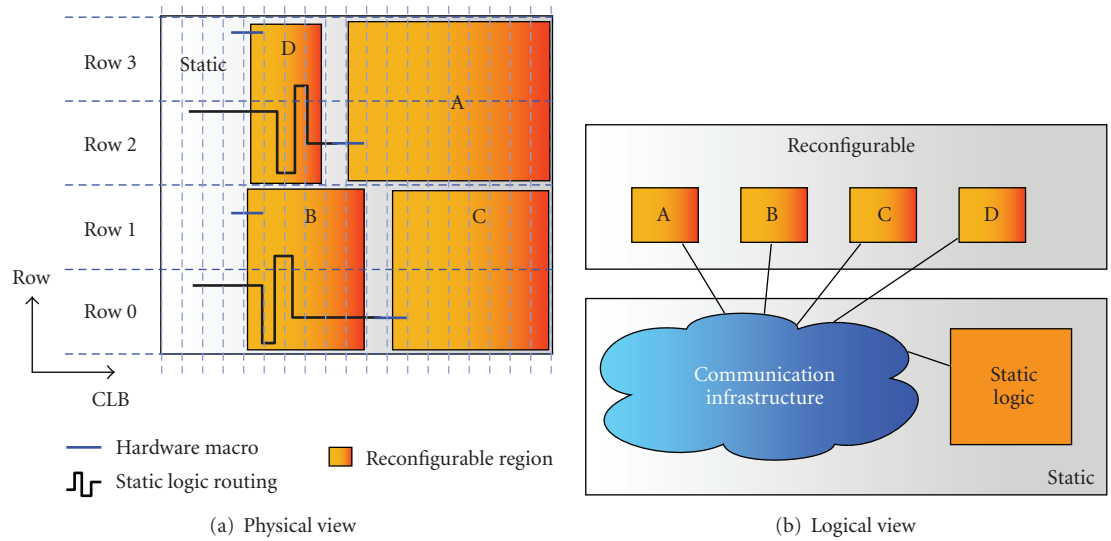


Figure 1: Target Reconfigurable Architecture based on PR: (a) physical and (b) logical views.

a number of interfaces at least equal to the number of RRs (each RR may provide more than one link to the communication infrastructure). Each RFU can communicate with other configured modules or with the static part by just hooking up to the hardware macros corresponding to the communication infrastructure interfaces. The most general view of the target architecture is given in Figure 1: from a physical point of view RRs communicate with the static part through to PR stand-alone static-logic nets routing, while from a logical point of view the static logic is responsible for managing intermodule communication implementing different communication infrastructures (such as bus-based, NoC-based, point to point, and so on). In conclusion, PR allows a more scalable communication infrastructure for partial dynamic reconfigurable architectures, since it allows an RFU to communicate with static logic regardless of the position of other RFUs. This approach simplifies the communication infrastructure design, while the frame size still needs to be considered during RRs floorplacement in order to prevent the configuration of an RFU inside an RR from interfering with other RFUs being executed on other parts of the FPGAs.

5. The Floorplacement Framework

In this section, we introduce our proposed framework for solving the floorplacement problem targeting the dynamically reconfigurable architecture and the reconfiguration technology described in the previous section.

The floorplacement framework accepts as input a scheduled task graph (TG) composed of a node for each RFU. A task graph is a Directed Acyclic Graph whose nodes represent a single task of a given application (or part of an application). A TG representation of an application has been chosen according to most of the related works [9–11, 13]. The TG can be scheduled according to different requirements (e.g., timing requirements or target device). Dividing time into *time slots* the concept of static snapshot can be defined as the set of TG's nodes (i.e., tasks) that must be configured and must be running in a given time slot. A partial dynamic reconfigurable system can be seen as a finite state automaton according to the following definition.

(i) *States*. There is one state for each time slot (hence one for each static snapshot).

(ii) *Transitions*. The transition is a reconfiguration process.

After presenting the chosen scheduling technique, we will describe three algorithms that comprise our framework [5]. The floorplacement process starts with a *Partitioning* step, where RFUs are first grouped into RRs according to two criteria, wirelength for external routing to IOBs and utilization of resources. In the second step, once the partitions have been computed, the position of each RFU inside the corresponding RR needs to be determined. This is performed by the *Temporal floorplacement* step inside RRs. Finally, the design is completed by placing the RRs on the FPGA using the *Reconfigurable Regions Floorplacement* step.

5.1. Static Scheduling Phase. The heuristic used to compute the scheduled task graph has been defined starting from the Napoleon scheduler [19] and the ILP formulation proposed in [20]. This heuristic is a reconfiguration-aware scheduler for dynamically partially reconfigurable architectures that exploits *configuration prefetching*, *module reuse*, and also *antifragmentation* techniques.

In the following, nodes that have to be scheduled while their ancestors have already been will be called *available nodes*. The heuristic performs a list-based scheduling using as priority function the ALAP value of the tasks. This function has been slightly changed: an available task can be scheduled if (i) it has an ALAP value greater than the minimum ALAP value of the available nodes, (ii) if the possibility of

```
sLength ← 0
t ← 1
g ← readGraph()
setALAP(g)
RNs ← getRootNodes(g)
while ∃ not scheduled tasks do
   Control possibility of reuse for available tasks in RNs
   if ∃ not scheduled tasks then
      avTask ← getFirstALAPAvailableNode(RNs)
      endT ← findEndTime(avTask,t)
      while all the available nodes in RNs have been observed do
         if ∃ a position on the FPGA for avTask then
            avTask.terminationTime ← endT
            avTask.schedulingTime ← t
            avTask.setScheduled ← true
            if sLength < endT then
               sLength ← endT
            end if
            for all avTask child nodes chTask do
               if All chTask parents have been scheduled then
                  RNs ← RNs + chTask
               end if
            end for
            Control possibility of reuse for available tasks in RNs
            avTask ← getNextALAPAvailableNode(RNs)
         end if
      end while
   end if
   t ← nextControlStep
end while
```

Algorithm 1: Heuristic pseudocode.

scheduling for all the available tasks with an ALAP value less than its own has been verified and (iii) if there is also *enough space* onto the FPGA.

Two antifragmentation techniques have been designed.

(i) *Farthest Placement*. When a module needs to be reconfigured, it will be placed in the farthest position with respect to the center of the FPGA. We have to do this because when a large module (a module which is demanding many hardware resources) has to be placed, maintaining the emptiness of the center of the FPGA could increase the probability of placing large modules quickly. The same concept is applied to those tasks exploiting *module reuse*: when more than one module is available to be used on the FPGA, the farthest one with respect to the center of the FPGA is selected.

(ii) *Limited Deconfiguration*. The deconfiguration policy leaves on the FPGA all modules that are not involved directly in the cleaning process (the creation of enough contiguous space for a new task, increasing the possibility of reuse of those modules).

Algorithm 1 shows the pseudocode of the proposed algorithm.

The most important functions used in Algorithm 1 are as follows.

(i) ∃ *a Position on the* FPGA *for avTask*. This function involves the placer that, using *antifragmentation techniques*, tries to place the current task *avTask*. This function takes into account also that if a module is being reconfigured, no other modules can be reconfigured onto the FPGA, and in this case it returns *false*. This function has not to be confused with a later phase of the proposed framework. In the scheduling phase this function is used to make the scheduler aware of the resource, while in the next phase the it will be used to properly manage the RFUs into the RRs.

(ii) *Control the Possibility of Reuse for Available Tasks in Root Nodes* (RN). The pseudocode of this function is presented in Algorithm 2, and it simply considers all the available tasks in ALAP order and verifies for each one if there is a module available to be reused.

(iii) *nextControlStep*. This function returns the next time assignable to a task. This is done to reduce the complexity of the algorithm by reducing the number of iterations in the external *while* cycle. Not all the time instants are available to assign a task:

```
avTask ← getFirstALAPAvailableNode(RNs)
while ∃ an available task not yet considered do
  endT ← findEndTimeReusedTask(avTask,t)
  if ∃ a module usable by avTask then
    avTask.terminationTime ← endT
    avTask.schedulingTime ← t
    avTask.setScheduled ← true
    if sLength < endT then
      sLength ← endT
    end if
    for all avTask child nodes chTask do
      if All chTask parents have been scheduled then
        RNs ← RNs + chTask
      end if
    end for
  end if
  avTask ← getNextALAPAvailableNode(RNs)
end while
```

Algorithm 2: Reuse function pseudocode.

(a) when a task is being reconfigured, the scheduler cannot reconfigure any other task;

(b) when there is not enough available area on the FPGA to place any task, the scheduler has to wait for the termination of at least one running task;

(c) when a module exploits the *module reuse* concept and there are no available modules of the same type on the FPGA, the scheduler has to wait for the termination of at least one of those modules to schedule the selected tasks. For this reason *nextControlStep* assigns to the current time t a value given by *1* plus the minimum time between the last time in which the reconfiguration device is used and the first termination time of the tasks running on the FPGA.

In the worst case, the algorithm assigns only one task per time instant so the external *while* is executed $O(n)$ times where n is the number of tasks in the task graph, the control for reused tasks takes $O(f_o)$ time, where f_o is the maximum fanout of the nodes of the task graph, the internal *while* is executed $O(f_o)$. The functions that return the tasks in ALAP order can be designed by implementing the binomial search in $O(1)$ time, but in this case the process of inserting a new available node into RNs will take $O(\log f_o)$. Also the *for* used to verify the availability of the children nodes of *avTask* is executed $O(f_o)$ times. Hence the complexity of the algorithm in the worst case is $O(n f_o^2 \log f_o)$.

5.2. Partitioning into RRs. Given N RRs, the N-RRs partitioning problem consists of finding a surjective binding $c_{m,n}$ of RFUs into RRs (i.e., each RFU has to be bound to one and only one RR and each RR has to contain at least one RFU). Algorithm 3 firstly aims at grouping together

```
Buckets B;
For all Externally connected RFU r do
  B.add(r);
end for
Wirelength-driven-partition(B);
Fix-existing-associations(B);
for all Remaining RFU q do
  B.add(q);
end for
Resource-driven-partition(B);
```

Algorithm 3: Partitioning into RRs.

externally connected RFUs (i.e., RFUS that are connected to IOBs) having nearest centroids and keeping RFUs with distant centroids in different RRs. We refer to this as the (*wirelength-driven partitioning*).

Secondly, the remaining RFUs are partitioned to minimize the variance of RRs' resource requirements along different static snapshots. In other words, for a given RR, the algorithm tries to keep the amount of resources needed by the RFUs configured and running inside the considered RR constant as much as possible across different static snapshots. We refer to this as the *resource-driven partitioning*.

The problem of wirelength aware partitioning of RFUs into RRs, can be reduced to the problem of clustering the corresponding centroids in a two-dimensional space (i.e., chip area). Each identified cluster is associated with one and only one RR and the RFUs are partitioned into RRs according to the association between their corresponding centroids and clusters (a partition belongs to an RR if and only if its centroid belongs to the cluster associated with the RR). Once the wirelength-driven partitioning has been performed, the created partition is used as an initial solution by the resource-driven partitioner to further partition the RFUs that are not externally connected. This means that the surjective binding $c_{m,n}$ is no longer modified for externally connected RFUs.

While data-mining algorithms provide several tools to solve the clustering problem (like the well-known k-means or fuzzy-k-means algorithm), they are primarily geared towards very large datasets. For our purposes, real life task graphs consist of fewer than a hundred RFUs and only a few of them are connected to IOBs. Hence, we adopted a simulated annealing-based approach.

Data Structure. Let us consider a bucket data structure having a set of buckets B_n for each Reconfigurable Region n. A given RFU m belongs to a bucket B_n (and only to that one) if and only if m is going to be placed in the Reconfigurable Region n at TIME(m).

Annealer's Moves. Let the simulated annealer's moves be the following.

(i) *Randomly Move One RFU*. Move one module between two buckets: randomly pick up a module $m \in B_n$ and move to a bucket $B_{n'}$ where $n \neq n'$. This

move can be performed if and only if B_n contains another module $m' \neq m$.

(ii) *Swap Two RFUs*. Swap modules belonging to different buckets: randomly pick up two modules m and m', respectively, $m \in B_n$ and $m' \in B_{n'}$ such that $B_n \neq B_{n'}$. The move consists in swapping modules' buckets such that $m \in B_{m'}$ and $m' \in B_m$.

Once the partitions have been computed, the position of each RFU inside the corresponding RR needs to be determined.

5.3. Temporal Floorplacement inside Reconfigurable Regions. The aim of the Temporal Floorplacement inside Reconfigurable Regions (TFiRR) is to compute, for each RR, a set of height-width pairs describing rectangular areas where all RFUs bound to this RR can be successfully floorplaced. In this phase the final on-chip position of the rectangular area is not considered. For a target FPGA device that is divided by up to k rows for reconfiguration, the goal of this algorithm is to determine for each RR n a set of pairs

$$\Omega = \left\{ \left\langle n_h^1, n_w^1 \right\rangle, \left\langle n_h^2, n_w^2 \right\rangle, \ldots, \left\langle n_h^k, n_w^k \right\rangle, \ldots \right\} \tag{5}$$

such that an eventual actual placement of this RR on the device given as $A_n = \langle n_x, n_y, n_h^i, n_w^i \rangle$ for all n_x, n_y, i, results in a feasible floorplacement independently of the final position n_x and n_y decided for this RR. Here, h, w, x, and y stand, respectively, for the height, width, and the two coordinates of the bottom-leftmost corner of the rectangular area.

Consequently, for each RR n, the set of height-width pairs Ω can be described by providing just four elements (due to technological constraints related to Xilinx Virtex 4 and 5 FPGAs that are divided in 4 rows)

$$\Omega = \{\langle 1, n_w^1 \rangle, \langle 2, n_w^2 \rangle, \langle 3, n_w^3 \rangle, \langle 4, n_w^4 \rangle\}, \tag{6}$$

where n_w^1 is the smallest width that RR n, floorplaced in 1 row, should have in order to feasibly host all the associated RFUs, n_w^2 is the smallest width that RR n (floorplaced in 2 rows) should have in order to feasibly host all the associated RFUs, n_w^3 and n_w^4 are the smallest widths that RR n, floorplaced in 3 and 4 rows, respectively, should have in order to feasibly host all the associated RFUs.

The core of the TFiRR step is the computation of the pairs $\langle i, n_w^i \rangle$. In order to find, for a given height i, the minimum feasible width n_w^i, the algorithm has to check that every RFU can be successfully floorplaced inside the area described by the pair $\langle i, n_w^i \rangle$, that is, for each RFU the algorithm has to provide a height, width, and a position within the RR n.

Such a problem is itself three-dimensional (i.e., two spatial dimensions and a temporal one). In order to simplify the problem the following assumption is introduced: *all RFUs' heights are equal to RRs' heights*. Fixing the height dimension of the RFUs, the problem is reduced to a bidimensional packing problem such that the static snapshot and the width are the only two considered coordinates. Given a RFU m, and a height i, the smallest position-independent width required by the RFU in order to be hosted inside an area of height equal to i rows, can be easily computed by taking into account the FPGA's resources periodic distribution. The TFiRR algorithm works as follows.

For each RR n and each possible height $i \in \{1, 2, 3, 4\}$

(1) consider RFU m such that $c_{m,n} = 1$ (i.e., RFUs belonging to RR n), let RFUs' height be equal to RR's height, then, compute the minimum feasible width of RFUs m;

(2) pack all the RFUs inside the RR in order to *minimize* the maximum width of RR n.

The packing of the RFUs can be performed with a zero temperature simulated annealing (ZT-SA) algorithm. For each static snapshot p an ordered list of RFUs m, such that $p \in \text{TIME}(m)$, is kept. The RFUs are ordered from the leftmost to the rightmost with respect to the RR's area. The following moves are applied.

(1) *Randomly Move an RFU*. Randomly pick an RFU and move to an integer position belonging to the interval $[0, \text{width}]$, where width is the current width of the RR.

(2) *Randomly Swap Two Concurrent RFUs*. Randomly pick two RFUs m' and m'', being concurrently configured and running in at least one static snapshot and swap their position inside RR.

In order to keep the floorplacement compact, each step of the annealer is followed by a *compression* function that computes, for each RFU, the leftmost feasible solution preventing overlaps between RFUs. The computation of the objective function is the most expensive operation, requiring in the worst case $\Theta(R \cdot P)$ time (where R is the number of RFUs), but experimental data on randomly generated partitions indicate that in practice such complexity is asymptotic $O(R \cdot \log R)$. From a memory complexity point of view the algorithm requires only the management of a list for each static snapshot, hence the memory requirements are $\Theta(R)$.

The quality of this algorithm at first seems to be strictly related to the quality of the partition provided by the previous step. Our experiments showed that TFiRR applied on nonpartitioned task graphs can reach the results of TFiRR applied to a partitioned task graph by increasing the number of iterations by at least two orders of magnitude. On the other hand, the difference between the objective functions remains fairly low. We observed degradations ranging between 1–5%.

At the end of this second step, each resulting RR is annotated with a centroid whose coordinates are given by the arithmetic average of the corresponding coordinates of the RFUs associated with the considered RR. This identifies the ideal position where each RR should be placed in order to globally minimize the external wirelength of the associated RFUs. This particular formula for computing the centroid places more emphasis on the most heavily utilized IOBs. For example, if three RFUs are connected only to the USB interface, then all of them will have the same centroid C_{USB} that will occur three times in the set of centroids associated with the RR. In the third and final step, the centroids of the RRs will be used during the final floorplacement of RRs.

5.4. Reconfigurable Regions Floorplacement. The aim of this step is to define, for each RR n, an area

$$A_n = \langle n_x, n_y, n_h, n_w \rangle. \quad (7)$$

The algorithm has to choose one $\langle n_w, n_h \rangle$ couple, for each RR n, out of the set provided by the TFiRR step. Furthermore, it has to determine the specific x and y positions on the FPGA area. According to classical floorplanning this task can be performed through simulated annealing. The RR Floorplacement algorithm is divided in two steps: the first one floorplacing the RRs connected to IOBs (wirelength-driven RRs floorplacement) and the second one floorplacing the remaining RRs (area-driven RRs floorplacement).

Data Structure. To represent the floorplacement, a Horizontal Constraint List (HCL) is used for each row of the device. The HCL for row r is a list containing all the RRs occupying row r and ordered by increasing n_x.

Objective Functions. The first step is characterized by an objective function that must take directly into account the wirelength:

$$\Gamma = \left(\sum_{r \in \mathrm{RR}} d(r, C(r)) \cdot \#\{\mathrm{RFU} \in r\} \right)^f, \quad (8)$$

where $d(r, C(r))$ represents the distance of the RR from its ideal position (centroid), $\#\{\mathrm{RFU} \in r\}$ is the number of RFUs connected to IOBs belonging to the rth RFU, while f is a positive number. If f is small, this indicates that the floorplacement is feasible and if it is large the floorplacement is not feasible. The goal of the floorplacement is to minimize Γ. Note that the objective function is weighted by the number of externally connected RFUs. This means that RRs containing more RFUs connected to the external world would benefit from a partitioning in the neighborhood of the centroid. Once the wirelength-driven RR floorplacement has been performed, the remaining RRs can be floorplaced by a purely area-driven algorithm. This second step is guided by an objective function involving free area and feasibility of the final floorplacement. Given an RRs' floorplacement, the following quantities are defined: *Negative area slack* (N), that is, the area of the floorplan crossing target device boundaries, and *positive area slack* (P), that is, the greatest contiguous free area starting from the right-top most corner of the device and with nonincreasing width going bottom-ward. Figure 2 shows an example of such slacks. Given such slacks, the RR Floorplacement objective function is defined as follows (where $M \in \mathbb{N}$ and greater than the number of frames on the target FPGA area):

$$\Theta = P - M \cdot N. \quad (9)$$

This objective function Θ is positive if the floorplacement is feasible (i.e., $N = 0$), otherwise it is negative (because $P < N$). The aim of the annealer is to maximize Θ, consequently, to provide a feasible floorplacement maximizing the contiguous FPGA area left free for static logic.

Annealers' Moves. Given the HCL data structure, the following moves are defined.

(1) *Randomly Swap Two RRs*. Randomly choose two RRs and swap their positions.

(2) *Move an RR to a Randomly Chosen Position*. Randomly pick an RR n and two coordinates $\langle \overline{x}, \overline{y} \rangle$ belonging to the device area.

(3) *Span a Randomly Chosen RR over Rows*. Randomly choose an RFU, having height less than number of ROWS, and increase its height by 1 row.

(4) *Unspan a Randomly Chosen RR*. Randomly choose an RFU, having height greater than 1, and decrease its height by 1 row. It is the inverse of the span move.

It can be noticed how the floorplacement of a large number of small functional units is easier than the floorplacement of a small number of large functional units, as shown in Figure 3. A bad choice of floorplacement of a large functional unit during the early stages of the floorplacement is difficult to correct in the later steps, particularly when temperature decreases rapidly and each correcting move is likely to be rejected because it results in a worse objective function.

5.5. Identifying Optimal Number of Partitions. The three steps described above comprise our floorplacement framework. The overall framework relies on the concept of partitioning, hence, some final remarks on how we control the granularity of these partitions will be useful. In order to identify the most suitable number of partitions let us consider the maximum number of concurrently configured C_C RFUs, that is, how many RFUs are present at most in one static snapshot in any partition. We define this quantity as follows:

$$C_C = \max\left\{ n_{p,b} \mid \#\{\mathrm{RFU}\ r \mid p \in \mathrm{TIME}(r) \wedge r \in B_b\} \right\}, \quad (10)$$

where $B_1, \ldots, B_{\#\mathrm{partitions}}$ represent the different output partitions. C_C can be considered as a good metric to describe the complexity of the entire partition. Let Γ be defined as the global normalized variance in resource requirements, used to represent the heterogeneity of the partitions. We have observed experimentally that the number of partitions minimizing the product of Γ and C_C provides (in resource-driven partitioning algorithm) a good tradeoff between partition complexity and intra-partition variance.

6. Experimental Results

Our proposed approach for the wirelength and resource management has been validated both on randomly generated task graphs and real-world applications from the domain of data processing for biomedical applications (i.e., collecting data from sensors, performing some preprocessing like FIR filtering, computing error detection codes, and sending data through a network). General consideration about the performance in finding the optimal results can be found

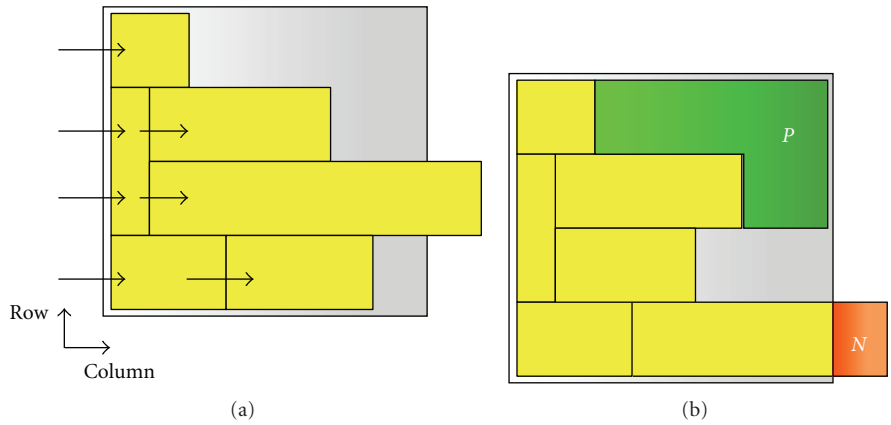


Figure 2: (a) A variation of horizontal constraint graph used for floorplan representation. (b) Negative (N) and positive (P) area slacks (the empty space on the second row is not included in P because it has a width greater than one of the empty areas in the upper row).

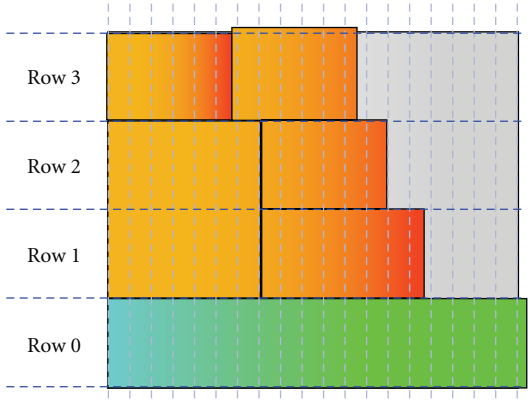


Figure 3: Limits of the floorplacement of big modules.

Table 2: Quality metrics and the summary of the variation in these metrics compared with existing work [7].

Metric	Variation/Value
External wirelength	Reduction in external wirelength ranging between (90, 30)% compared to existing area-driven method
Links	Reduction in number of links required ranging between (90, 0)% compared to existing area-driven approach
Blank Area	(5, 35)% of the final floorplacement

in [21]. The results introduced by this new approach are application dependent, therefore we will describe three different metrics used to evaluate the results. For each metric, we will provide the description of the task graphs. (Table 2 shows a summary of the metrics and a summary of our comparison with an existing floorplacement method that is only resource aware, but does not consider the wirelength implications of resource management [7].)

Our results confirm that introducing the wirelength awareness indeed improves the interconnect cost significantly. We observed a reduction of 90% in external wirelength in the best case and an average reduction of 50%. Task graphs with few externally connected RRs lead to the biggest wirelength reduction. On the other hand, the reduction for external wirelength in task graphs with many externally connected RRs with near centroids is less. In such task graphs only few RRs can be placed near their centroids and the other RRs have to be placed far away. The number of hardware macros provided by the static part can be reduced by 90% in the best case. Task graphs containing several RFUs accessing (in different static snapshots) the same set of IOBs (e.g., RFUs connected to the same external interface like USB) would particularly benefit from our approach, while task graphs having all the RFUs connected to a distinct sets of IOBs would not benefit from links reduction (this is the 0% reduction case referred in Table 2).

Figure 4 shows how our proposed approach drastically reduces the number of required links.

In this figure we also observe one weakness of our approach. The most relevant drawback of our approach is referred to as the *blank area* problem (i.e., the amount of area being surrounded by RRs but not assigned to any RR). Let us consider a set of RRs, each one being externally connected and apply our approach several times, each time increasing the percentage of RRs that are considered (by the floorplacer) as attached to IOBs. During the first iteration no RR is considered as attached to IOBs, while during the last iteration all RRs are considered as attached to IOBs. Figure 5 plots the percentage of the final floorplacement that remains as blank area with respect to the percentage of RRs considered as attached to IOBs as a result of this experiment.

When no RR is considered as externally connected a 5% blank area is obtained (same as the purely area-driven approach). The peak is obtained when half of the RRs are managed by the wirelength-driven algorithm and the other half by the area driven, in such a case the blank area may reach 30–35% of the final floorplacement. On the other hand, the blank area is generally divided in no more than three or four areas that are wide enough to be used by the static part of the design according to the PR design

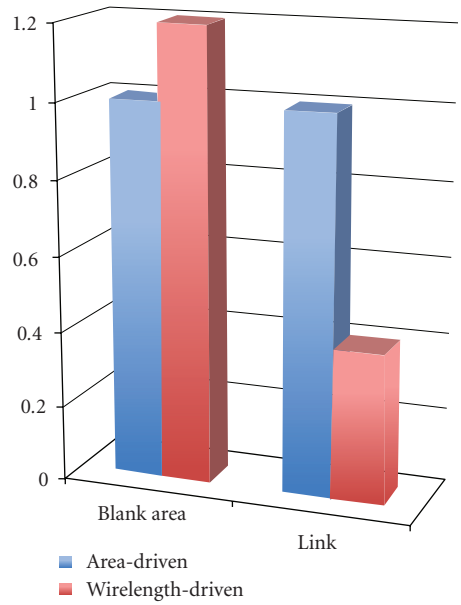


Figure 4: Comparison of the area and wirelength metrics among the area-driven and wirelength-driven approaches (normalized w.r.t. area-driven approach).

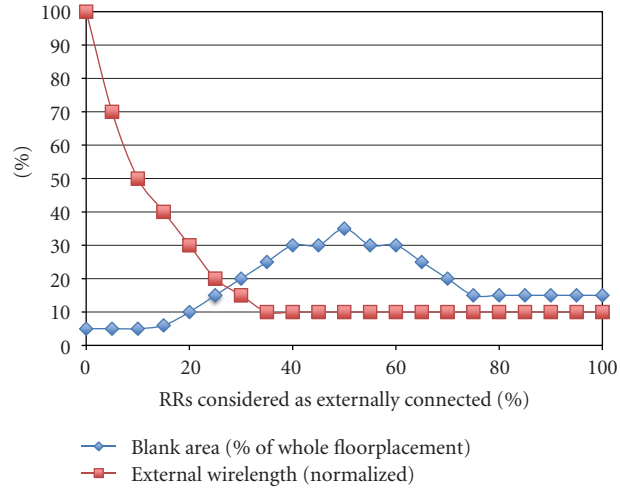


Figure 5: Example of wirelength reduction (normalized) and blank area left (percentage of the final floorplacement) for the floorplacement of only externally connected RRs plotted with respect to the percentage of RRs that are considered as externally connected by the algorithm.

flow. Figure 5 shows also how the first externally connected RRs obtain a great wirelength improvement, while the latest one cannot obtain such improvement due to the previously introduced nonoverlapping constraints.

Therefore, we observe that our approach can yield significant improvements in the cost of the communication infrastructure by tradingoff a reasonable amount of blank area. Using our framework, the designer can chose between (a) considering more RRs as externally connected, thereby, decreasing external wirelength or (b) considering that beyond a certain point blank area overcomes benefits provided by the wirelength-driven approach. Hence, the designer may decide to floorplace in a wirelength-driven way only for the most relevant RRs leaving the others to be floorplaced by an area-minimizing approach. Finally, we observe that the blank area problem is not an issue for task graphs requiring most of the resources of the target FPGA (because the feasibility of the floorplacement requires as much area as possible to be used, hence blank area is reduced as a consequence) and for task graphs having few RFUs connected to IOBs (or many of them connected to the same IOBs).

7. Conclusions and Future Work

In this paper, we presented a resource- and configuration-aware floorplacement framework, tailored for Xilinx Virtex 4 and 5 FPGAs, using an objective function based on *external wirelength*. The proposed approach has achieved a shorter external wirelength and a highly increased probability of reuse of existing communication links. The reduction in wirelength ranges from 30% to 90% in comparison to a purely area-minimizing approach. Task graphs with few externally connected RRs lead to the biggest decrease, while external wirelength in task graphs with many externally connected RRs show lower improvement. Future improvement for the work presented in this paper can be done in considering a hybrid approach between the area and the wire length solution. Furthermore we aim at directly addressing the blank area problem by modifying the objective function or floorplacing all the RRs, not connected to the IOBs, around the RRs connected to IOBs, trying to keep the RRs as close as possible in order to further reduce the blank area.

References

[1] *ISE 9.2i Manual*, Xilinx Incorporation, 2007.

[2] V. Betz and J. Rose, *VPR: A New Packing, Placement and Routing Tool for FPGA Research*, Springer, London, UK, 1997.

[3] J. A. Roy, S. N. Adya, D. A. Papa, and I. L. Markov, "Min-cut floorplacement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1313–1326, 2006.

[4] S. N. Adya, S. Chaturvedi, J. A. Roy, D. A. Papa, and I. L. Markov, "Unification of partitioning, placement and floorplanning," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers (ICCAD '04)*, pp. 550–557, November 2004.

[5] A. Montone, M. D. Santambrogio, and D. Sciuto, "Wirelength driven floorplacement for FPGA-based partial reconfigurable systems," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, pp. 1–8, 2010.

[6] Y. Feng and D. P. Mehta, "Heterogeneous floorplanning for FPGAs," in *Proceedings of the IEEE 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design*, vol. 2006, pp. 257–262, 2006.

[7] A. Montone, F. Redaelli, M. D. Santambrogio, and S. O. Memik, "A reconfiguration-aware floorplacer for FPGAs," in

Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08), pp. 109–114, December 2008.

[8] S. N. Adya and I. L. Markov, "Fixed-outline floorplanning: enabling hierarchical design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 1120–1135, 2003.

[9] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "3-D floorplanning: simulated annealing and greedy placement methods for reconfigurable computing systems," in *Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping (RSP '99)*, pp. 38–43, June 1999.

[10] M. Vasilko, "Dynasty: a temporal floorplanning based cad framework for dynamically reconfigurable logic systems," in *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL '99)*, pp. 124–133, Springer, London, UK, 1999.

[11] P. H. Yuh, C. L. Yang, and Y. W. Chang, "Temporal floorplanning using the T-tree formulation," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers (ICCAD '04)*, pp. 300–305, November 2004.

[12] P. H. Yuh, C. L. Yang, and Y. W. Chang, "Temporal floorplanning using the three-dimensional transitive closure subGraph," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 4, article 37, 2007.

[13] L. Singhal and E. Bozorgzadeh, "Multi-layer floorplanning on a sequence of reconfigurable designs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–8, 2006.

[14] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 12, pp. 1518–1524, 1996.

[15] *Virtex 5—Family Overview*, Xilinx Incorporation, 2007.

[16] Xilinx Inc., "Virtex-5 user guide," Tech. Rep. ug190, Xilinx Inc., 2007, http://www.xilinx.com/bvdocs/userguides/ug190.pdf.

[17] *Partial Reconfiguration User Guide*, Xilinx Incorporation, 2010.

[18] *Xilinx Application Note 290*, Xilinx Incorporation, 2007.

[19] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 519–522, March 2008.

[20] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto, "Partitioning and scheduling of task graphs on partially dynamically reconfigurableFPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 662–675, 2009.

[21] A. Montone, F. Redaelli, M. D. Santambrogio, and S. O. Memik, "A reconfiguration-aware floorplacer for FPGAs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 109–114, 2008.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 121353, 16 pages
doi:10.1155/2011/121353

Research Article

Operating System for Runtime Reconfigurable Multiprocessor Systems

Diana Göhringer,[1] Michael Hübner,[2] Etienne Nguepi Zeutebouo,[1] and Jürgen Becker[2]

[1] *Object Recognition Department, Fraunhofer IOSB, 76275 Ettlingen, Germany*
[2] *ITIV, Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany*

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

Received 20 August 2010; Revised 30 January 2011; Accepted 14 February 2011

Academic Editor: Aravind Dasu

Operating systems traditionally handle the task scheduling of one or more application instances on processor-like hardware architectures. RAMPSoC, a novel runtime adaptive multiprocessor System-on-Chip, exploits the dynamic reconfiguration on FPGAs to generate, start and terminate hardware and software tasks. The hardware tasks have to be transferred to the reconfigurable hardware via a configuration access port. The software tasks can be loaded into the local memory of the respective IP core either via the configuration access port or via the on-chip communication infrastructure (e.g. a Network-on-Chip). Recent-series of Xilinx FPGAs, such as Virtex-5, provide two Internal Configuration Access Ports, which cannot be accessed simultaneously. To prevent conflicts, the access to these ports as well as the hardware resource management needs to be controlled, e.g. by a special-purpose operating system running on an embedded processor. For that purpose and to handle the relations between temporally and spatially scheduled operations, the novel approach of an operating system is of high importance. This special purpose operating system, called CAP-OS (Configuration Access Port-Operating System), which will be presented in this paper, supports the clients using the configuration port with the services of priority-based access scheduling, hardware task mapping and resource management.

1. Introduction

Scheduling of tasks within a given time frame and with respect to a required deadline due to real-time aspects is well known in computer science from operating systems (OSes), especially in real-time operating systems (RTOSes). Scheduling strategies of conventional OSes vary between pre-emptive and non-pre-emptive scheduling. They can be further classified into static and dynamic scheduling, where static scheduling occurs at design time and dynamic scheduling at runtime. Therefore, dynamic scheduling is more suitable for runtime adaptive systems. Well-known dynamic scheduling algorithms are earliest deadline first (EDF) or rate monotonic algorithm (RMA) (see [1] for detailed descriptions). The classical scheduling and task mapping process of software-based systems with a traditional OS has its counterpart in novel runtime reconfigurable hardware systems. Within these systems, tasks can be presented additionally to the traditional software representation, as physical hardware realization, for example, on an FPGA. That means that an additional degree of freedom for task mapping on hardware resources is available for the OS layer. For example, compared to a task in a traditional software-based system that was mapped and executed on a resource as a software thread, the hardware reconfigurable variant of such a system would also allow running this task as a hardware block realized with logic resources on an FPGA. This difference and the new degree of freedom in task representation require the consideration of a novel concept for hardware task scheduling and mapping. In order to handle this process, a detailed analysis of the consequences, for example, due to data dependencies, priority, and real-time aspects, has to be investigated and formalized into a feasible algorithm for an efficient, special-purpose OS. Furthermore, the underlying hardware resources, including the internal configuration access port (ICAP), have to be

characterized in terms of timing, determinism, behavior in termination cases, and so forth. Also, these results have to be accounted for in the special-purpose OS approach by a cost function. The described investigation and the results can be exploited efficiently in the runtime adaptive multiprocessor system-on-chip (RAMPSoC) approach as described in [2]. In this approach, several processors, coprocessors, and hardware accelerators are available for concurrent task realization on an FPGA. The approach presented in this paper allows scheduling tasks of a control dataflow graph (CDG) and mapping these tasks either in hardware or in software on a reconfigurable multicore system on the FPGA. The algorithm, therefore, considers data dependencies; physical constraints from the configuration interface and the reconfigurable resources; the capability of the parallel data processing hardware of the RAMPSoC approach.

The paper is organized as follows: related work is presented in Section 2. Section 3 describes briefly the RAMPSoC approach and its features. In Section 4, the concept and the features of CAP-OS (configuration access port-operating system) are described. Section 5 presents how CAP-OS is integrated into the RAMPSoC hardware architecture. The implemented system and first results are presented in Section 6. A case study with an image-processing application is shown in Section 7. Finally, the paper is closed by presenting the conclusions and an outlook in Section 8.

2. Related Work

Scheduling for a hardware reconfigurable architecture is used in approaches reported in various publications. The selected publications discussed in this paper are only a subset of the numerous approaches developed in academic and industrial environment. However, the selected papers reflect the significant aspects in respect to the presented approach and allow an objective comparison of the benefits achieved in the proposed solution of the special-purpose OS named CAP-OS.

Garcia et al. [3] give an overview of the requirements for runtime- and operating systems for reconfigurable hardware-based systems. The authors especially point out the fact that physical constraints, such as the availability of hardware resources (in terms of area) and the configuration time, limited by the bandwidth of the configuration memory and interface, have to be taken into account for the scheduling. Especially, these are the challenges which have to be taken into account when reconfigurable hardware aware operating systems are introduced or developed.

Dittmann and Frank [4] describe a scheduling approach for a single processor and several accelerators, which can be configured at runtime. The solution provides a pre-emptive reconfiguration, which is important if a task with a higher priority has to substitute the configuration process of a task with lower priority. The scheduling strategy is based on a deadline monotonic (DM) algorithm with some extensions related to the fact that a hardware/software reconfigurable system is targeted. The approach has some restrictions due to the fact that only homogeneously shaped reconfigurable areas are supported. Because of this, only a fixed time frame for reconfiguration of the hardware is considered in the algorithms. In real systems, especially, when different hardware IPs have to be reconfigured, this time can vary significantly. A further restriction is that data dependencies between the tasks are not considered within the scheduling algorithm. The CAP-OS approach incorporates this into the metrics for the scheduling in order to achieve a beneficial scheduling of the hardware tasks. Furthermore, the approach requires drivers supporting the physical reconfiguration of the FPGA. This certainly could be a standard ICAP driver with the related IP cores.

Ullmann et al. [5] also target a single-processor solution with reconfigurable accelerators in a homogeneous shape and size, similar to the previously described approach. The scheduling is priority based and non-pre-emptive due to the fact that this approach was developed for automotive applications where pre-emption of a certain task is not allowed. The reported runtime system in the paper includes the hardware drivers for the configuration access port. The runtime system included some features, such as context load and save, which allows the resumption of tasks in hardware or software, or even a migration of the tasks from hardware to software or vice versa. The restrictions of this approach are mainly in the high overhead if a different application scenario needs to be realized. A time-consuming and hand crafted adaptation of the runtime system needs to be done. Furthermore, the fact that this approach was developed for the automotive domain limits the reuse in other application domains, such as image processing, where a more flexible scheduling is required.

ReconOS [6] uses an eCos (embedded configurable operating system) real-time operating system as basis for the scheduling approach. Also, here a single processor with loosely coupled reconfigurable accelerators is the target hardware architecture. In comparison to the previously described approach, the authors use a fixed priority scheduling approach. For synchronization purposes, a communication method for the software and hardware threads over the eCOS RTOS was developed. An interesting result is that a task graph with dependent and independent tasks is used as the input description for the scheduler. However, the limitation of using a single processor to perform the applications differs the ReconOS approach from CAP-OS. In CAP-OS, a variety of processors and accelerators can be handled.

In [7], the authors describe a concept for a quality of service- (QoS-) based operating system for multimedia applications on hand-held computers (e.g., PDA) with a reconfigurable accelerator. The paper includes also the strategy to abstract from the hardware layer in order to hide the complexity of the heterogeneous architecture consisting of a network-on-chip (NoC) and the processing elements from the developer. In relation to the RAMPSoC approach, the solution does not include a heterogeneous and adaptive communication infrastructure and processors and does not include the design flow for generating the required sources. The paper describes a definitely pioneering work in the area of operating systems for reconfigurable

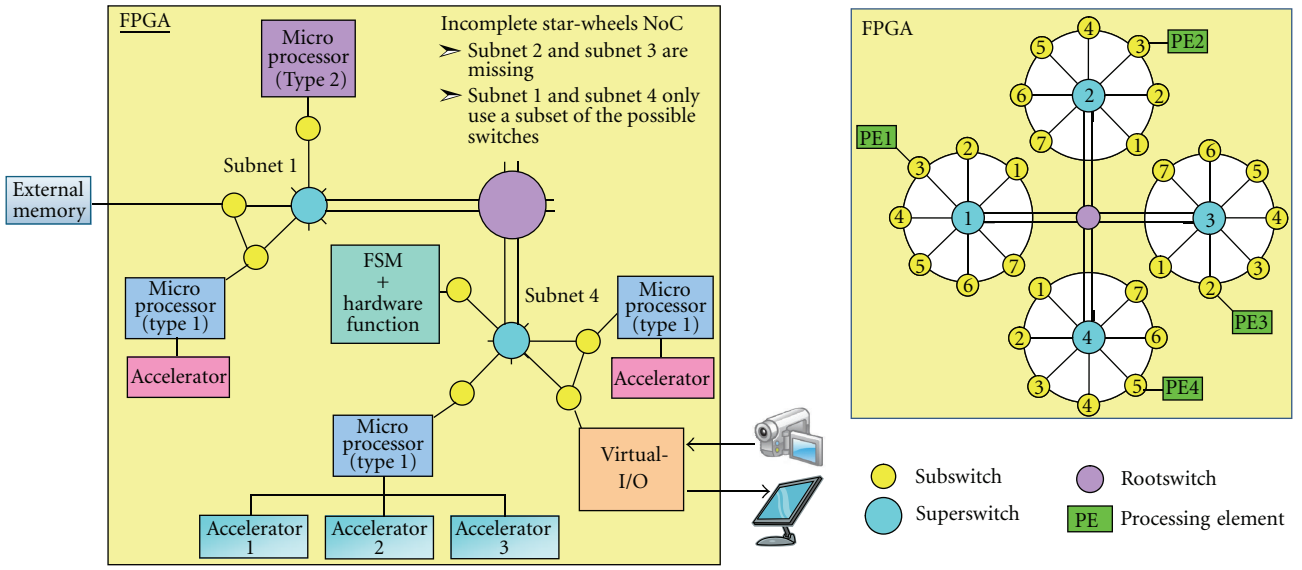


Figure 1: Example of an RAMPSoC architecture connected over an incomplete star-wheels network-on-chip. For comparison, on the top right side, an example of a complete star-wheels NoC is illustrated.

hardware and can be seen as the root for the evolution in this domain.

Based on the reported approaches, it is obvious that a novel OS approach for a runtime reconfigurable multiprocessor systems, such as RAMPSoC, has to be developed and introduced. One simple example for this necessity is the fact that the reconfigurable regions are no longer homogeneous in their footprint and, therefore, the configuration times vary between the different tasks, which may have to be allocated to the hardware. This and other parameters have to be handled with the novel approach of the CAP-OS.

3. The RAMPSoC Approach

The CAP-OS is used for runtime scheduling, task mapping, and resource management on a runtime reconfigurable multiprocessor system, such as RAMPSoC [2]. Figure 1 shows an example for a RAMPSoC architecture at one point in time. RAMPSoC is a heterogeneous multiprocessor system-on-chip (MPSoC) with distributed memory. It consists of a number of different processors connected over a communication infrastructure, which is a heterogeneous network-on-chip (NoC) called star-wheels NoC [8] in this example. Between others, the advantages of the star-wheels NoC are that it supports runtime adaptation and that it does not need to be implemented completely. Therefore, if additional switches are demanded, they can be added at runtime. Furthermore, different clock domains are supported, which is important, to achieve a good performance per watt ratio for multiprocessor systems. Additionally, a high throughput and therefore a low latency are supported, which is important for, for example, image-processing applications. Depending on the application requirements and the number of needed processors, also other communication infrastructures, such as point-to-point connections, buses, or other NoCs are supported and can be selected from a library at design time.

Each processor can be extended with one or several hardware accelerators to increase their performance for special-purpose instructions. Also, a finite-state machine (FSM) together with a hardware function can be used instead of a processor. The FSM is required to support the communication protocol over the NoC for communicating with the other processing elements.

Different processors can be chosen from a library (e.g., Xilinx MicroBlaze [9], Leon Sparc [10], etc.), and also a small library for image-processing hardware accelerators exists.

Dynamic and partial reconfiguration [11] is used to adapt the RAMPSoC hardware architecture at runtime. The software executables for the processors can be loaded at runtime either also by exploiting dynamic and partial reconfiguration (similar to the approach described by Sander et al. [12]) or by transferring the software executables via the communication infrastructure (e.g., the NoC). Furthermore, the clock frequency of the different processing elements can also be adapted at runtime. Like for the software executables, also here two possibilities exist: either by reconfiguring the appropriate digital clock manager (DCM) [13] or by switching to a different clock domain. In summary, the following runtime adaptations are supported by RAMPSoC:

(i) number and characteristics of processors,

(ii) communication infrastructure (e.g., size, bandwidth, and topology),

(iii) number and functionality of hardware accelerators,

(iv) software executables of the processors,

(v) clock frequency of the processing elements and network domains.

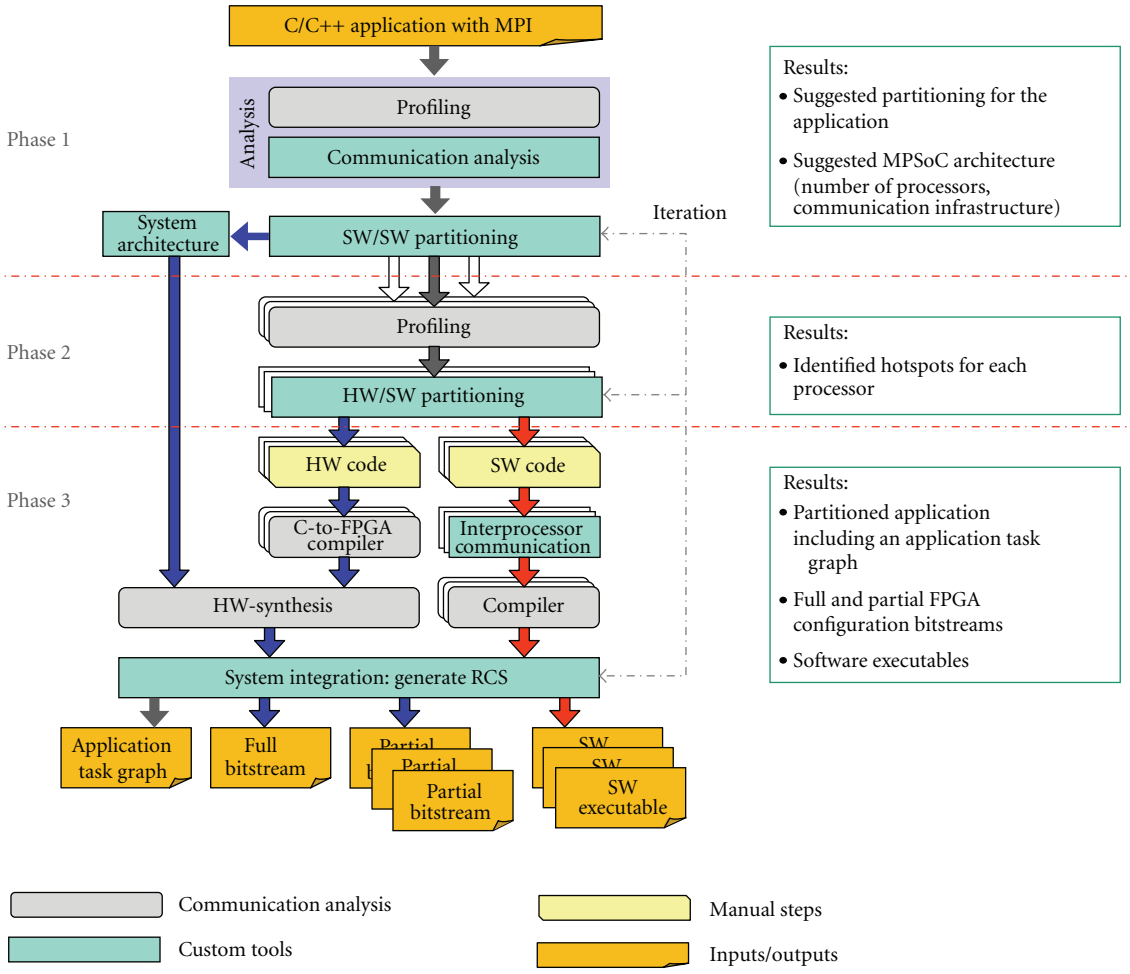


Figure 2: Design methodology of RAMPSoC.

A well balance between performance, power consumption, and area requirements can be achieved through runtime adaptation of the hardware architecture in respect to the requirements of the applications. More details about the hardware architecture of the RAMPSoC and its benefits can be found in [2].

For an efficient programming of such a flexible hardware architecture, an easy-to-use design methodology is required, which guides the user in application partitioning and in generating the appropriate hardware architecture at design time. As a result of the different analysis and partitioning steps, a task graph is generated for each partitioned application. The design suite also generates the partial bitstreams for the several hardware modules (e.g., processors, accelerators) as well as the software executables for the different processors. Figure 2 shows an overview of the current status of the design flow, which can be used for normal C/C++ applications or C/C++ applications using the message passing interface (MPI) [14]. MPI is a standard parallel programming model, which is used for supercomputing applications and especially to program multiprocessor systems with distributed memory. As the processors have only local memory, they exchange information by sending messages using the MPI standard protocol. RAMPSoC has its own MPI implementation layer, which translates the MPI standard protocol commands into the appropriate communication protocol required by the star-wheels NoC. The support for further possible communication infrastructures is currently under development as well as the improvement of the design methodology. A more detailed description of the functionality of the different tools within the design methodology can be found in [15].

The partial bitstreams, the software executables, and the task graphs of the applications are required by the CAP-OS, which will be presented in detail in the next section. The CAP-OS is responsible for the runtime scheduling of the configurations of the different tasks, allocating the tasks to the processing elements and for resource management. Furthermore, the CAP-OS needs to respond to runtime demands of the application, such as one or several processors requesting additional or different accelerators.

4. Concept of the CAP-OS

For an adaptive MPSoC, such as RAMPSoC, a flexible RTOS is required, which schedules the reconfiguration of the tasks and their runtime allocation to a specific processing

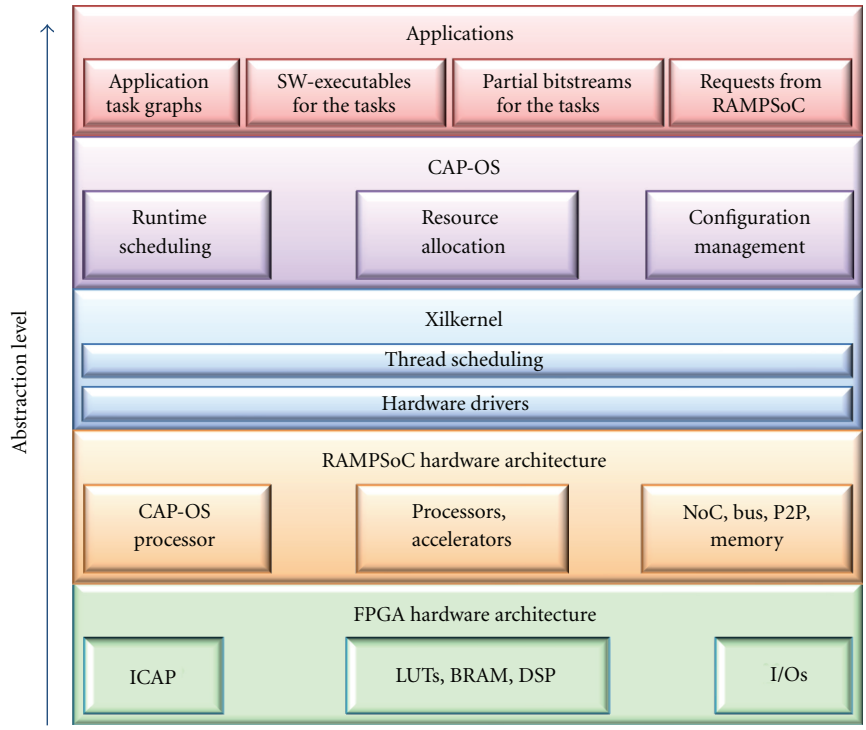


Figure 3: CAP-OS embedded in the several abstraction layers of the system approach.

element. Furthermore, this RTOS has to assure that the different applications meet their real-time requirements and that the utilization of the hardware resources and therefore the power consumption is kept low. Figure 3 shows how the CAP-OS manages the underlying RAMPSoC hardware architecture to fulfill the real-time requirements of the user applications. The CAP-OS further hides the complexity of the underlying dynamic RAMPSoC architecture from the user.

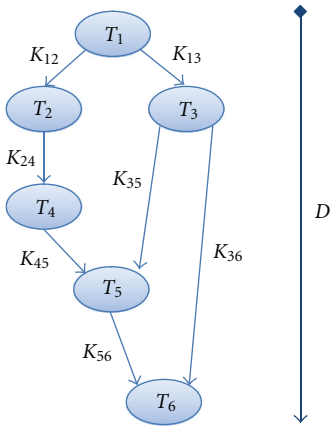
Physical resource allocation at runtime is done by performing partial and dynamic reconfiguration using the ICAP. Software task can be loaded either using the ICAP or using the interprocessor communication architecture. Therefore, the scheduling algorithm has to consider the time required for reconfiguring/loading a module, which depends on the data throughput of the ICAP interface or the communication infrastructure and certainly on the size of the module. This time frame is not negligible since the data amount for hardware modules can be very small, but also several hundred kilobytes. The software modules are normally smaller than 250 Kbytes, due to the restricted on-chip memory, while the hardware modules can be bigger. For each task, two different implementation options exist. A task can either be executed in software on a processor or in hardware as a hardware accelerator. The hardware task is normally a codesign, where parts of the task are executed in software on the processor and the compute intensive part is executed in hardware with the closely coupled accelerator. The choice to implement a task in software or in hardware depends normally on different parameters such as varying allocated hardware area, performance, and reconfiguration/loading time. The scheduling algorithm has to choose the appropriate type of realization to fulfill the real-time constraints. Moreover, the presented scheduling approach tries to reuse existing resources, which were already configured onto the chip in a previous point of time, with the goal to reduce the overall reconfiguration overhead. Furthermore, the scheduling algorithm has to support preemptive reconfiguration, because while reconfiguring one task, it can happen that a request for the reconfiguration of another task with higher priority occurs. As only one ICAP is available, the reconfiguration of the previous task has to be terminated and the new task needs to be reconfigured. After this procedure, the reconfiguration of the interrupted task has to restart, because a continuation of the terminated reconfiguration is not supported by the FPGA vendor. In contrary to this, the loading of an SW task via the communication infrastructure can be preempted and resumed. Here, a restart as for the ICAP interface is not required. The communication infrastructure allows the loading of more than one software task simultaneously, but then the bandwidth will be divided between the tasks. Therefore, it is more beneficially to load one task after the other based on the priorities of the tasks.

The scheduling approach presented in this paper can handle both independent and dependent tasks. A group of interrelated tasks is called a task graph (TG). Each TG must fulfill the following requirements:

(i) the TG is a directed acyclic graph (DAG),

(ii) each task runs on processors/hardware accelerators,

(iii) each task has an identity (ID)



T_x: task x
D: global deadline
K_{xy}: communication costs between task x and task y

Figure 4: Example task graph with global deadline, interrelation, and communication costs.

(iv) each task has the following information:

(1) neighborhood relation (predecessor/successor),
(2) algorithm type or hardware constraints (Algo-ID),
(3) execution time, reconfiguration/loading time,
(4) communication costs,
(5) name of the corresponding partial bitfile or software executable,

(v) the TG has a global deadline (D),

(vi) the TG has either hard or soft real-time constraints, which are inherited by the tasks belonging to the TG.

For the configuration of a task, the following three rules apply:

(i) it can be terminated;

(ii) it can be interrupted (only for SW tasks loaded over the communication infrastructure);

(iii) it is only feasible, after all predecessor tasks are completely reconfigured/loaded.

Figure 4 illustrates an example of such a TG including the global deadline, the interrelation, and the communication costs.

Within the CAP-OS, each task within a TG has a life cycle as shown in Figure 5.

Table 1 describes each of the states, which are traversed by a task during its life cycle, in detail.

Important here is that if the configuration of a task is interrupted, the task returns into the *Ready* state, the configuration data is lost and has to start all over again. The loading of a software task via the communication infrastructure on the other hand can be resumed if it has been interrupted. The address of the last word, which has been transferred via the communication infrastructure to the target processor, is stored, so that CAP-OS can later resume loading the software executable. Exceptions are

(i) the interrupted software will be remapped to a different processor;

(ii) the software had been interrupted by another software task, which will be loaded on the same processor. After the task has finished executing, the interrupted software will be loaded again on this processor.

In both cases, the loading of the interrupted software task has to restart from the beginning.

For simplicity, in the following section, the terms *configuration* and *configuration time* will be used for both configuration via ICAP and loading via the communication infrastructure. Loading via the communication infrastructure will only be used when explicitly requested.

As already mentioned in the previous section, the multiprocessor model used for the scheduling is a heterogeneous runtime adaptive MPSoC that uses a message passing communication scheme. The runtime scheduling algorithm is only performed for tasks, which are in state *Ready*. The novel runtime scheduling approach is described in detail in the next subsection.

4.1. The Novel Runtime Scheduling Approach. The novel runtime scheduling algorithm is divided into two main steps. First, a static scheduling algorithm is used to roughly assign priorities to the tasks of each TG using the information given by the TG description. The TG description has been received together with the bitstreams and software executables from the RAMPSoC design methodology. The TG description is written using the XML standard format and includes the following information: list of all tasks and detailed information for each task. For each task, these files contain the ID, the algo type, the successor tasks, the communication costs, the name of the bitstream file or/and software executable, the reconfiguration/loading time, and the execution time. Furthermore, for each task graph, the global deadline is given. Finally, this file also includes a list of possible processors, their configurations (e.g., pipeline length, memory size, specialized instructions, etc.), and the name of the corresponding bitfile. CAP-OS parses this XML file and updates its internal tables for the tasks and the processors. Also it stores the bitstreams and software executables, which have been received by the user, for example, via a Compact Flash card or an Ethernet connection, in the external memory.

For the priority-based static scheduling, the list scheduling algorithm is used, because it respects resource constraints. The available resources are the single ICAP, the communication infrastructure, and the maximum number of possible processors, which depends on the size of the chosen FPGA. First conservative estimates for the ASAP (as soon as possible) and the ALAP (as late as possible)

Table 1: Description of the life cycle states of a task.

Configuration and execution	Description
Not_ready	This task is not ready for reconfiguration, because its predecessors are not completely reconfigured.
Ready	This task is ready for reconfiguration and competes with the other *Ready* task for the access to the ICAP. Only tasks without predecessors, or whose predecessors have already been reconfigured, can enter this state.
Config	The task is under configuration/loading via the ICAP/the communication infrastructure onto the RAMPSoC. If a task with higher priority becomes *Ready*, the reconfiguration/loading process is terminated/interrupted, and the task returns into the *Ready* state and waits for a new possibility to access the ICAP/communication infrastructure.
Exec	After successful configuration/loading, the task starts execution and enters this state. An execution cannot be interrupted.
Exit	After the execution, the task enters this state. The allocated processing element is now free for the next task.

Figure 5: Life cycle states of a task.

start times for each task of a TG, consisting of m tasks, are calculated using the following formulas:

$$\begin{aligned} \mathrm{ASAP}(T_x) &= \sum_{T \in \mathrm{pre}(T_x)} (t_{\mathrm{con}}(T) + t_{\mathrm{exe}}(T)), \\ \mathrm{pre}(T_x)&\text{: Predecessor of task } T_x, \\ t_{\mathrm{con}}(T)&\text{: Configuration time of task ,} \\ t_{\mathrm{exe}}(T)&\text{: Execution time of task } T, \end{aligned} \tag{1}$$

$$\begin{aligned} \mathrm{ALAP}(T_x) &= D - \sum_{T \in \mathrm{succ}(T_x)} (t_{\mathrm{con}}(T) + t_{\mathrm{exe}}(T)), \\ \mathrm{succ}(T_x)&\text{: Successor of task } T_x, \\ D&\text{: Global deadline of the task graph,} \end{aligned} \tag{2}$$

$$\begin{aligned} \mu(T_x) &= \mathrm{ALAP}(T_x) - \mathrm{ASAP}(T_x), \\ \mu(T_x)&\text{: Mobility of task } T_x. \end{aligned} \tag{3}$$

The task loading via the communication infrastructure is the default procedure for software task to keep the ICAP interface available for the hardware task, because they do not have an alternative data path. Only when the communication infrastructure is blocked with a high-priority task, and another high-priority software task needs to be loaded as well, then the ICAP interface will be used to load the second software task. At this moment, it is not known, if an already configured processor can be reused for the software task. Therefore, t_{con} is the time required to reconfigure a new processor together with the software required for this task. This results in the worst case ALAP and ASAP start time.

Based on the ASAP and ALAP start time of each task, a priority can be assigned to each task in the TG using the urgency or the mobility of each task. The urgency depends on the maximum number of successors of a task. The mobility of a task (see Formula (3)) is the difference between its ALAP and ASAP start time and favors the tasks along the critical path. The TG in Figure 6 has, for example, the following critical path: $T1 \rightarrow T2 \rightarrow T4 \rightarrow T5 \rightarrow T6$. Because of this, the mobility is used here to assign the priorities to the tasks. The smaller the mobility, the higher is the priority of the task.

At runtime, only the *Ready* tasks are scheduled for configuration according to their priorities, which have been calculated with the list scheduling algorithm. Figure 6 shows such a TG which is processed by the CAP-OS for the purpose of scheduling the configuration of the different tasks. In the current time step shown in Figure 6, $T1$ has already been configured, and therefore $T2$ and $T3$ are now in the *Ready* state. Normally, the task with the highest priority will be configured first. If there are two or more *Ready* tasks and the difference between the mobility of the two tasks with the highest priority is smaller than the configuration time of the task with the lower priority (see Formula (4)), a dynamic cost function $K(T_x, T_y)$ (Formula (5)) is used to reassign the priorities of these two tasks.

$K(T_x, T_y)$ considers the ratio between the mobility of the two tasks $K_1(T_x, T_y)$ (Formula (6)) and the ratio between the number of successors of the two tasks $K_2(T_x, T_y)$ (Formula (7)). $K(T_x, T_y)$ is computed using Formulas (5) to (7), and it is only computed for the current two tasks with the highest priority to be scheduled. K_1 gets a greater weight in the cost function compared to K_2, because for real-time applications the execution time is the most important factor. Therefore, the default values were set to 0.6 for ω_1 and 0.4 for ω_2. These weights can be modified by the user depending on the requirements of the application. Additionally, multiple TGs can be scheduled at runtime. If some of these TGs have hard real-time and others only soft real-time requirements, then all tasks of the TGs with the soft real-time constraints will be delayed. They will be configured after the tasks with the hard real-time constraints, even though they might have a higher priority according to the list scheduling algorithm. This is

important, to assure that the hard real-time TGs meet their constraints.

T_x gets highest priority if

$$\mu(T_y) - \mu(T_x) > CT(T_y), \quad \mu(T_x) < \mu(T_y),$$
$$\mathrm{CT}(\mathrm{T_y}): \text{Configuration time of task } y. \tag{4}$$

Else decision is made using $K(T_x, T_y)$:

$$\begin{cases} K(T_x, T_y) < 0, & T_y \text{ gets highest priority,} \\ K(T_x, T_y) \geq 0, & T_x \text{ gets highest priority,} \end{cases}$$
$$K(T_x, T_y) = \omega_1 * \left(K_1(T_x, T_y) - K_1(T_y, T_x)\right) + \omega_2 * \left(K_2(T_x, T_y) - K_2(T_y, T_x)\right), \tag{5}$$
$$\omega_1, \omega_2: \text{ Weighting factors,}$$

$$K_1(T_x, T_y) = \begin{cases} \dfrac{\mu(T_y)}{\mu(T_x)}, & \mu(T_x) < \mu(T_y) \wedge \mu(T_x) \neq 0, \\ 0, & \text{else} \end{cases}$$
$$\mu(T_x): \text{ Mobility of task } x, \tag{6}$$

$$K_2(T_x, T_y) = \begin{cases} \dfrac{N(T_x)}{N(T_y)}, & N(T_x) > N(T_y) \wedge N(T_y) \neq 0, \\ 0, & \text{else} \end{cases}$$
$$N(T_x): \text{ Number of successors of task } x. \tag{7}$$

Finally, an additional feature is supported by CAP-OS. This feature allows for increasing the clock frequency of a processing element at runtime. This can be done by either reconfiguring the corresponding digital clock manager (DCM) [16] or by using clock multiplexers to switch to a different clock frequency. Both approaches have been described in [17]. The reconfiguration of a DCM takes more time than the use of clock multiplexers. However, the DCM reconfiguration provides a greater variety of possible clock frequencies than the clock multiplexers, because only a limited number of clock multiplexers are available on the FPGA. Both approaches are supported here. Therefore, in the following, DCM reconfiguration stands for both approaches. The user has to select at design time the approach, which is more appropriate for the target application.

DCM reconfiguration is used to speed up the execution time of a task. Hereby, it is assumed that the execution time stays in strong relation to the clock frequency. This DCM reconfiguration is used if a task cannot complete within its ALAP time or if another task urgently requires the same processor.

Therefore, the single steps of the CAP-OS scheduling algorithm can be summarized as follows:

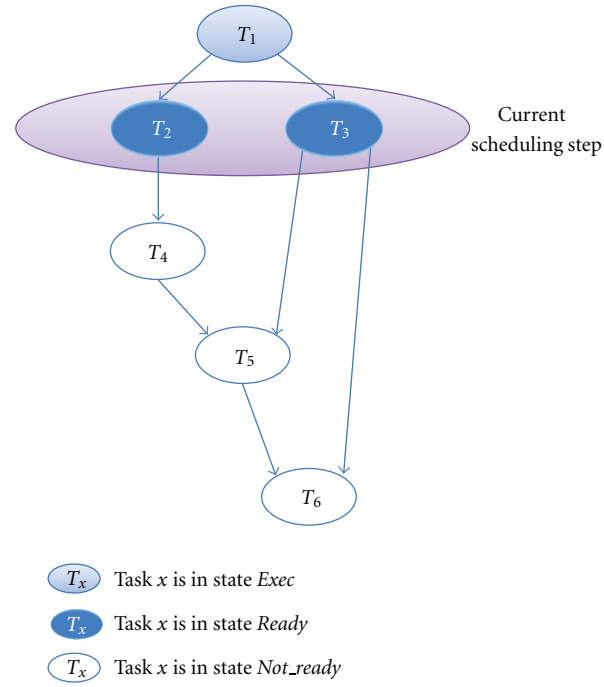


Figure 6: Task graph to illustrate the functionality of the scheduling.

(1) calculate ASAP and ALAP start time for each task in the task graph,

(2) calculate the mobility of each task and schedule their priorities using a list scheduling algorithm,

(3) select the *Ready* tasks, and schedule them dynamically:

   (a) delay tasks with soft real-time constraints,

   (b) reassign priorities using the cost function if necessary,

   (c) reconfigure the DCM, if necessary,

   (d) terminate the current configuration if a task with a higher priority occurs.

This results in a pre-emptive scheduling approach, which allows the termination of a configuration. Furthermore, it uses a combination of static list scheduling and a novel dynamic scheduling approach. It considers resource constraints, such as a single ICAP, the communication infrastructure, or the maximal number of possible processors. Furthermore, it is used to schedule both hardware and software tasks. Moreover, the clock frequency of processing elements can be increased at runtime, if necessary, and the configuration times as well as the communication costs between tasks are considered. Another degree of freedom is that, while a hardware task is loaded via the ICAP interface, a software task can be loaded simultaneously via the on-chip communication infrastructure.

4.2. Resource Allocation of the CAP-OS. After the scheduling, the CAP-OS tries to allocate a resource for the *Ready* task

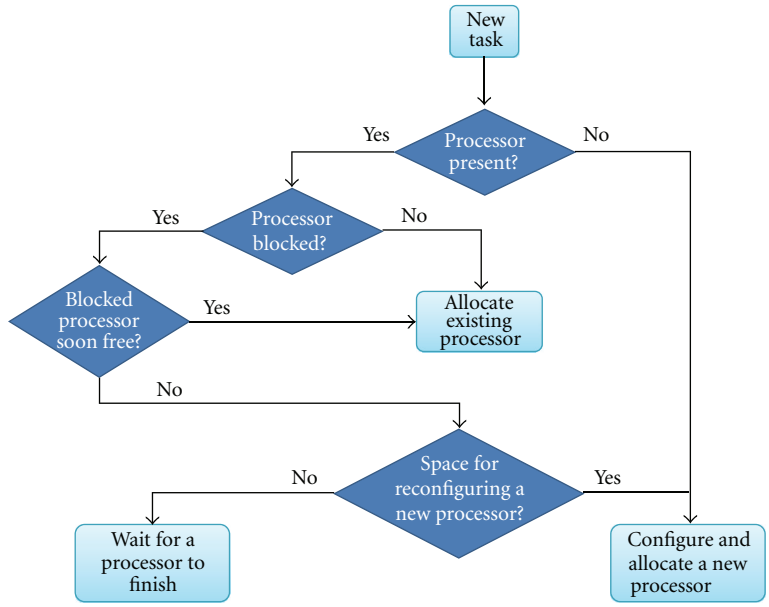


Figure 7: Decision tree for resource allocation for an SW task.

with the highest priority. For the resource allocation, the decision is made as shown in Figure 7.

First CAP-OS analyzes if a processor is present and available on the reconfigurable hardware. If no processor is present, a new one is configured and reserved for the new task. If processors are present in the system, it searches for one, which is not blocked by another task. If all existing processors are blocked, it is checked if one of them will finish its execution soon. This is important, because the reconfiguration and allocation takes time. If an existing processor finishes in a shorter amount of time than the reconfiguration time of a new processor, the reuse of this existing processor is preferred. This also has the benefit to reduce the area utilization and therefore to reduce the overall power consumption. If none of the existing processors will finish soon, it is analyzed if the maximal number of processors is reached or if there is still space to reconfigure a new processor. If there is space on the reconfigurable hardware, a new processor is reconfigured and allocated for the new task. If not, the new task has to wait until one of the processors becomes available.

The same procedure is done for a codesign task, because this task is a combination of an SW task and a hardware accelerator. Here, the available processor is extended with an accelerator, while the software part of the codesign task is loaded into the chosen processor.

Pure hardware tasks also exist. These are requests from existing RAMPSoC processors, which either need an accelerator to increase performance or which want to exchange the existing accelerator against a different one due to requests from the environment. Such exchange requests could be, for example, the exchange of an image-processing filter due to a change in the incoming frame of the image. In that case, no decision tree is required, and the requesting processor is extended by configuring the requested hardware accelerator via the ICAP interface.

4.3. Configuration Management. After the *Ready* task with the highest priority has been successfully assigned to a processor, this task is assigned to the configuration management. The configuration management is responsible for handling the configuration of the tasks via the ICAP and also for loading software tasks into already existing processors over the communication infrastructure. It is also responsible for pre-empting a current configuration if another task with a higher priority needs to be configured. As mentioned before, a terminated configuration has to restart again from the beginning, because Xilinx FPGAs do not support the continuation of a terminated configuration so far. On the other side, software tasks, under configuration via the communication infrastructure, can be interrupted if a task with higher priority occurs. Afterwards, the terminated configuration can be resumed. Furthermore, it is possible to configure a hardware and a software task or two software tasks simultaneously by using the configuration via the communication infrastructure for one of the software tasks, while the other tasks are configured via the ICAP interface. Therefore, the configuration management of the CAP-OS distinguishes between three types of configurations as shown in Table 2.

The term *soft* and *medium* means an interruptible and *hard* means a noninterruptible configuration. Soft configurations are new software tasks that get loaded via the communication infrastructure into an existing processor. As they can be pre-empted and continued easily, they can be interrupted any time if a task with high priority occurs. Medium configuration types are, for example, the configuration over the ICAP interface of software tasks or hardware accelerators for existing processors. As soon as 80% of the corresponding bitstream of a medium configuration type is configured, this element changes to be a hard configuration type. The reason is to prevent the termination of a nearly finished configuration, because the already configured data would

Table 2: Configuration types.

Configuration type	Features	Elements
Soft (communication infrastructure)	Interruptible	Software
Medium (ICAP)	Interruptible until 80% of the bitstream are reconfigured	Software, accelerator
Hard (ICAP)	Not interruptible	Processor, DCM

Table 3: Performance of the currently supported different interfaces in RAMPSoC.

Interface	Performance (100 MHz, Virtex-4)
FSL-ICAP [18]	28,28 MB/s
Point-to-point via FSLs (Fast Simplex Links) [19]	13,09 MB/s

be lost. 80% is a default parameter and can be changed by the user, depending on the application requirements. Other examples of hard configuration types are the configuration of the DCMs and of the processors, because the configuration of a DCM is urgent and fast, and the processor is far less task specific than an accelerator.

The decision to configure a software task via the ICAP interface or via the communication infrastructure depends on the mobility of the task, the availability of the interfaces, and the loading speed of the interface, which depends on the target platform and the chosen interface. Table 3 gives an overview about possible interfaces and how is their performance.

It is possible to increase the throughput of these interfaces by connecting them directly to the external memory instead of using the processor to load the data from external memory. An example is the PLB-ICAP from Claus et al. [20], which achieves a throughput of 400 MB/s. It is therefore planned to provide a direct memory access also to the FSL-ICAP to further increase its performance. The performance of communication infrastructure can be increased in a similar fashion.

4.4. Communication Establishment between Tasks. After successfully configuring a task, the CAP-OS tries to establish a communication with this task and to transfer information about the IDs of the communication partners to it. Figure 8 illustrates the required steps, to successfully establish a communication between the different tasks at runtime.

The five runtime communication establishment steps required after a task has been mapped onto a processor x are

(1) CAP-OS sends sync word to processor x;

(2) processor x responds with the same sync word to ensure a correct communication;

(3) CAP-OS sends task info (Task ID, number of predecessor/successor tasks, and their IDs) to processor x. This task info is required by the task to find its communication partners at runtime;

(4) processor x sends its Task ID to all other processors, and it checks each of its communication links for the Task ID of its communication partners. It has to send its Task ID to all other processors, because it could happen that a predecessor and a successor will be mapped onto the same processor. An example for such a case will be given in Section 6;

(5) after execution, processor x informs CAP-OS that it is now free for a new task.

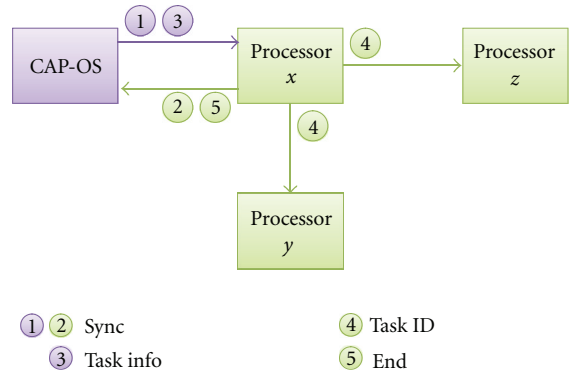


Figure 8: Runtime communication establishment steps between different tasks.

5. Integration of CAP-OS on RAMPSoC

CAP-OS software is integrated into an RAMPSoC on one of the available microprocessors. On the selected microprocessor, a state-of-the-art RTOS with multithreading capabilities is implemented. On top of this RTOS, the CAP-OS is implemented using different threads for the different functionalities. As shown in Figure 9, this microprocessor is directly connected with the Xilinx ICAP primitive via an FSL connection. Furthermore, the processor has access to an external memory, in which the partial bitstreams of the tasks are stored.

The microprocessor is connected with the other processors in this example over the star-wheels NoC. A point-to-point connection with each of the other partners or a connection over a different NoC or a bus is also supported. Several possible choices for an on-chip microprocessor exist. The IBM PowerPC 405 (PPC405) [21] was chosen for running CAP-OS. It is available on Xilinx Virtex-4FX FPGAs as a hard IP core. The main reasons for choosing the PPC405 are the support of high frequencies up to 450 MHz and the availability on the Virtex-4FX100 FPGA on the used target FPGA board from Alpha-Data [22]. High frequencies are important to execute the CAP-OS with a low latency to support and enable the real-time requirements. Other

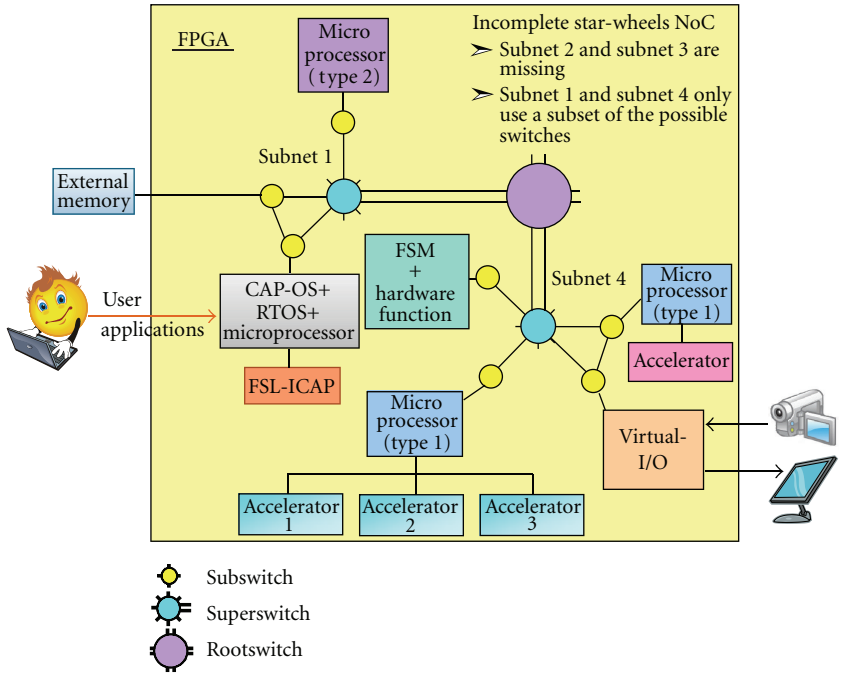


Figure 9: Integration of the CAP-OS on the RAMPSoC.

possible microprocessors would be soft IP cores, such as Xilinx MicroBlaze or Leon SPARC, but they lack with the support of such high frequencies. As the new Xilinx FPGAs, such as Virtex-6, does not provide the PowerPC anymore, an alternative version of CAP-OS for the MicroBlaze processor was also realized and implemented.

After selecting the processor, an appropriate RTOS was chosen. The demands for the RTOS are

(i) proven support of PPC405 and MicroBlaze,

(ii) multithreading capabilities,

(iii) small memory footprint.

Several different RTOSes exist, but due to the reasons above, the Xilkernel [23] from Xilinx was selected. The CAP-OS is programmed in C, and its functionalities are implemented in several different threads, which are executed in Xilkernel using multithreading. For scheduling the different threads, Xilkernel offers two scheduling policies: round robin or priority-based scheduling. Priority-based scheduling was chosen to execute the different CAP-OS threads according to their priorities.

Furthermore, the processor is directly connected to the ICAP primitive and to an external memory (DDR2 SDRAM), in which the bitstreams are stored. The CAP-OS and Xilkernel are executed using on-chip memory for maximum performance. In the following subsection, the implementation of the different CAP-OS threads is described in detail.

5.1. Implementation of the CAP-OS. The CAP-OS is programmed using six threads as shown in Table 4.

The priorities are sorted with increasing numbers starting with the highest priority from 0. Test_main is the startup thread and has a fixed priority. The priorities of the other five threads can change at runtime depending on the demands of the applications. The three threads with priority level 3 (Schedule, Configure, and Contr_Exit_Task) compete against each other, after the first three threads with higher priority have finished executing. While the other threads only execute in the beginning once, these three concurring threads execute until the last task finishes executing.

6. Implementation and First Results

The functionality of the CAP-OS as specified was evaluated by implementing an RAMPSoC system on the target Alpha-Data FPGA board. The CAP-OS was implemented using one of the available PPC405s and the Xilkernel RTOS. The maximum number of reconfigurable processors was set to four, to be artificially below the number of tasks within our evaluation task graphs. As the target Virtex-4FX 100 FPGA provides a large number of reconfigurable resources, a higher number of processors could be used, if necessary. As reconfigurable processor, the Xilinx MicroBlaze (μBlaze) [9] was chosen due to its small area footprint and the compatibility to the PPC405. As shown in Figure 10, the Fast Simplex Links (FSLs) [19] are utilized for the communication between the processors. The decision for these communication infrastructures has its basis in the fact that FSLs offer an FIFO-based unidirectional communication and that for the limited number of processors, an NoC would create a high overhead in terms of utilized area. The PPC405 can be connected via FSL to 32 data sinks and sources, while each μBlaze could be connected to 16.

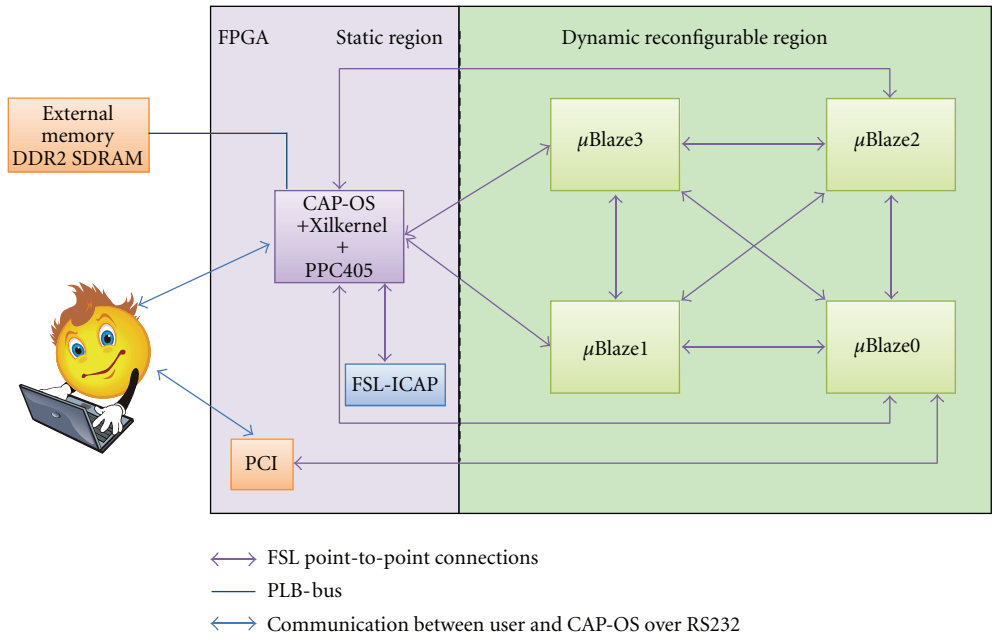


Figure 10: Implemented RAMPSoC system.

Table 4: Realized threads of the CAP-OS.

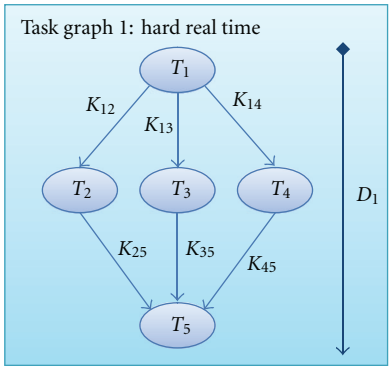
Thread	Priority	Description
Test_main	0	Initial thread. Launches the other five threads.
Init_proc	1	Generates a list containing all possible processors and their attributes. Executes only once.
Task_graph	2	Initialization of the tasks and generation of the task graphs. Calculation of ALAP and ASAP start time and the mobility of each task. Matching of tasks with equal requirements (HW constraints, same algorithm)
Schedule	3	Scheduling of the *Ready* tasks and processor allocation.
Configure	3	Configuration management for the scheduled and allocated task and communication establishment between the new configured task and its neighbors.
Contr_Exit_Task	3	Controls the executing tasks. If a task finishes execution, the occupied processing element is freed.

Additionally, the FSL-ICAP IP core from Xilinx together with an external DDR2 SDRAM is connected to the PPC405. The user interface to the CAP-OS is realized through an RS232 port. For the preliminary tests, the dynamic and partial reconfiguration was not deployed, because the scope was to verify the CAP-OS and not the FSL-ICAP primitive. Instead of sending the partial bitstreams to the ICAP core, a counter within the *Configure* thread was used, to simulate artificially the reconfiguration times of the different tasks. For reconfiguring a whole processor via the ICAP interface 5 ms, and for loading a software task onto an existing processor via the FSL communication infrastructure, 2 ms were assumed. These times are-worst case scenarios, and certainly the reconfiguration time and the loading time vary in real scenarios depending on the size of the bitstream/the software executable. Table 5 shows the estimated reconfiguration times for an average size MicroBlaze and for the size of a typical image processing application software.

At system startup, it is assumed that only the static part is present and the other processors will be "reconfigured" on demand. Physically, the system shown in Figure 10 was present from the beginning, and after the artificial simulation of the reconfiguration time is finished, the corresponding processor is activated. For the verification of the CAP-OS functionality and to measure the timing overhead of the current CAP-OS implementation, TG1 shown in Figure 11 was used. TG1 has hard real-time constraints. This could be for example, an image-processing application, which receives the images from a camera and has to provide the results of an image-processing algorithm to the user via a monitor in real time. Therefore, the global deadline (D_1) of TG1 is 40 ms

Table 5: Reconfiguration/loading times for the chosen interfaces: FSL-ICAP and FSL point-to-point links.

Type	Size	Reconfiguration/loading time
Reconfiguring a MicroBlaze	ca. 120 KB	4,24 ms
Loading a software	16 KB	1,22 ms



Task	Description	Algo-ID
T_1, T_5	Software tasks with same hardware requirements: for example, processor with PCI connection	0
T_2, T_3, T_4	Software tasks, for example, different image processing filter	1, 2, 3

Figure 11: Task graph used for the CAP-OS evaluation: $D_1 = 40$ ms.

Table 6: Timing overhead of CAP-OS for processing TG1.

Thread	Average number of clock cycles per call
Init_proc	2118
Task_graph	9022
Schedule	650
Contr_Exit_Task	227

using a camera with a frame rate of 25 Hz. If this deadline is missed, frames will be lost.

To measure the timing overhead, the CAP-OS was executed on the FPGA using TG1. To proof, if the CAP-OS correctly reuses existing resources, the two tasks $T1$ and $T5$ were set to have the same algorithm (same Algo-ID) as shown in Figure 11. During the execution on the FPGA, the number of clock cycles, required per call by each thread, were measured. The results for the timing overhead provided by the CAP-OS are shown in Table 6.

The clock cycles of the *Configure* thread depend on the size of the bitstream and on the speed of the ICAP primitive. Therefore, they are not explicitly presented here. *Test_main* only launches the other five threads, but itself does not produce timing overhead and is therefore also not mentioned here. Of course, *Init_proc* depends on the number of processors (here four), and *Task_graph* depend on the TG (here TG1 with five tasks). Therefore, these numbers are just an example for the given TG. The clock cycles required for the *Schedule* thread depends on the complexity of the scheduling. For example, they increase slightly if the cost function needs to be evaluated for two tasks. *Contr_Exit_Task* is very stable.

With this example, it can be shown that CAP-OS worked correctly as specified and assigned the tasks of TG1 without violating the global deadline. Also, the resource reuse worked correctly. $T5$ was allocated onto the same processor as $T1$, because they have the same algorithm, and this way the reconfiguration time could be saved.

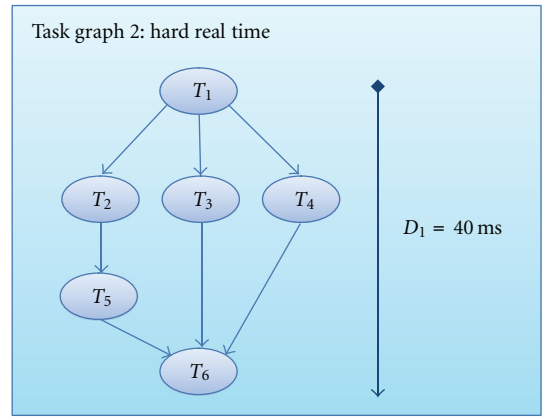


Figure 12: Task graph TG2 of an image-processing application, which detects template-based and point-like objects within an image.

7. Case Study with Image-Processing Scenario

Finally, a case study using an image-processing application for object recognition was designed. Figure 12 illustrates the task graph of the application. Task 1 is a pure software task and receives an input image via PCI bus. It then forwards the complete image to task 2. After partitioning the image into two overlapping tiles, it forwards the upper half of the image to task 3 and the lower half to task 4. Task 2 equalizes the histogram of the image. As this is a very compute intensive task, parts of the algorithm are outsourced in an accelerator. Therefore, task 2 requires both a processor for the software part of the algorithm and a closely coupled accelerator for the compute intensive part of the algorithm. Task 2 sends its results to task 5. Task 5 tries to find objects within the equalized image by comparing a predefined template with the input image using the SAD (sum of absolute differences) algorithm. It is implemented in software and forwards its results to task 6. Task 3 and task 4 execute both the hotspot detector algorithm in software on different parts of the image. The hotspot detector algorithm is an image-processing algorithm, which searches inside an image for bright point-like objects. The results of task 3 and task 4 are then forwarded as well to task 6, which is responsible for collecting all results and forwarding them to the Host PC via the PCI connection. Like task 1, task 6 is also implemented as a software task, and both require a processor with a PCI connection.

To measure the execution times of each task separately, each task was implemented on a single MicroBlaze processor running at 100 MHz on the target FPGA platform. For the measurement, an input image with the size of 64×64 pixels was used. An exception is task 2. The execution time of this task was measured using a single MicroBlaze connected via FSL with the hardware accelerator. The size of the software executable file for each algorithm is received using the GCC compiler within the Xilinx Platform Studio [24]. This size is important for calculating the loading time via the FSL communication infrastructure into the local memory of a processor on the FPGA.

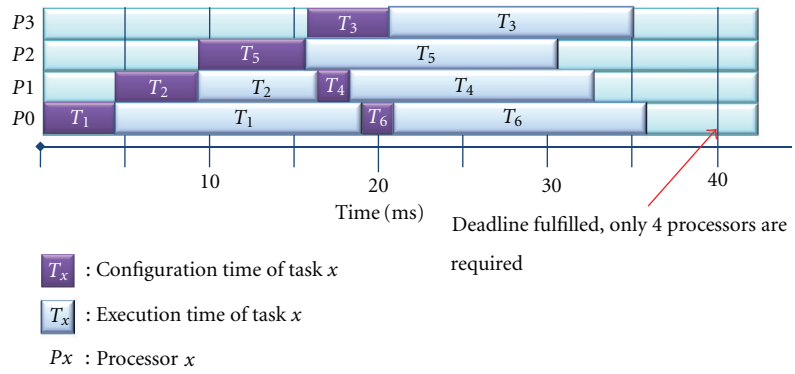


Figure 13: Theoretical results of CAP-OS for the image-processing applications and respecting heterogeneous configuration times.

For each design, the place and route report was used to extract the values for the utilized area, this means the required number of CLBs, block RAM (BRAM), and DSPs for the MicroBlaze processor. For task 2, the amount of resources for the accelerator was also taken into account.

The smallest addressable segment of a Virtex-4 FPGA configuration memory space is called a frame and covers a height of 16 CLBs or 4 BRAMs or 8 DSPs. It has a length of 1312 bits [25]. 22 frames are needed to reconfigure a column of 16 CLBs. To reconfigure 4 BRAMs, 64 frames are needed for the BRAM content, and 20 frames for the BRAM interconnect. To reconfigure 8 DSPs, also 21 frames are needed. Therefore, the number of frames required for the partial bitstream is calculated as shown in the following Formula:

$$\#\text{Frames} > \left\lceil \frac{\#\text{CLBs}}{16} \right\rceil * 22 + \left\lceil \frac{\#\text{DSPs}}{8} \right\rceil * 21 + \left\lceil \frac{\#\text{BRAMs}}{4} \right\rceil * (64 + 20). \tag{8}$$

The reconfiguration time is then calculated by using this Formula:

$$\text{Reconfiguration Time} = \frac{\#\text{Frames} * \text{Framelength}}{8 * \text{Throughput FSL\_ICAP}}. \tag{9}$$

In Table 7, a detailed description for each task of TG2 together with the measured execution times and the estimated reconfiguration times are presented. The reconfiguration times are worst-case reconfiguration times. This means that for each task the reconfiguration of a new processor has been assumed. These reconfiguration times are the input for the static list scheduling algorithm of the CAP-OS described in Section 4. The static list scheduling algorithm is used to assign each task to a priority, based on the mobility of the task. The mobility of a task is the subtraction of its ASAP start time from its ALAP start time as shown in Formula (3). Table 7 shows the resulting priorities for TG2.

Furthermore, the last column in Table 7 shows the Algo-ID of each task. The Algo-ID is used by the dynamic scheduling decision of CAP-OS to decide which task may reuse an existing processor. $T3$ and $T4$ have the same Algo ID, because they execute the same software algorithm on a different data set. $T1$ and $T6$, which both execute a different algorithm, have the same Algo-ID, because they have a common hardware restriction, as they require both a processor with a connection to the PCI interface.

Figure 13 shows the calculated results of CAP-OS for the TG2. As can be seen, four processors $P0$ to $P3$ were used. Figure 10 shows how the final implemented system would look like, where P0 is the only processor with a PCI connection. Due to this, that processor is reused by task 6 after task 1 has finished. Task 3 and task 4 are mapped onto different processors even though they have the same Algo-ID, because otherwise the global deadline would be violated. Also, the benefit of using the FSL communication infrastructure for loading software into an existing processors and therefore keeping the ICAP interface available for the hardware reconfiguration can be seen. For example, processor P3 is being reconfigured via the ICAP interface while simultaneously the software executable of task 4 is loaded over the FSL communication infrastructure to processor P1.

Table 8 shows the estimated execution times for mapping the application sequentially on one processor in the first row, parallelizing it using for each task a processor in the second row and the solution proposed by CAP-OS, which uses four processors in the last row. Furthermore, Table 8 shows the part of the execution time, which is spent for reconfiguring the hardware and for loading the software.

As can be seen, both the uni-processor design and the 6-processor designs are static and do not require any reconfiguration time.

The uni-processor design is the slowest solution and violates the global deadline of the application, because it can only execute the complete application sequentially.

The 6-processor design is the fastest solution, but also has the highest resource requirements and therefore the highest power consumption. Furthermore, some of the processors are idle (e.g., processor with task 6), while others are executing (e.g., task 3, 4, and 5). This results in a bad workload balancing.

Table 7: Detailed information about each task of the TG 2 at 100 MHz.

Task	Description	Rec. time in ms	Exe. time in ms	Priority	Algo ID
$T1$	SW Task: Read from PCI	4,24	1	1	1
$T2$	SW Task + Accelerator: Histogram equalization	4,5	2,25	2	2
$T3$, $T4$	SW Task: Hotspot detector	4,24	14,5	4,5	3
$T5$	SW Task: SAD (sum of absolute differences)	4,24	17,25	3	4
$T6$	SW Task: Send to PCI	4,24	1	6	1

Table 8: Estimated execution times for the TG2.

System	Estimated execution time in ms	Time spent for HW reconfiguration/SW loading time
UniProcessor	50,5	0
6 Processors	17,25	0
CAP-OS (4 Processors)	31,72	17,22/2,44

CAP-OS with its four processors provides a meet-in-the middle solution by reusing existing processors. It therefore achieves a good balance between performance and power consumption. It is faster than the uni-processor design and fulfils the global deadline of the application. Moreover, it requires fewer processors than the 6-processor solution and therefore has lower power consumption. Also, as can be seen in Figure 13, the workload between the processors is well balanced. It can be further seen that as soon as a processor finishes its current task, it is allocated for executing the next task of the application.

It has to be mentioned here that the execution times of the different tasks, for example, $T1$, are longer in this figure than the ones given in Table 7. The reason for this is the consideration of the communication between the tasks. For example, task 1 has to wait until all its successors are reconfigured, before it can finish its execution.

8. Conclusions and Outlook

In this paper, the concept and the features of a special-purpose OS called CAP-OS were presented. The CAP-OS is responsible for the scheduling, the resource allocation, and reconfiguration and for managing the access to the configuration access port. The CAP-OS has been integrated into the RAMPSoC approach to handle the runtime organization for the adaptive RAMPSoC hardware architecture. The CAP-OS was implemented using six threads on the Xilkernel RTOS running on a PPC405 and on a MicroBlaze processor. The correct functionality and the timing overheads of the CAP-OS were measured on the FPGA using an example task graph. The benefits of the CAP-OS were shown using a case study with a task graph from an image-processing application and comparing the results against both a uni-processor design and a complete parallel design.

Future work will be the extension of the CAP-OS to support the reconfiguration of the communication infrastructure. Furthermore, it will be extended to handle not only the demands of the user, but also the reconfiguration demands of the other processors within the RAMPSoC. These demands are mainly the reconfiguration of the accelerators if at runtime, for example, a different accelerator is required depending on the currently processed data. Furthermore, the CAP-OS will be further evaluated and will be also tested using real dynamic and partial reconfiguration. The implementation of a partial reconfigurable design of the presented case study is currently under development to evaluate that the calculated results are equal to the statically measured ones. Additional extensions of CAP-OS will be the support of merging several bitstreams and supporting bitstream relocation. Bitstream relocation is important to reduce the amount of required external memory for storing each bitstream for each possible location.

References

[1] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, *Scheduling Computer and Manufacturing Processes*, Springer, Berlin, Germany, 2001.

[2] D. Göhringer and J. Becker, "High performance reconfigurable multi-processor-based computing on FPGAs," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, pp. 1–4, Atlanta, Ga, USA, April 2010.

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

[4] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '07)*, pp. 123–128, April 2007.

[5] M. Ullmann, M. Hübner, B. Grimm, and J. Becker, "On-demand FPGA run-time system for dynamical reconfiguration with adaptive priorities," in *Proceedings of the International Conference on Field Programmable Logic and Application (FPL '04)*, pp. 454–463, August 2004.

[6] E. Lübbers and M. Platzner, "ReconOS: an RTOS supporting hard- and software threads," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 441–446, August 2007.

[7] J.-Y. Mignolet, S. Vernalde, D. Verkest, and R. Lauwereins, "Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances," in *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture*, 2002.

[8] D. Göhringer, B. Liu, M. Hübner, and J. Becker, "Star-wheels network-on-chip featuring a self-adaptive mixed topology and a synergy of a circuit- and a packet-switching communication

protocol," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '09)*, Prague, Czech Republic, September 2009.

[9] "MicroBlaze Processor Reference Guide, Embedded Development Kit, EDK 9.2i," UG081 (v8.1)., http://www.xilinx.com/.

[10] "Leon Sparc," http://www.gaisler.com/.

[11] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Application (FPL '06)*, pp. 1–6, Madrid, Spain, August, 2006.

[12] O. Sander, L. Braun, M. Hübner, and J. Becker, "Data reallocation by exploiting FPGA configuration mechanisms," in *Proceedings of the 4th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC '08)*, vol. 4943 of *Lecture Notes in Computer Science*, pp. 312–317, London, UK, March 2008.

[13] D. Göhringer, J. Obie, M. Hübner, and J. Becker, "Impact of task distribution, processor configurations and dynamic clock frequency scaling on the power consumption of FPGA-based multiprocessors," in *Proceedings of the 5th International Workshop on Reconfigurable Communication Centric Systems-on-Chip (ReCoSoC '10)*, KIT Scientific Publishing, Karlsruhe, Germany, 2010.

[14] "MPI: A Message-Passing Interface Standard, Version 2.2," Message Passing Interface Forum, September 2009, http://www.mpi-forum.org/.

[15] D. Göhringer, M. Hübner, M. Benz, and J. Becker, "A design methodology for application partitioning and architecture development of reconfigurable multiprocessor systems-on-chip," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, pp. 259–262, Charlotte, NC, USA, May 2010.

[16] "Virtex-4 FPGA Configuration User Guide," UG071 (v1.11), June 2009, http://www.xilinx.com/.

[17] D. Göhringer, J. Obie, A. Braga, M. Hübner, C. Llanos, and J. Becker, "Exploration of power-performance tradeoffs through parameterization of FPGA-based multiprocessor systems," in *Proceedings of the the 5th International Workshop on Reconfigurable Communication Centric Systems-on-Chip (ReCoSoC '10)*.

[18] M. Hübner, D. Göhringer, J. Noguera, and J. Becker, "Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs," in *Proceedings of the Reconfigurable Architectures Workshop (RAW '10)*, Atlanta, Ga, USA, April, 2010.

[19] "Fast Simplex Link (FSL) Bus (v2.11a)," DS449, June 2007, http://www.xilinx.com/.

[20] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hübner, and J. Becker, "A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, Heidelberg, Germany, September 2008.

[21] "PowerPC Processor Reference Guide," UG011 (v.1.2), January 2007, http://www.xilinx.com/.

[22] "Alpha Data," http://www.alpha-data.com/.

[23] "Xilkernel v3_00_a," EDK 9.1i, December 2006, http://www.xilinx.com/.

[24] "Embedded System Tools Reference Manual, Embedded Development Kit, EDK 9.2i," UG111 (v9.2i), September 2007, http://www.xilinx.com/.

[25] "Virtex-4 Configuration Guide," UG071 (v1.5), January 2007, http://www.xilinx.com/.