International Journal of Reconfigurable Computing

Selected Papers from the International Conference on Reconfigurable Computing and FPGAs (ReConFig'10)

Guest Editors: Claudia Feregrino, Miguel Arias, Kris Gaj, Viktor K. Prasanna, Marco D. Santambrogio, and Ron Sass

Selected Papers from the International Conference on Reconfigurable Computing and FPGAs (ReConFig'10)

International Journal of Reconfigurable Computing

Selected Papers from the International Conference on Reconfigurable Computing and FPGAs (ReConFig'10)

Guest Editors: Claudia Feregrino, Miguel Arias, Kris Gaj, Viktor K. Prasanna, Marco D. Santambrogio, and Ron Sass

Editorial Board

Contents

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 319827, 2 pages
doi:10.1155/2012/319827

Editorial

Selected Papers from the International Conference on Reconfigurable Computing and FPGAs (ReConFig'10)

Claudia Feregrino,[1] Miguel Arias,[1] Kris Gaj,[2] Viktor K. Prasanna,[3] Marco D. Santambrogio,[4] and Ron Sass[5]

[1] *Instituto Nacional de Astrofísica, Óptica y Electrónica, 72840 Puebla, PUE, Mexico*
[2] *George Mason University, Fairfax, VA 22030, USA*
[3] *University of Southern California, Los Angeles, CA 90033, USA*
[4] *Politecnico di Milano, 20133 Milano, Italy*
[5] *The University of North Carolina at Charlotte, Charlotte, NC 28223, USA*

Correspondence should be addressed to Claudia Feregrino, cferegrino@inaoep.mx

Received 23 August 2012; Accepted 23 August 2012

The sixth edition of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'10) was held in Cancun, Mexico, from November 30 to December 2, 2010. This special issue covers actual and future trends on reconfigurable computing and FPGA technology given by academic and industrial specialists from all over the world. All articles in this special issue are extended versions of selected papers presented at ReConFig'10, for final publication they were peer reviewed to ensure that they are presented with the breadth and depth expected from this high quality journal.

There are a total of 16 articles in this issue. The following 8 papers correspond to the track titled general sessions. In "*Evaluation of Runtime Task Mapping Using the rSesame Framework*", K. Sigdel et al. present the rSesame framework to perform a thorough evaluation (at design time and at runtime) of various task mapping heuristics from the state of the art. The experimental results suggest that such an extensive evaluation can provide a useful insight both into the characteristics of the reconfigurable architecture and on the efficiency of the task mapping. In "*Exploration of Uninitialized Configuration Memory Space for Intrinsic Identification of Xilinx Virtex-5 FPGA Devices*", O. Sander et al. demonstrate an approach to utilize unused parts of configuration memory space for FPGA device identification. Based on a total of over 200,000 measurements on nine Xilinx Virtex-5 FPGAs, it is shown that the retrieved values have promising properties with respect to consistency on one device, variety between different devices, and stability considering temperature variation and aging. In "*Placing Multimode Streaming Applications on Dynamically Partially Reconfigurable Architectures*", S. Wildermann et al. discuss the architectural issues to design reconfigurable systems where parts of the hardware can be dynamically exchanged at runtime in order to allow streaming applications running in different modes of the systems to share resources. The authors propose a novel algorithm to aggregate several streaming applications into a single representation, called merge graph, in addition, they propose an algorithm to place streaming application at runtime which not only considers the placement and communication constraints, but also allows to place merge tasks. In "*On the Feasibility and Limitations of Just-in-Time Instruction Set Extension for FPGA-Based Reconfigurable Processors*", M. Grad and C. Plessl study the feasibility of moving the customization process to runtime and evaluate the relation of the expected speedups and the associated overheads. The authors present a tool flow that is tailored to the requirements of this just-in-time ASIP specialization scenario. The methods are evaluated by targeting a previously introduced Woolcano reconfigurable ASIP architecture for a set of applications from the SPEC2006, SPEC2000, MiBench, and SciMark2 benchmark suites. In "*A Fault Injection Analysis of Linux Operating on an FPGA-Embedded Platform*" J. S. Monson et al. present an FPGA-based Linux test bed for the purpose of measuring its sensitivity to single-event upsets. The test bed consists of two ML410 Xilinx development boards

connected using a 124-pin custom connector board. The Design Under Test (DUT) consists of the "hard core" PowerPC, running the Linux OS, and several peripherals implemented in "soft" (programmable) logic. In "*NCOR: An FPGA-Friendly Nonblocking Data Cache for Soft Processors with Runahead Execution*" K. Aasaraai and A. Moshovos propose an FPGA-friendly nonblocking cache that exploits the key properties of runahead execution. In "*A Dynamically Reconfigured Multi-FPGA Network Platform for High-Speed Malware Collection*", S. Mühlbach and A. Kock refine the base NetStage architecture for better management and scalability. By using dynamic partial reconfiguration it is possible to update the functionality of the honeypot during operation. The authors describe the technical aspects of these modifications and show results evaluating an implementation on a current quad-FPGA reconfigurable computing platform. In "*Exploring Many-Core Design Templates for FPGAs and ASICs*", I. Lebedev et al. present a highly productive approach to hardware design based on a many-core microarchitectural template used to implement compute-bound applications expressed in a high-level data-parallel language such as OpenCL. The template is customized on a per-application basis via a range of high-level parameters such as the interconnect topology or processing element architecture.

Two papers are within the area of security and cryptography. In "*Implementation of Ring-Oscillators-Based Physical Unclonable Functions with Independent Bits in the Response*", F. Bernard et al. analyze and propose some enhancements of Ring-Oscillators-based Physical Unclonable Functions (PUFs). PUFs are used to extract a unique signature of an integrated circuit in order to authenticate a device and/or to generate a key. The authors show that designers of RO PUFs implemented in FPGAs need a precise control of placement and routing and an appropriate selection of ROs pairs to get independent bits in the PUF response. In "*Blind Cartography for Side Channel Attacks: Cross-Correlation Cartography*", L. Sauvage et al. present a localisation method based on cross-correlation, which issues a list of areas of interest within the attacked device. It realizes an exhaustive analysis, since it may localise any module of the device and not only those which perform cryptographic operations. The method is experimentally validated using observations of the electromagnetic near field distribution over a Xilinx Virtex 5 FPGA.

Two papers are within the area of high performance reconfigurable computing. Nonuniform random numbers are key for many technical applications, and designing efficient hardware implementations of nonuniform random number generators is a very active research field. However, most state-of-the-art architectures are either tailored to specific distributions or use up a lot of hardware resources. At ReConFig'10, we have presented a new design that saves up to 48% of area compared to state-of-the-art inversion-based implementation, usable for arbitrary distributions and precision. In "*A Hardware Efficient Random Number Generator for Nonuniform Distributions with Arbitrary Precision*", C. de Schryver et al. introduce a more flexible version of a non-uniform random number generators presented at ReConFig'10.the authors introduce a refined segmentation scheme that allows to reduce the approximation error significantly.

Two papers are within the area of multiprocessor systems and networks on chip. In "*Redsharc: A Programming Model and On-Chip Network for Multi-Core Systems on a Programmable Chip*", W. V. Kritikos et al. document the API, describe the common infrastructure, and quantify the performance of a complete implementation of the reconfigurable data-stream hardware-software architecture (Redsharc). The authors also report the overhead, in terms of resource utilization, along with the ability to integrate hard and soft processor cores with purely hardware kernels being demonstrated. In "*Combining SDM-Based Circuit Switching with Packet Switching in a Router for On-Chip Networks*", A. Kuti Lusala and J. D. Legat present a hybrid router architecture for Networks-on-Chip "NoC". The architecture combines Spatial Division Multiplexing- "SDM-" based circuit switching and packet switching in order to efficiently and separately handle both streaming and best-effort traffic generated in real-time applications. Combining these two techniques allows mitigating the poor resource usage inherent to circuit switching.

Finally, two more papers are within the area of reconfiguration techniques. In "*Dynamic Circuit Specialisation for Key-Based Encryption Algorithms and DNA Alignment*", T. Davidson et al. explain the core principles behind the dynamic circuit specialization technique. Parameterised reconfiguration is a method for dynamic circuit specialization on FPGAs. The main advantage of this new concept is the high resource efficiency. Additionally, there is an automated tool flow, TMAP, that converts a hardware design into a more resource-efficient runtime reconfigurable design without a large design effort. In "*Using Partial Reconfiguration and Message Passing to Enable FPGA-Based Generic Computing Platforms*", M. Saldaña et al. introduce a new partition-based Xilinx PR flow to incorporate PR within the previously proposed MPI-based message-passing framework to allow hardware designers to create template bitstreams, which are predesigned, prerouted, and generic bitstreams that can be reused for multiple applications.

Acknowledgments

It is our pleasure to express our sincere gratitude to all who contributed in any way to produce this special issue. We would like to thank all the reviewers for their valuable time and effort in the review process and to provide constructive feedbacks to the authors. We thank all the authors who contributed to this Special Issue for submitting their manuscript and sharing their latest research results. We hope that you will find in this Special Issue a valuable source of information to your future research.

Claudia Feregrino
Miguel Arias
Kris Gaj
Viktor K. Prasanna
Marco D. Santambrogio
Ron Sass

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 234230, 17 pages
doi:10.1155/2012/234230

Research Article

Evaluation of Runtime Task Mapping Using the rSesame Framework

Kamana Sigdel,[1] Carlo Galuzzi,[1] Koen Bertels,[1] Mark Thompson,[2] and Andy D. Pimentel[2]

[1] *Computer Engineering Group, Technical University of Delft, Mekelweg 4, 2628 CD, Delft, The Netherlands*
[2] *Computer Systems Architecture Group, University of Amsterdam, Science Park 904, 1098 XH, Amsterdam, The Netherlands*

Correspondence should be addressed to Kamana Sigdel, k.sigdel@tudelft.nl

Received 8 May 2011; Revised 20 December 2011; Accepted 30 December 2011

Academic Editor: Viktor K. Prasanna

Performing runtime evaluation together with design time exploration enables a system to be more efficient in terms of various design constraints, such as performance, chip area, and power consumption. rSesame is a generic modeling and simulation framework, which can explore and evaluate reconfigurable systems at both design time and runtime. In this paper, we use the rSesame framework to perform a thorough evaluation (at design time and at runtime) of various task mapping heuristics from the state of the art. An extended Motion-JPEG (MJPEG) application is mapped, using the different heuristics, on a reconfigurable architecture, where different Field Programmable Gate Array (FPGA) resources and various nonfunctional design parameters, such as the execution time, the number of reconfigurations, the area usage, reusability efficiency, and other parameters, are taken into consideration. The experimental results suggest that such an extensive evaluation can provide a useful insight both into the characteristics of the reconfigurable architecture and on the efficiency of the task mapping.

1. Introduction

In recent years, reconfigurable architectures [1, 2] have received an increasing attention due to their adaptability and short time to market. Reconfigurable architectures use reconfigurable hardware, such as Field Programmable Gate Array (FPGA) [3, 4] or other programmable hardware (e.g., Complex Programmable Logic Device (CPLD) [5], reconfigurable Datapath Array (rDPA) [6]). These hardware resources are frequently coupled with a core processor, typically a General Purpose Processor (GPP), which is responsible for controlling the reconfigurable hardware. Part of the application's tasks is executed on the GPP, while the rest of the tasks are executed on the hardware. In general, the hardware implementation of an application is more efficient in terms of performance than a software implementation. As a result, reconfigurable architectures enhance the whole application through an implementation of selected application kernels onto the reconfigurable hardware, while preserving the flexibility of the software execution with the GPP at the same time [7, 8]. The design of such architectures is subject to numerous design constraints and requirements, such as performance, chip area, power consumption, and memory. As a consequence, the design of heterogeneous reconfigurable systems imposes several challenges to system designers such as hardware-software partitioning, Design Space Exploration (DSE), task mapping, and task scheduling.

Reconfigurable systems can evolve under diverse conditions due to the changes imposed either by the architecture, by the applications, or by the environment. A reconfigurable architecture can evolve under different conditions, for instance, processing elements shutdown in order to save power, or additional processing elements are added in order to meet the execution deadline. The application behavior can change, for example, due to the dynamic nature of the application-application load changes due to the arrival of sporadic tasks. In such systems, the design process becomes more sophisticated as all design decisions have to be optimized in terms of runtime behaviors and values. Due to changing runtime conditions with respect to, for example, user requirements or having multiple simultaneously executing applications competing for platform resources, design time evaluation alone is not enough for any kind of architectural exploration. Especially in the case of partially

dynamic reconfigurable architectures that are subject to changes at the runtime, design time exploration and task mapping are inadequate and cannot address the changing runtime conditions. Performing runtime evaluation enables a system to be more efficient in terms of various design constraints, such as performance, chip area, and power consumption. The evaluation carried at runtime can be more precise and can evaluate the system more accurately than at design time. Nevertheless, such evaluations are typically hard to obtain due to the enormous size and complexity of the search space generated by runtime parameters and values.

In order to benefit from both design time and runtime evaluations, we developed a modeling and simulation framework, called rSesame [9], which allows the exploration and the evaluation of reconfigurable systems at both design time and runtime. With the rSesame framework, designers can instantiate a model that can explore and evaluate any kind of reconfigurable architecture running any set of streaming applications from the multimedia domain. The instantiated model can be used to evaluate and compare various characteristics of reconfigurable architectures, hardware-software partitioning algorithms, and task mapping heuristics. In [10], we used the rSesame framework to perform runtime exploration of a reconfigurable architecture. In [11], we proposed a new task mapping heuristic for runtime task mapping onto reconfigurable architectures based on hardware configurations reuse. In this paper, we present an extension of the work presented in [10, 11]. In particular, we present an extensive evaluation and comparison of various task mapping heuristics from the state of the art (including the heuristics we presented in [11]) both at design time and at runtime using the rSesame framework. More specifically, the main contributions of this paper are the following:

(i) a detailed case study using the rSesame framework for mapping different runtime task mapping heuristics from the state of the art (including the runtime task mapping heuristics in [11]). For this case study, we use an extended MJPEG application and a reconfigurable architecture;

(ii) an extensive evaluation of the different heuristics for a given reconfigurable architecture. This evaluation is performed by considering different number of FPGA resources for the same reconfigurable architecture model;

(iii) a thorough comparison of the aforementioned heuristics under different resource conditions using various nonfunctional design parameters, such as execution time, number of reconfiguration, area usage, and reusability efficiency. The comparison is done both at design time as well as at runtime.

The rest of the paper is organized as follows. Section 2 provides the related research. Section 3 discusses the rSesame framework, which is used as a simulation platform for evaluating task mapping at runtime, while Section 4 presents a detailed case study using the different heuristics. In Section 5, a detailed analysis and evaluation of the task mapping at runtime using the rSesame framework is presented. Finally, Section 6 concludes the paper.

2. Related Work

Task mapping can be performed in two mutual nonexclusive ways: at design time and at runtime. The task mapping performed at the design time can generally be faster, but it may be less accurate as the runtime behavior of a system is mostly captured by using offline (static) estimations and predictions. Examples of techniques for task mapping at design time are dynamic programming [12], Integer Linear Programming (ILP) [13], simulated annealing [14, 15], tabu search [16], genetic algorithm [17, 18], and ant colony optimization [19].

In another way of performing task mapping, the reconfigurable system is evaluated for any changes in the runtime conditions and the task mapping is performed at runtime based on those conditions. Under such scenario, the changes in the system are considered and the task mapping is performed accordingly. In [20], the authors present a simple approach for runtime task mapping in which a mapping module evaluates the most frequently executed tasks at runtime and maps them onto a reconfigurable hardware component. However, this work [20] focuses on the lower level and it targets only loop kernels. A similar approach for high-level runtime task mapping is presented in [21] for multiprocessor System on Chip (SoC) containing fine-grain reconfigurable hardware tiles. This approach details a generic runtime resource assignment heuristic that performs fast and efficient task assignment. In [22], the authors define the dynamic coprocessor management problem for processors with an FPGA and provide a mapping to an online optimization based on the cumulative benefit heuristic, which is inspired by a commonly used accumulation approach in online algorithm work.

In the same way, the study in [23] presents runtime resource allocation and scheduling heuristic for the multithreaded environment, which is based on the status of the reconfigurable system. Correspondingly, [24] presents a dynamic method for runtime task mapping, task scheduling and task allocation for reconfigurable architectures. The proposed method consists of dynamically adapting an architecture to the processing requirement. Likewise, the authors in [25, 26] present an online resource management for heterogeneous multiprocessor SoC systems, and the authors in [27] present a runtime mapping of applications onto a heterogeneous reconfigurable tiled SoC architecture. The approach presented in [27] proposes an iterative hierarchical approach for runtime mapping of applications to a heterogeneous SoC. The approach presented in [28] consists of a mapper, which determines a mapping of application(s) to an architecture, using a library at runtime. The approach proposed by authors in [29] performs mapping of streaming applications, with real-time requirements, onto a reconfigurable MPSoC architecture. In the same way, Faruque et al. [30] present a scheme for runtime-agent-based distributed application mapping for on-chip communication for adaptive NoC-based heterogeneous multiprocessor systems.

There are few attempts which combine design time exploration together with runtime management and try to evaluate the system at both stages [21, 31]. However, these methodologies are mostly restricted to the MPSoC domain and do not address the reconfigurable system domain. Unlike existing approaches that are either focused on design time or on runtime task mapping, we are focused on exploring and evaluating reconfigurable architectures at *design time* as well as at *runtime* during early design stages.

3. rSesame Framework

The rSesame [9] framework is a generic modeling and simulation infrastructure, which can explore and evaluate reconfigurable systems at early design stages both at design time and at runtime. It is built upon the Sesame framework [32]. The rSesame framework can be efficiently employed to perform DSE of the reconfigurable systems with respect to hardware-software partitioning, task mapping, and scheduling [10]. With the rSesame framework, an application task can be modeled either as a hardware (HW), or as a software (SW), or as a *pageable* task. A HW (SW) task is always mapped onto the reconfigurable hardware component (microprocessor), while a *pageable* task can be mapped on either of these resources. Task assignment to the SW, HW, and pageable categories is performed at design time based on the design time exploration of the system. At runtime, these tasks are mapped onto their corresponding resources based on time, resources, and conditions of the system.

The rSesame framework uses the Kahn Process Network (KPN) [33] at the granularity of coarse-grain tasks for application modeling. Each KPN process contains functional application code instrumented with annotations that generate read, write, and execute events describing the actions of the process. The generated traces are forwarded onto the architecture layer using an intermediate mapping layer, which consists of Virtual Processors (VPs) to schedule these traces. Along with the VPs, the mapping layer contains a Runtime Mapping Manager (RMM) that deals with the runtime mapping of the applications on the architecture. Depending on current system conditions, the RMM decides where and when to forward these events. To support its decision making, the RMM employs an arbitrary set of user-defined policies for runtime mapping, which can simply be plugged in and out of the RMM. The RMM also collaborates with other architectural components to gather architectural information. The architecture layer in the framework models the architectural resources and constraints. These architectural components are constructed from generic building blocks provided as a library, which contains components for processors, memories, on-chip network components, and so forth. As a result, any kind of reconfigurable architecture can be constructed from these generic components. Beside the regular parameters, such as computation and communication delays, other architectural parameters like reconfiguration delay and area for the reconfigurable architecture can also be provided as extra information to these components.

The rSesame framework provides various useful design parameters to the designer. These include the total execution time (in terms of simulated cycles), area usage, number of reconfigurations, percentage of reconfiguration, percentage of HW/SW execution, and reusability efficiency. These design parameters are described in more detail in the following.

3.1. Execution Time. The execution time is recorded in terms of simulated clock cycles. The SW execution time is the total number of cycles when all the tasks are mapped only on the GPP. The HW execution time is recorded when the tasks are mapped onto the FPGA. The speedup is calculated as a ratio of these two values.

3.2. Percentage of HW and SW Execution Time. The percentage of HW (SW) execution is computed as the total percentage of the execution time contributed by the FPGA (GPP) for HW (SW) execution of an application. Similarly, the percentage of reconfiguration time represents the percentage of the total execution time spent in reconfigurations. This provides an indication on the total time spent in the computation and in the reconfiguration. These values are calculated as follows.
The percentage of SW execution time is given by

$$\text{SW Exec}(\%) = \frac{\sum_{i=1}^{N} \#\text{SWEx}(T_i) \cdot T_{\text{SW}(i)}}{\text{TotalExecTime}} \cdot 100, \quad (1)$$

where $\#\text{SWEx}(T_i)$ is the total number of SW executions counted by the model for task T_i, $T_{\text{SW}(i)}$ is the software execution latency for task T_i, and TotalExecTime is the total simulated execution time.
The percentage of HW execution time is given by

$$\text{HW Exec.}(\%) \leq \frac{\sum_{i=1}^{N} \#\text{HWEx}(T_i) \cdot T_{\text{HW}(i)}}{\text{TotalExecTime}} \cdot 100, \quad (2)$$

where $\#\text{HWEx}(T_i)$ is the total number of HW executions counted for task T_i by the model, $T_{\text{HW}(i)}$ is the hardware execution latency for task T_i, and TotalExecTime is the total execution cycles incurred while running an application onto the given reconfigurable architecture.

Note that, the HW execution percentage can only be given here as an upper bound, since the execution of tasks on the FPGA can be performed in parallel. The metric calculated here is an accumulated value. The simulator, however, can give the actual value. A similar equation holds for the time spent reconfiguring, which is given as a percentage of the total execution time as follows:

$$\text{Recon}(\%) \leq \frac{\sum_{i=1}^{N} \#\text{Recon}(T_i) \cdot T_{\text{Recon}(i)}}{\text{TotalExecTime}} \cdot 100, \quad (3)$$

where $\#\text{Recon}(T_i)$ is the number of times T_i is configured, $T_{\text{Recon}(i)}$ is the reconfiguration delay of T_i, and TotalExecTime represents the total execution cycles incurred while running an application onto the given reconfigurable architecture.

3.3. Number of Reconfigurations. The number of reconfigurations is recorded as the total number of reconfigurations incurred during the execution of an application onto the given architecture. This provides an indication on how efficiently the reconfiguration delay is avoided, while mapping tasks onto the FPGA. For example, the mapping of task A, task B, and then task A again on the FPGA requires 3 reconfigurations, while by changing this mapping sequence to task A, task A and then task B, only 2 reconfigurations are required.

3.4. Time-Weighted Area Usage. The weighted area usage factor is a metric that computes how much area is used throughout the entire execution of an application on a particular architecture. This provides an indication on how efficiently the FPGA area is utilized. This metric is calculated as follows:

$$\text{Area Usage}(\%) = \frac{\sum_{i=1}^{N} \text{Area}(T_i) \cdot T_{\text{HW}(i)} \cdot \#\text{HWEx}(T_i)}{\text{TotalExecTime} \cdot \text{Area}(\text{FPGA})} \cdot 100, \tag{4}$$

where $\text{Area}(T_i)$ is the area occupied by task T_i on the FPGA, $T_{\text{HW}(i)}$ is the hardware execution latency of T_i, $\#\text{HWEx}(T_i)$ is the total number of HW executions counted by the model for task T_i, Area(FPGA) is the total area available on the FPGA, and TotalExecTime is the total execution time of the application.

3.5. Reusability Efficiency. A task execution onto the FPGA has two phases: the *configuration phase*, where its configuration data that represents a task is loaded onto the FPGA, and the *running phase*, where the task is actually processing data. In an ideal case, a task can be configured onto the FPGA only once and it is executed in all other cases. Nonetheless, this is not always possible as the FPGA has limited area. The Reusability Efficiency (RE) is the ratio of the reconfiguration time that is saved due to the hardware configuration reuse to the total execution time of any task. The RE of a task can be defined as follows:

$$\text{RE}_{\text{task}} = \frac{(\#\text{HWEx} - \#\text{Recon}) \cdot T_{\text{Recon}}}{\#\text{HWEx} \cdot T_{\text{HW}} + \#\text{SWEx} \cdot T_{\text{SW}} + \#\text{Recon} \cdot T_{\text{Recon}}}, \tag{5}$$

where #HWEx, #SWEx, and #Recon are the number of HW executions, SW executions, and reconfigurations of a task, respectively. Similarly, T_{HW}, T_{SW}, and T_{Recon} are the corresponding hardware, software, and reconfigurable latencies.

The RE of a task indicates the percentage of the total time saved by a task when multiple reconfigurations are avoided or, in other words, a task configuration is reused. The numerator in (5) represents the time that is saved when a mapping of a task is reused, and the denominator represents the total execution time. The total RE for an application can be calculated as the summation of the numerator in (5) for all N tasks divided by the total execution time for the whole application as follows:

$$\text{RE}_{\text{App}} \leq \frac{\sum_{i=1}^{N} (\#\text{HWEx}(i) - \#\text{Recon}(i)) \cdot T_{\text{Recon}(i)}}{\text{TotalExecTime}}. \tag{6}$$

Note that the RE calculated in this way for the whole application can only be given here as an upper bound, since the execution of tasks on the reconfigurable hardware can be performed in parallel. A higher RE can obtain a higher speedup. To study this relation, we use the RE as an evaluation parameter to study the behavior of each task.

4. Case Study

We use the rSesame framework as a simulation platform for performing extensive evaluation of the various task mapping heuristics from the state of the art. In order to perform this case study, we constructed a Molen model using the rSesame framework for mapping an extended MJPEG application (see Section 4.2) onto the Molen reconfigurable architecture [34] (see Section 4.1). The Molen model is used to evaluate the different task mapping heuristics under consideration. We incorporated these heuristics as strategies for the Molen model to perform runtime task mapping of the extended MJPEG application onto the Molen architecture. We conducted an evaluation of these task mapping heuristics based on various system attributes recorded from the model.

The rSesame framework allows easy modification and adjustment of individual components in the model, while keeping other parts intact. As a result, the framework allows designers to experiment with different kinds of runtime task mapping heuristics. The considered heuristics have variable complexity in terms of their implementation and the nature of their execution. In the original context, they were used at different system stages, ranging from the lower architecture level to Operating System (OS), and the higher application levels. These heuristics are used as a strategy to perform runtime mapping decisions in the model. They are taken from literature, and have been adapted to fit in the framework. In the following, we discuss these heuristics in more detail.

4.1. As Much As Possible Heuristic (AMAP). AMAP tries to maximize the use of FPGA resources (such as area) as much as possible, and it performs task mapping based on resource availability. In this case, tasks are executed on the FPGA if the latter has enough resource to accommodate them; otherwise, they are executed on the GPP. This straightforward heuristic can be used as a simple resource management strategy in various domains.

Algorithm 1 presents the pseudocode that describes the functionality of the AMAP heuristic for performing runtime mapping of a task T_i. The heuristic chooses to execute task T_i onto the FPGA if there are sufficient resources (e.g., area in Algorithm 1) for T_i (line 3 to 6 in Algorithm 1). In all other conditions, tasks are executed on the GPP (line 7 to 9 in Algorithm 1).

```
(1) HW ← set of tasks mapped onto the FPGA
(2) SW ← set of tasks mapped onto the GPP
(3) if T_i.area ≤ area then
(4)    {T_i is mapped onto FPGA}
(5)    HW = HW ∪ T_i
(6)    area = area − T_i.area
(7) else
(8)    {Map T_i onto the GPP}
(9)    SW = SW ∪ T_i
(10) end if
```

Algorithm 1: Pseudocode for the As Much As Possible heuristic (AMAP) for mapping task T_i.

```
(1) HW ← set of tasks mapped onto the FPGA
(2) SW ← set of tasks mapped onto the GPP
(3) if T_i.area ≤ area then
(4)   if CB(T_i) > (T_SW(i) − T_HW(i)) then
(5)      {T_i is mapped onto the FPGA}
(6)      HW = HW ∪ T_i
(7)      area = area − T_i.area
(8)   end if
(9) else
(10)   {Not enough area, swap the mapped tasks.}
(11)   while area ≤ T_j.area and j ∈ HW do
(12)    if CB(T_i) - (T_SW(i) − T_HW(i)) > CB(Tj) then
(13)       area = area + T_j.area
(14)    end if
(15)   end while
(16)   if T_i.area ≤ area then
(17)    {T_i is mapped onto the FPGA}
(18)    HW = HW ∪ T_i
(19)    area = area −T_i.area
(20)   else
(21)    {Map T_i onto the GPP}
(22)    SW = SW ∪ T_i
(23)   end if
(24) end if
```

Algorithm 2: Pseudocode for the cumulative benefit heuristic (CBH) for the mapping on task T_i.

4.2. Cumulative Benefit Heuristic (CBH). CBH maintains a cumulative benefit (CB) value for each task that represents the amount of time that would have been saved up to that point if the task had always been executed onto the FPGA. Mapping decisions are made based on these values and on the available resources. For example, if the available FPGA resources are not sufficient to load the current task, other tasks can be swapped if the CB of the current task is higher than that of the to-be-swapped-out set. Huang and Vahid [22] used this heuristic for dynamic coprocessor management of reconfigurable architectures at architecture level.

Algorithm 2 presents the pseudocode that describes the functionalities of CBH for performing runtime mapping of a task T_i. If resources, such as area slices, are available in the FPGA, then T_i is executed onto the FPGA only if the CB of T_i is larger than its loading time defined by the difference between $T_{SW(i)}$ and $T_{HW(i)}$, where $T_{SW(i)}$ and $T_{HW(i)}$ are the software and the hardware latencies of task T_i, respectively (line 3 to 8 in Algorithm 2). In other cases, when the FPGA lacks current capacity for executing the task, the heuristic searches for a subset of FPGA-resident tasks, such that removing the subset yields sufficient resources in the FPGA to execute the current task. The condition, however, is such that all the tasks in the subset must have smaller CB value than the current task (line 9 to 18 in Algorithm 2). If such a subset is not attained, then the current task is executed by the GPP (line 19 to 22 in Algorithm 2).

```
(1) T ← set of all tasks.
(2) while T ! = ∅ and area ≤ Total_area do
(3)    Select T_i with maximum frequency count
(4)    if area + T_i.area ≤ Total_area then
(5)       map T_i onto the FPGA
(6)       area = area + T_i.area
(7)    else
(8)       map T_i onto the GPP
(9)    end if
(10)   Remove T_i from T
(11) end while
(12) Map rest of the tasks from T onto the GPP
```

Algorithm 3: Pseudocode for the Interval Based Heuristics (IBH) for the mapping on task T_i.

4.3. Interval-Based Heuristic (IBH). In IBH, the execution is divided into a sequence of time slices (intervals) for mapping and scheduling. At the beginning of each interval, a task is examined for its execution. In each interval, the execution frequency of each task is counted, and the mapping decisions are made based on the frequency count of the previous intervals, such that tasks with the highest frequency count are mapped onto the FPGA. In [23], this heuristic is used for resource management in a multithreaded environment at OS level.

Algorithm 3 presents the pseudocode that describes the functionalities of the IBH heuristic for performing runtime mapping in each interval for a set T of tasks. Working from the highest to the lowest frequency count, each task $T_i \in T$ that satisfies the current resource conditions is selected for FPGA execution. The area constraint is updated accordingly before considering the next task. This process continues until the FPGA is full or until there is no task left in T (line 2 to 6 in Algorithm 3). If the FPGA current capacity is not enough for executing any task from T, then these tasks are executed with the GPP (line 8 to 12 in Algorithm 3). As it can be seen in Algorithm 3, tasks are executed onto the FPGA based on frequency count, but other mapping criteria, such as speedup, can also be used.

4.4. Reusability-Based Heuristic (RBH). RBH is based on the hardware configuration reuse concept, which tries to avoid

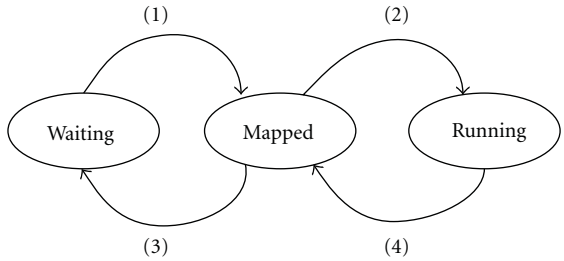

Figure 1: A finite-state machine (FSM) showing the different states of a task.

the reconfiguration overhead by reusing the configurations, which are already available on the FPGA. The basic idea of the heuristic is to avoid reconfiguration as much as possible, in order to reduce the total execution time. Especially in case of application domains, such as streaming and networking, where certain tasks are executed in a periodic manner, for example, on the basis of pixel blocks or entire frames, hardware configuration reuse can easily be exploited. To take advantage of such characteristics of streaming applications, we proposed this heuristic in [11].

For certain tasks that are mapped onto the FPGA, RBH preserves them in the FPGA after their execution. These tasks are not removed from the hardware, so that their hardware configurations can be reused when the task is re-executed. Reusing hardware configurations multiple times can significantly avoid reconfiguration overhead; thus, performance can be considerably improved. Unfortunately, preserving hardware configurations is not possible for all tasks. For this reason, the heuristic tries to preserve hardware configurations for selected tasks. For example, tasks that have higher reconfiguration delay and occur more frequently in the system have priority on being preserved in the FPGA.

We define three states for a task as shown in Figure 1: a waiting state, a mapped state, and a running state. A task is in the waiting state if it waits to be mapped. A task is in the mapped state if it is already configured onto the FPGA, but it is not being executed; however, it may be re-executed later. A task is in the running state when the task is actually processing data. Figure 1 depicts a finite-state machine (FSM) showing the different states of a task, where the numbers 1 to 4 refer to the following state transitions:

(1) area becomes available or task dependency ends;

(2) task execution starts;

(3) other tasks need to be executed;

(4) task execution finishes, but the task may re-execute.

It should be noted that the mapped state has a reconfiguration delay associated with it. If a task transits from a waiting state to a running state, this delay is considered. However, if the task is already in the mapped state, its hardware configuration is saved in the FPGA and this delay is ignored. Thus, when the task needs to be re-executed, it can immediately start processing without reconfiguration. The performance can be significantly improved by avoiding the former transition.

Algorithm 4 presents the pseudocode that describes the functionality of the RBH heuristic for performing runtime mapping of a task T_i. If T_i is already configured, then it starts directly processing data (line 1 to 4 in Algorithm 4). However, if T_i is not currently available in the FPGA, then the task is evaluated for its speedup. If resources are available, T_i is executed onto the FPGA only if there is a performance gain (line 5 to 10 in Algorithm 4). The performance gain in this case is measured in terms of speedup. The speedup for each task is measured at runtime by using the following equation:

$$\text{Speedup} = \begin{cases} \dfrac{T_{\text{SW}}}{T_{\text{HW}}} & t = 0, \\ \dfrac{\text{T}_{\text{SW}} \cdot (\#\text{HWEx} + \#\text{SWEx})}{\#\text{SWEx} \cdot \text{T}_{\text{SW}} + \#\text{HWEx} \cdot \text{T}_{\text{HW}} + \#\text{Recon} \cdot \text{T}_{\text{Recon}}} & t > 0, \end{cases} \tag{7}$$

where #HWEx, #SWEx, and #Recon are the number of HW executions, SW executions, and reconfigurations of a task, respectively. Similarly, T_{HW}, T_{SW}, and T_{Recon} are the corresponding hardware, software, and reconfigurable latencies, and t is the execution time-line. When the application execution starts, $t = 0$. The heuristic maintains a profiling count of HW executions, SW executions, and reconfigurations for all tasks. Each time a task is executed, these counters for that task are updated. For instance, if a task is executed with the GPP, its SW count is incremented, and if the task is executed on the FPGA, its HW count is incremented. Similarly, the reconfiguration count of a task is incremented when a task is (re)configured. These count values for each task are accumulated from all the previous executions. As a result, they reflect the execution history of a task. The speedup calculated with these count values indicates the precise speedup of a task up to that point of execution.

If the available resources are not enough in the FPGA, a set of tasks from the FPGA is swapped to accommodate T_i in the FPGA. The task swapping, in this case, is done based on two factors: (a) speedup and (b) reconfiguration-to-execution ratio (RER). In the first step, a candidate set of tasks from the FPGA is selected, in such a way that these tasks are less beneficial than the current task in terms of speedup (line 12 to 16 in Algorithm 4). The speedup in this case is also calculated by using (7). In the second step, the candidate set is examined for its RER ratio, such that tasks with the lowest RER values are swapped first (line 17 to 21 in Algorithm 4). The RER value for each task is computed as follows:

$$\text{RER} = \frac{T_{\text{Recon}}}{T_{\text{HW}}} \cdot \text{Exec\_Freq}, \tag{8}$$

where Exec_Freq is the average execution frequency of the task in its past history. The execution frequency of a task can be simply computed from the execution profile of each task with respect to the total execution count of that application as follows:

$$\text{Exec}_{\text{Freq}} = \frac{\#\text{HWEx}}{\sum_{i=1}^{N} \text{HW}_i\text{Ex}}, \tag{9}$$

```
(1) {Task already mapped onto the FPGA, do not configure.}
(2) if T_i == MAPPED then
(3)    T_i.state ← RUNNING;
(4) else
(5)    if area ≥ T_i.area then
(6)       if Speedup(T_i) > 1 then
(7)          {Task not mapped onto the FPGA, configure it.}
(8)          configure(T_i);
(9)          T_i.state ← RUNNING;
(10)      end if
(11)   else
(12)    for All tasks T_j onto the FPGA do
(13)          if SpeedUp(T_j) < SpeedUp(T_i) then
(14)             candidateSet = candidateSet ∪T_j
(15)      end if
(16)   end for
(17)   while area ≤ T_i.area do
(18)      Select T_k ∈ candidate Set with lowest RER
(19)      removeSet = removeSet ∪T_k
(20)      area = area + T_k.area;
(21)   end while
(22)   if T_i.area ≤ area then
(23)      for All task T_m ∈ removeSet do
(24)      T_m.state = WAITING;
(25)         end for
(26)         {Task not mapped onto the FPGA, configure it.}
(27)         configure(T_i);
(28)         T_i.state ← RUNNING;
(29)         end if
(30)      end if
(31)   end if
```

Algorithm 4: Pseudocode for the Reusability Based Heuristics (RBH) for the mapping on task T_i.

where the numerator represents the number of times a task is executed on the FPGA. The denominator represents the total hardware execution count of the entire application, and N represents the total number of tasks in the application.

A task with a high RER value indicates that it has high reconfiguration-per-execution delay, and it has executed frequently, in its history in the system, making it a probable candidate for future execution. The heuristic makes a careful selection while removing tasks from the FPGA. By preserving tasks with high RER values as long as possible in the FPGA, we try to avoid the reconfiguration of the frequently executed tasks. We would like to stress the fact that the speedup value computed using (7) is not a constant factor. This value is continuously updated based on the execution profile of the task at runtime. Hence, mapping tasks onto the FPGA based on such value represents the precise system behavior at that instance of time. Note that the RBH is a generic heuristic, and it is not restricted to one type of resource or to one type of architecture. To perform runtime mapping decisions considering multiple resources (such as memory or DSP slices) for different architectural components, the parameters defining the heuristic can be easily customized, hence making it a flexible approach.

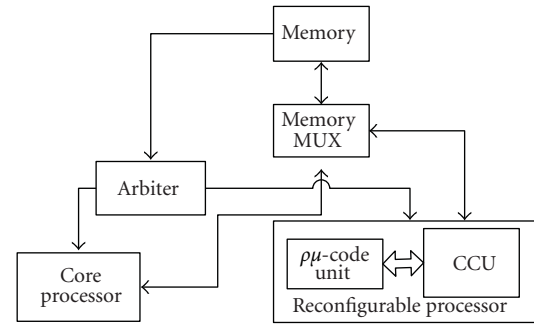

Figure 2: The machine organization of the Molen reconfigurable architecture. The architecture consists of a General Purpose Processor (GPP) and a Reconfigurable Processor (RP), which are coordinated by an arbiter.

4.4.1. The Molen Architecture. Figure 2 depicts the machine organization of the Molen polymorphic processor that is established on the basis of the tightly coupled coprocessor architectural paradigm [34, 35]. It consists of two different kinds of processors: the core processor that is a GPP and a Reconfigurable Processor (RP), such as an FPGA.

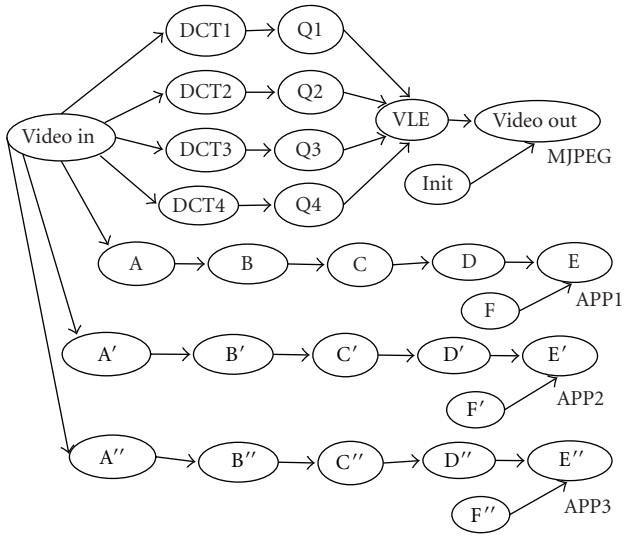

Figure 3: The Motion-JPEG (MJPEG) application model considered for the case study. The MJPEG application is extended by injecting sporadic applications in each frame.

The reconfigurable processor is further subdivided into the reconfigurable microcode ($\rho\mu$-code) unit and a *Custom Computing Unit* (CCU). The CCU is executed on the FPGA, and it supports additional functionalities, which are not implemented in the core processor. In order to speed up the program execution, parts of the code running on a GPP can be implemented on one or more CCUs.

The GPP and the RP are connected to an arbiter. The arbiter controls the coordination of the GPP and the RP by directing instructions to either of these processors. The code to be mapped onto the RP is annotated with special `pragma` directives. When the arbiter receives the `pragma` instruction for the RP, it initiates an "*enable reconfigurable operation*" signal to the reconfigurable unit, gives the data memory control to the RP, and drives the GPP into a waiting state. When the arbiter receives an "*end of reconfigurable operation*" signal, it releases the data memory control back to the GPP and the GPP can resume its execution. An operation executed by the RP is divided into two distinct phases: `set` and `execute`. In the `set` phase, the CCU is configured to perform the supported operations, and in the `execute` phase the actual execution of the operation is performed. The decoupling of `set` and `execute` phase allows the `set` phase to be scheduled well ahead of the `execute` phase and thereby hiding the reconfiguration latency.

4.4.2. The Application Model. We extend a Motion-JPEG (MJPEG) encoder application to use it as an application model for this case study. The corresponding KPN is shown in Figure 3. The frames are divided into blocks, and each task performs a different function on each block as it is passed from task to task. MJPEG operates on these blocks (partially) in parallel. A random number (0 to 3) of applications (APP1 to APP3) is injected in each frame of the MJPEG application in order to create a dynamic application behavior. These applications are considered as sporadic ones, which randomly appear in the system and compete with MJPEG for the resources. In this case study, we want to evaluate task mapping under different resource conditions; therefore we use only one application as a benchmark for comparing different heuristics. Nevertheless, the rSesame framework allows to evaluate any number of applications, architectures, and task mapping heuristics.

Table 1: Available area (in slices) for different FPGAs from the Xilinx Virtex4 FX family [36].

Hardware	Area (slices)
XC4VFX12	5472
XC4VFX20	8544
XC4VFX40	18624
XC4VFX60	25280
XC4VFX100	42176
XC4VFX140	63168

4.4.3. Experimental Setup. As discussed before, for this case study, we consider a model instantiated from the rSesame framework for the Molen reconfigurable architecture. The model instantiated for this case study consists of 30 CCUs allowing each task to be mapped onto one CCU. Note that the number of CCUs is a parameter that can be defined based on the number of pageable and HW tasks. For this case study, we consider all tasks as pageable to fully exploit the runtime mapping by deciding *where* and *when* to map them at runtime depending on the system condition. The model allows dynamic partial reconfiguration and, therefore, if the FPGA cannot accommodate all tasks at once, the latter can be executed after runtime reconfiguration.

We study and evaluate different task mapping heuristics from various domains by considering, for the same architecture model, different FPGA sizes. We consider six FPGAs from the Xilinx Virtex-4 FX family [36], namely, XC4VFX12, XC4VFX20, XC4VFX40, XC4VFX60, XC4VFX100, and XC4VFX140. These FPGAs have different available area (slices) as shown in Table 1. As a result, they are used to evaluate the runtime task mapping under different resource conditions. Note that, in this case study, we have used area as one dimensional space. Nevertheless, rSesame can evaluate any other types and numbers of architectural parameter. We assume that the Processor Local Bus (PLB) of these FPGAs is 4 bytes wide, and the Internal Configuration Access Port (ICAP) functions at 100 MHz; thus, its configuration speed is considered at 400 MB/sec [37].

We use estimated values of the computational latency, the area occupancy (on the FPGA), and the reconfiguration delay for each CCU. The computational latency values for the GPP model are initialized using the estimates obtained from literature [38, 39] (non-Molen specific).

We estimated area occupancy for each process mapped onto the CCU using the Quipu model [40]. Quipu establishes a relation between hardware and software, and it

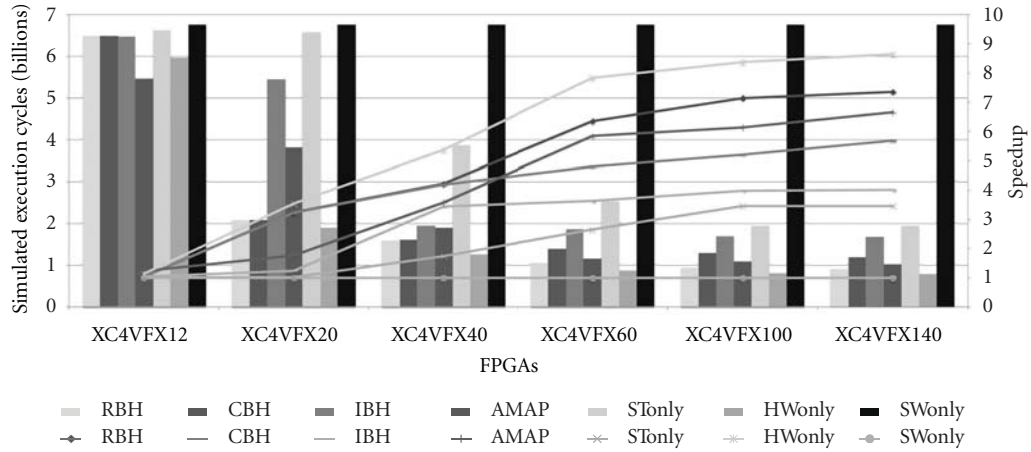

Figure 4: Comparison of the different heuristics tested in the proposed case study under different FPGAs conditions in terms of simulated execution time with corresponding application speedup. The application performance is proportional to the FPGA size. HWonly mapping has the best performance followed by RBH, AMAP, CBH, and IBH. STonly has the worst performance.

predicts FPGA resources from a C-level description of an application using Partial Least Squares Regression (PLSR) and Software Complexity Metrics (SCMs). Kahn processes contain functional C-code together with annotations that generate events such as read, execute, and write. As a result, Quipu can estimate area occupancy of each Kahn process. Such estimations are accepted while exploring systems at very early design stages with rSesame. In later design stages, other more refined models can be used to perform more accurate architectural explorations.

Based on the reconfiguration delay of each FPGA and the estimated area of each Kahn process, we computed the reconfiguration delay of each CCU using the following equation:

$$T_{\text{Recon}} = \frac{\text{CCU\_slices}}{\text{FPGA\_slices}} \cdot \frac{\text{FPGA\_bitstream}}{\text{ICAP\_bandwidth}}, \quad (10)$$

where CCU_slices is the total number of area slices a CCU requires, FPGA_slices is the total number of slices available on a particular FPGA, FPGA_bitstream is the bitstream size in MBs of the FPGA, and ICAP_bandwidth is the ICAP configuration speed. As a final remark, we assume that there is no delay associated with the runtime mapping, such as task migration and context switching.

5. Heuristics Evaluation

In this section, we provide a detailed analysis of the experimental results and their implications for the aforementioned case study. We conducted a wide variety of experiments on the above-mentioned task mapping heuristics with the Molen architecture by considering various FPGAs of different sizes. We evaluated and compared these heuristics based on the following parameters:

(i) the execution time,

(ii) the number of reconfigurations,

(iii) the percentage of hardware/software executions,

(iv) the reusability efficiency.

The detailed description of these parameters has been provided in Section 3. In the rest of this section, we discuss the evaluation results by using these parameters in more detail.

5.1. Execution Time. Figure 4 depicts the results of running different task mapping heuristics for mapping an extended MJPEG application onto the Molen architecture with various FPGAs of different sizes. The primary y-axis (left) in the graph represents the application execution time measured for each heuristic. The software-only (SWonly) execution is measured when all the tasks are mapped onto the GPP. Similarly, the hardware-only (HWonly) execution is measured when all the tasks are mapped onto the FPGA. In HWonly, tasks are forced to be executed on the FPGA. However, if the task does not fit on the entire FPGA, the task is executed on the GPP. The static execution (STonly) is measured when only design time exploration is performed. In STonly execution, a fixed set of hardware tasks is considered for the FPGA mapping and this set does not change during the application runtime. For this experiment, tasks considered as fixed hardware are DCT1–DCT4 and Q1–Q4. The secondary y-axis (right) in Figure 4 represents the application speedup for each heuristic compared to the SWonly execution. The x-axis lists different types of FPGAs, which are ranked (from left to right) based on their sizes, such that XC4VFX12 has the smallest number of area slices and XC4VFX140 has the largest number of area slices (see Table 1). Several observations in terms of FPGA resources and speedup for different heuristics can be made from Figure 4.

A first observation that can be noticed from Figure 4 is that the application performance is proportional to the FPGA size to a certain degree: the bigger the available area in the FPGA, the higher the application performance. In the case of XC4VFX12, there is no significant performance gain

Table 2: The performance increase in different heuristics with the corresponding area increase in the FPGA. There is no linear relation between the area and the corresponding performance improvement.

Heuristics	Performance increase (%)	
	XC4VFX12 ⇒ XC4VFX20 (54% slice increase)	XC4VFX100 ⇒ XC4VFX140 (33% slice increase)
HWonly	67.9	3.14
STonly	0.69	0.007
AMAP	30	7.7
IBH	15	0.87
CBH	67	8.5
RBH	70	2.8

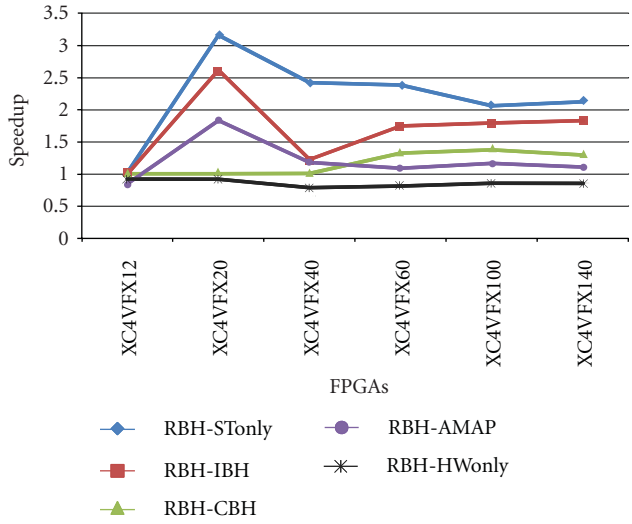

Figure 5: The performance increase of the RBH compared to HWonly, STonly, IBH, CBH, and AMAP. RBH performs better than AMAP under all resource conditions except XC4VFX12. RBH performs better than STonly, IBH, and CBH under all resource conditions.

by using any heuristic compared to the software execution. As there is a limited area, only few tasks can be mapped onto the FPGA; thus, performance is limited. Nevertheless, there is a notable performance improvement with the other FPGAs.

Secondly, while comparing the results of different heuristics for different FPGAs in Figure 4, we observe that there is no linear relation between the FPGA area and the corresponding performance. For instance, although XC4VFX20 has 54% more slices than XC4VFX12, the corresponding increase in the application performance is 67.9%, in the case of HWonly, as shown in Table 2. Similarly, there is 33% increase in area slices while comparing XC4VFX140 with XC4VFX100 in Table 1. Nevertheless, there is considerably lower increase in the performance in this case, as compared to the former case. The performance increase associated with the corresponding area increase in XC4VFX12 and XC4VFX20 as compared to XC4VFX100, and XC4VFX140 respectively, in case of different heuristics is reported in Table 2. The table depicts that there is no linear increase in the performance with area increase. This becomes obvious as the performance increase in an application is bounded by the degree of parallelism in that application. The use of more resources does not always guarantee a better application performance.

Another observation that can be made from Figure 4 is in terms of application performance of each heuristic. As it can be seen from the figure, STonly has the worst application performance, and HWonly has the best application performance. HWonly executes *all* tasks on the FPGA. As a result, it has approximately up to 9 times better performance than SWonly. STonly executes a fixed set of tasks on the FPGA, and mapping optimizations cannot be performed at runtime and, as a result, it has only upto 3 times better performance than SWonly. On the other hand, with runtime heuristics such as AMAP, IBH, CBH, and RBH, the task mapping is performed at runtime. When the application behavior changes due to the arrival of a sporadic application, task mapping is optimized, and better performance can be obtained in latter cases. This can be clearly seen in the figure, where the performance of the other heuristics, such as RBH, CBH, IBH, and AMAP, are bounded by HWonly and STonly.

While comparing the application performance of RBH against the other heuristics, we observe that RBH provides the best performance. RBH outperforms IBH under all resource conditions. RBH performs similar to CBH in the case of XC4VFX12, XC4VFX20, and XC4VFX40, while it performs better than CBH for the rest of the FPGAs. Task mapping is highly influenced by the task selection criteria and the FPGA size. CBH chooses a task with the highest SW/HW latency difference and executes that task in FPGA. RBH also maps tasks based on the speedup factor, but the major difference is in the way this value is calculated. RBH calculates the speedup value at runtime taking into account the past execution history, while with CBH, the SW/HW value is calculated statically. This difference significantly influences the performance of these heuristics. The performance increase of the RBH as compared to HWonly, STonly, IBH, CBH, and AMAP is reported in Figure 5. As it can be inferred from the figure, the performance improvement of the RBH compared to AMAP shows an irregular behavior. The RBH performs 10% worse than AMAP for XC4VFX12. However, the improvement significantly increases for XC4VFX20. For XC4VFX40, the improvement suddenly decreases to 10%. The improvement is regained for XC4VFX60 and stays identical for XC4VFX100 and XC4VFX140. AMAP performs task mapping based on the area availability in an ad hoc manner, in the sense that it tries to map as many tasks as possible at once. However, the RBH performs a selective task mapping based on the task speedup and the hardware configuration reuse. When area is limited, as in the case of XC4VFX12, not many hardware configurations can be preserved in the FPGA. Thus, configuration reuse cannot

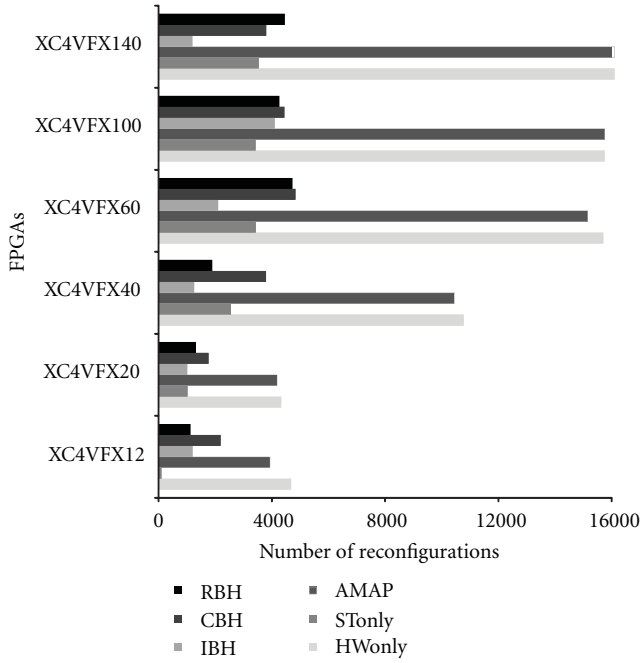

Figure 6: Heuristics comparison under different FPGAs conditions in terms of number of reconfigurations. There is a direct relation between the number of reconfigurations and the FPGA area.

be exploited with the RBH. As a result, AMAP performs better than RBH. With the increase in area, many hardware configurations can be preserved in the FPGA. Consequently, the RBH performs better than AMAP.

5.2. Number of Reconfigurations. Figure 6 depicts an overview of the number of reconfigurations for different heuristics, by considering different FPGAs. Several observations can be made from Figure 6 in terms of FPGA resources and the number of reconfigurations for the different heuristics. Only few tasks can be executed on the FPGA with limited area slices, contributing to the small reconfiguration counts. When the area slices increase, more tasks can be executed onto the FPGA, and, hence, reconfiguration counts increase. Nevertheless, the reconfiguration count is greatly influenced by the mapping strategies used. As it can be inferred from Figure 6, HWonly has relatively higher reconfigurations as compared to other heuristics. With HWonly, *all* tasks are executed to the FPGA, and hence, they are configured frequently. In large FPGAs, there is a possibility for CCUs to save and reuse their configurations and, hence, to avoid their reconfiguration. Therefore, reconfigurations saturate with large FPGAs. Similarly, STonly has a relatively low number of reconfigurations with small FPGAs, such as XC4VFX12 and XC4VFX20. The reconfiguration count increases in case of XC4VFX40 and XC4VFX60, and, then, it stays constant in all other cases. STonly executes a fixed set of HW tasks in all cases; since the number of HW task is constant, the reconfiguration also saturates.

We can observe from Figure 6 that AMAP has significantly higher reconfiguration counts unlike the other heuristics. AMAP performs task mapping based on the area availability in an adhoc manner, in the sense that any task can be mapped onto the FPGA. This leads to a significant increase in reconfiguration counts. It is worth noticing that the application performance in case of AMAP does not decrease drastically with the higher reconfiguration numbers. We may expect a significant performance decrease due to massive reconfigurations. The reconfiguration latency considered for a task is relatively small compared to the HW execution latency. Despite the larger number of reconfigurations, the performance can be considerably improved with the HW execution in such cases. Similarly, in the case of CBH, the reconfiguration counts are lower in smaller FPGAs due to lower hardware executions. This number increases with large FPGAs. There are no significant changes in the reconfiguration counts with the increase in area slices once sufficient area is available.

The number of reconfigurations for IBH is somewhat lower compared to the other heuristics, such as AMAP, CBH, and RBH under all FPGA conditions. This is not due to an efficient algorithm, which tries to optimize the reconfiguration delay, rather it is the effect of limited HW execution. In case of IBH, the mapping decision is changed only in the beginning of each interval, and the mapping behavior is fixed within an interval. Thus, a fixed set of tasks is mapped onto the FPGA during such an interval. This limits the hardware execution percentage, and hence, the reconfigurations. On the other hand, RBH reuses the hardware configurations to reduce the total number of reconfigurations. As a result, we observe a lower number of reconfigurations in case of RBH compared to CBH and AMAP in Figure 6. Note that IBH and STonly have lower reconfigurations than RBH as a consequence of their lower hardware execution. Nonetheless, RBH has a better reconfiguration-to-HW-execution ratio as compared to IBH and STonly, making the former better in terms of performance.

5.3. Percentage of Hardware Execution, Software Execution, and Reconfiguration. Figure 7 shows the comparison between different task mapping heuristics in terms of hardware execution, software execution and reconfiguration measured using (1), (2), and (3), respectively. The x-axis in the graph is stacked as 100%, and it shows the contribution of hardware execution, software execution, and reconfiguration to the total execution time. We observe that in few FPGAs the percentage of execution is greater than 100%. The hardware execution percentage measured in (1) is provided as an upper bound to address the parallel execution possibility of the FPGA. As a result, its value can go beyond the 100% limitation.

A first observation that can be made from Figure 7 is in terms of execution percentage and FPGA area. The limited area slices in the FPGA confines the HW execution percentage in smaller FPGAs. The hardware execution percentage increases considerably with more area slices, but this increase is not linear. As it can be seen from the figure, hardware

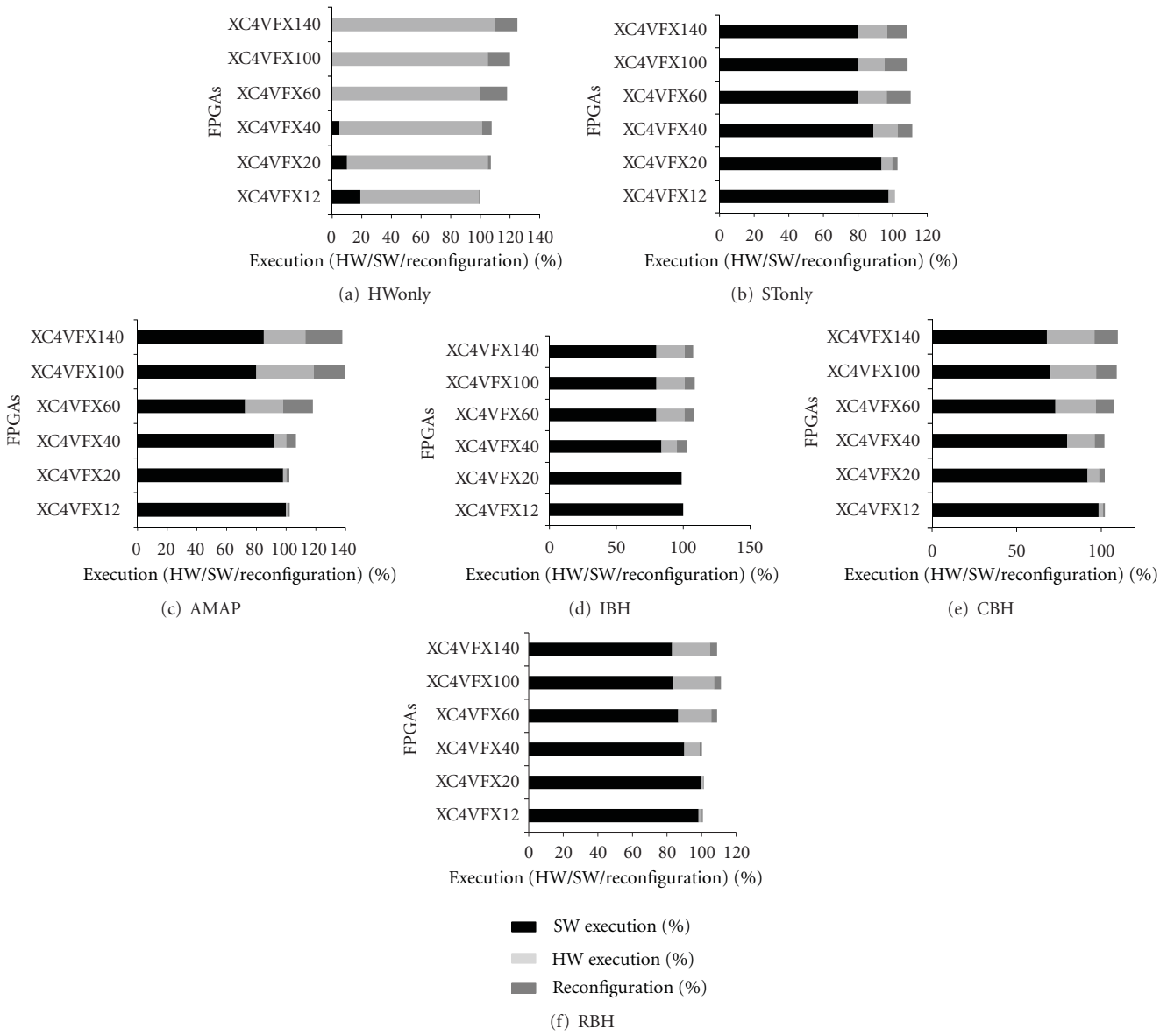

Figure 7: The comparison of different heuristics based on percentage of hardware execution, software execution, and reconfiguration. The hardware execution percentage is low in smaller FPGAs, and it increases considerably with more area slices.

execution percentage somewhat saturates with large FPGAs, such as XC4FX100 and XC4FX140. This observation is valid for all the runtime mapping heuristics including STonly and HWonly.

With HWonly mapping, *all* tasks are forced to be executed to the FPGA. However, if the task does not fit on the entire FPGA, then the task is executed with the GPP. Therefore, in Figure 7, we observe certain percentage of software execution with small FPGAs, but, with larger FPGA, there is only HW execution and the corresponding reconfiguration. With smaller FPGAs, almost no tasks are executed in hardware and, as a result, STonly has very minimal hardware execution (if any) and, therefore, the less reconfigurations. With the larger FPGAs, STonly has a relatively good but constant hardware execution and reconfiguration percentage, since it executes a fixed set of tasks on the FPGA.

While comparing the runtime heuristics, such as AMAP, CBH, IBH, and RBH, we can observe that AMAP has the best hardware execution percentage in larger FPGAs, followed by RBH and CBH. CBH and RBH somehow show similar behavior in terms of hardware execution percentage. However, in case of reconfiguration percentage, they do not follow the same trend. The reconfiguration is somewhat linear to the hardware execution in case of CBH. However, RBH does not show any linear increase in reconfiguration with hardware execution. RBH performs task mapping based on configuration reuse and, as a result, tries to avoid reconfiguration with more hardware executions. This behavior of RBH heuristic is apparent in the figure, especially

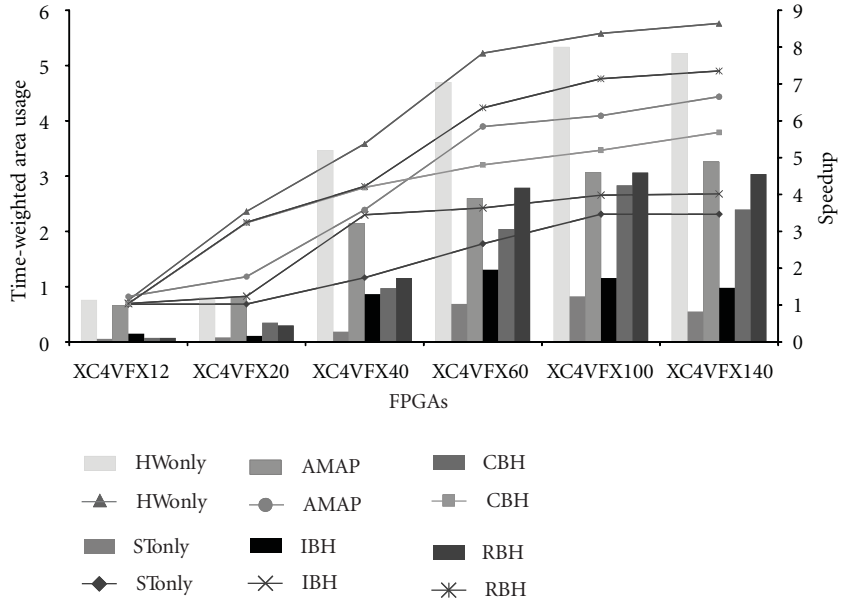

Figure 8: The comparison of different heuristics in terms of time-weighted area usage against speedup under different FPGAs conditions. The HWonly has the most time-weighted area usage, followed by the other runtime heuristics AMAP, CBH, IBH, and RBH.

in the case of moderate to large FPGAs, such as XC4FX60, XC4FX100, and XC4FX140. IBH follows a behavior similar to STonly in terms of software and hardware execution, as it also executes a fixed set of tasks on the FPGA.

By mapping more tasks onto the FPGA, the application can be accelerated, but it also has reconfiguration overhead. The efficiency of the mapping heuristics lies in finding the best mapping while minimizing the number of reconfigurations. Nevertheless, in Figure 7, we see almost a linear contribution of the reconfiguration overhead to the total execution time in all heuristics, except in RBH. This phenomenon is highly influenced by the policy implemented for task mapping. Another observation that can be made from the figure is the contribution of the hardware execution, SW execution, and reconfiguration to the total execution time. The figure shows that the GPP executes most of the application and the FPGA computes only less than 40% of the total application. This is due to the architectural restrictions of the Molen architecture. The GPP and the RP run in a mutual exclusive way, due to the processor/coprocessor nature of the architecture. This influences the mapping decision, which, in turn, contributes to the low hardware execution rates. This significantly increases the total execution time. Another reason for the lower percentage of hardware execution is due to the lower hardware latency for each task. The execution percentage is calculated as the ratio of execution latency of all tasks to the total execution time of an application. The hardware latency is comparatively lower than the SW latency for each task. Therefore, the corresponding hardware execution contribution is always lower compared to the percentage of SW execution.

5.4. Time-Weighted Area Usage. Figure 8 depicts an average time-weighted area usage measured using Equation (4) for different heuristics under different FPGA devices. The primary y-axis (left) in the graph represents the time-weighted area usage measured for each heuristic. The secondary y-axis (right) in the figure represents the application speedup for each heuristic compared to the SWonly execution. Several observations can be made from Figure 8 in terms of FPGA resources and time-weighted area usage of different heuristics. The first observation that can be made from the figure is in terms of time-weighted area usage and the hardware resource. As it can be seen from the figure, the time-weighted area usage is directly impacted by the number of area slices in the FPGA. With the limited area slices in small FPGAs, few tasks are executed in the FPGA, contributing to a smaller number of hardware executions. This, in turn, contributes to the lower area usage. With sufficient area slices, there is a considerable number of hardware executions and, hence, the area usage is high. Nonetheless, there is no linear relation between the time-weighted area usage and the available FPGA area. In XC4VFX140, the area usage is relatively low compared to XC4FX100, despite the fact that area slices are greater in the former. The area usage measured is the time-weighted factor, and it depends on the hardware execution, the total FPGA area and the total execution time, as shown in Equation (4). The increase in the area slices, with no significant increase in hardware executions, contributes to the lower area usage in the former case.

As it can be inferred from Figure 8, HWonly has the highest time-weighted area usage under all FPGA conditions. HWonly executes *all* tasks onto the FPGA and, as a result, the cost of using FPGA in this case is higher than all the other heuristics. STonly, however, has the lowest area usage due to its lower number of hardware executions and, therefore, its corresponding performance is also very poor. Similarly, AMAP has higher area usage compared to

other heuristics, such as CBH, IBH, and RBH, under all FPGA conditions, except XC4VFX60. AMAP performs task mapping based on area availability. As a matter of fact, it has a relatively higher number of hardware executions compared to the other heuristics and, therefore, it consumes additional area. RBH, on the other hand, has less time-weighted area usage. While comparing AMAP and RBH, we can observe that RBH performs somewhat better than AMAP in terms of performance. This implies that RBH reuses the hardware configuration already present in the FPGA to avoid reconfiguration overhead and, as a result, it can give better performance with the same amount of hardware resources as required by AMAP. Likewise, CBH has a comparable percentage of time-weighted area usage, but it lags behind in terms of speedup as compared to RBH. However, IBH has a considerably low percentage of area usage, as it also has lower hardware executions due to the constantly executed HW task set, and hence it also has lower performance. We can summarize that HWonly has the best performance but consumes more hardware resources. STonly has the lowest area usage but straggles behind in terms of performance. A tradeoff in terms of performance and resources can be obtained with task mapping at runtime, which performs selective task mapping onto the FPGA at runtime.

Another compelling observation that can be made from Figure 8 is about the lower value of the time-weighted area usage. The Molen architecture is based on processor/coprocessor paradigm. As a result, the GPP and the RP run in a mutual exclusive. This contributes to the lower number of hardware executions, which consequently increases the total execution time. Thus, these two factors significantly contribute to the low value of area usage. The area usage can be increased either by mapping more tasks onto the FPGA or by operating the RP and the GPP in parallel.

5.5. Reusability Efficiency. Figure 9 depicts the reusability efficiency (RE_{Task}) recorded for all CCUs using Equation (5) for different heuristics under different FPGA conditions. Several observations can be made from the figure in terms of FPGA area and the RE_{Task} of each CCU. Firstly, we observe that the CCU reuse is significantly affected by the number of area slices in the FPGA. Small FPGAs, such as XC4VFX12 and XC4VFX20, have many CCUs with RE_{Task} value zero. A CCU has an RE_{Task} value of zero under the following conditions:

(i) when a CCU is always mapped onto the GPP

(ii) when a CCU is configured every time it is executed on the FPGA.

With few resources in the FPGA, only a limited number of tasks can be executed to the FPGA. Additionally, in such cases, hardware configurations cannot be preserved for future reuse. As a result, CCUs have an RE_{Task} value of zero. Moreover, in this case, CCUs that are reused have a very small size in terms of area. With the increase in number of slices in the FPGA, more CCUs are reused. Medium-sized FPGAs, such as XC4VFX40 and XC4VFX60, reuse more CCUs compared to smaller FPGAs, but, in such cases, the reuse percentage is still low. With the larger FPGAs such as XC4VFX100 and XC4VFX140, more CCUs are reused with large RE_{Task} value.

As it can be inferred from Figure 9, HWonly has the best RE_{Task} for many CCUs in large FPGAs, such as XC4VFX100 and XC4VFX140. HWonly executes all the tasks on the FPGA and, as a result, it has high hardware execution count. However, with small FPGAs, all the tasks are configured, due to area restrictions, and there is no configuration reuse. On the other hand, with larger FPGAs, more configurations are saved and reused and, as a consequence, many CCUs have a considerably high RE_{Task} value. Similarly, STonly always maps a set of fixed tasks onto the FPGA. Out of these tasks, only a few number of small tasks can be reused. We notice that these CCUs have a relatively higher RE value compared to the ones reused with the AMAP heuristic in the figure. AMAP has higher HW execution percentage as compared to IBH and CBH. As a matter of fact, many tasks are reused in case of AMAP, but the reuse percentage of these CCUs is low. AMAP has no fixed pattern for task execution and, as a result, any task can be executed in FPGA. Therefore, the reusability is rather distributed among many CCUs. CBH, on the other hand, follows a specific policy for task execution in FPGA and hence executes a fixed set of selected task. As a result, a set of selected tasks is reused. Similar behavior is observed in the case of CBH. As it also executes a set of specific task within an interval, same tasks are reused (if any).

Likewise, from Figure 9, we observe that the RE_{Task} of RBH is better than that of other runtime heuristics for many CCUs. The impact of this hardware configuration reuse, in case of RBH, can be directly seen in terms of performance gain in Figure 4, where RBH has better speedup than the other heuristics. From Figure 10, we also observe that, for few tasks, RE_{Task} decreases when FPGA resources increase. With larger FPGAs, more tasks can fit onto the FPGA. As a result, these tasks are also mapped onto the FPGA, thus, over writing the saved configurations of other tasks. RE_{Task} for few tasks decreases in the FPGAs with moderate size. With the abundant resources, the hardware configuration can be saved for more tasks, and RE_{Task} increases again.

Note that STonly, AMAP, CBH, and IBH do not map the task based on the hardware configuration reuse. The reuse obtained in the case of STonly, AMAP, CBH, and IBH is a default value determined based on the arrival of the application task. If a CCU is already configured on the FPGA when its corresponding task arrives, the task can be executed without reconfiguration. However, the RBH reuses more hardware configurations than the other heuristics on top of the default value obtained.

Figure 10 depicts the total RE_{app} recorded using Equation (6) for different heuristics under different resource conditions. In the figure, we again observe that the reusability increases when using larger FPGAs. HWonly executes *all* the tasks in FPGA and, therefore, there can be a possibility that many of these tasks are reused when sufficient area is available, resulting into higher RE_{app}. STonly has almost a constant RE_{app} in larger FPGA, since it executes a constant set of tasks in FPGA. While comparing runtime heuristics, such as AMAP, CBH, IBH, and RBH, we can observe that

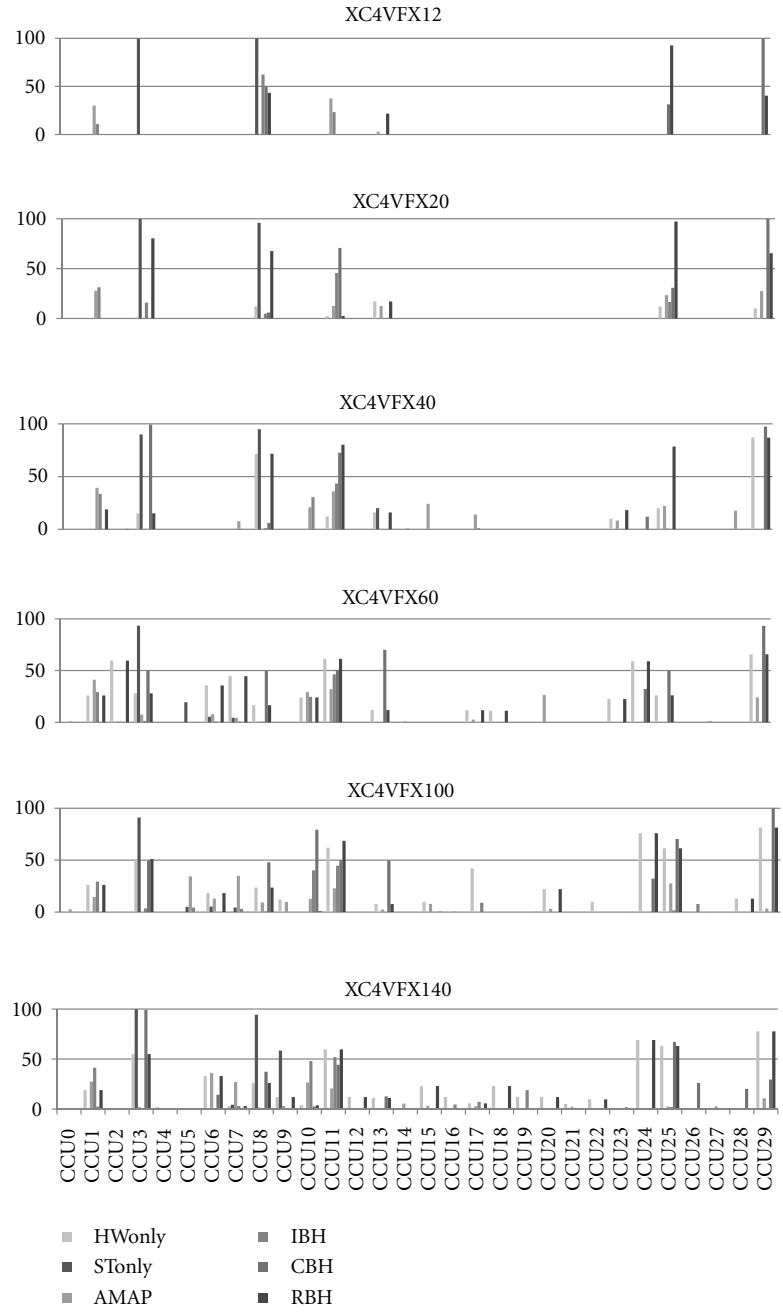

Figure 9: Reusability efficiency (RE_{Task}) of CCUs for different heuristics under different FPGA conditions. The CCU reuse is significantly affected by the number of area slices in the FPGA.

since the RBH has more CCUs reused than other heuristics, as shown in Figure 7(f), RBH has a better RE_{app} value than other heuristics in all resource conditions but XC4VFX12. Since XC4VFX12 has less area, all the heuristics have approximately the same value for RE_{app}. RE_{app} is the accumulation of the time saved due to hardware configuration reuse of each CCU. If all CCUs obtain the same value of the RE for a task mapping heuristic, then the application RE_{app} depends on the corresponding total execution time of that heuristics.

6. Observations and Conclusions

In this paper, we evaluated the task mapping of application(s) onto reconfigurable architectures under different resource conditions. We thoroughly evaluated various task mapping heuristics from the state of the art with the rSesame framework for a reconfigurable architecture with different FPGA resources using an extended MJPEG application. Based on the evaluation discussed in the previous sections, we can summarize the following conclusions.

(i) The comparison of different FPGAs shows that with very limited resources (in case of small FPGAs), the

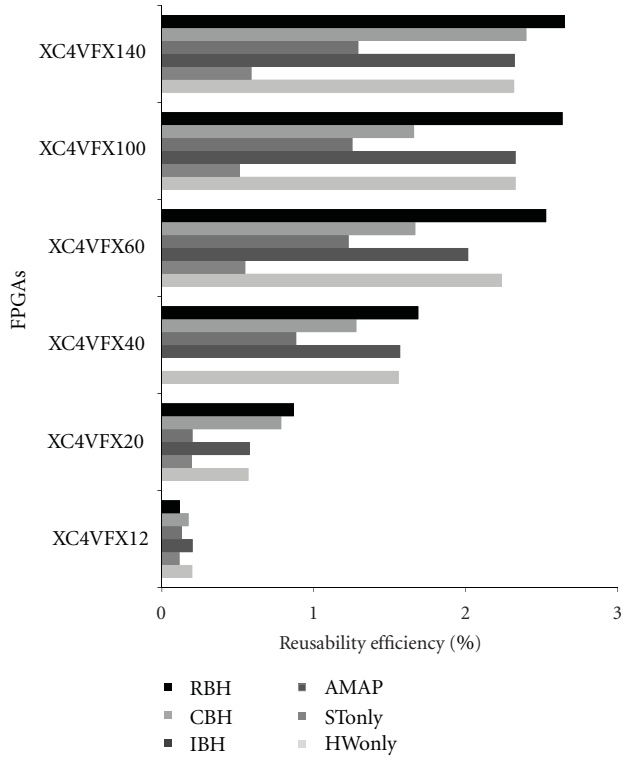

Figure 10: Heuristics comparison under different FPGAs conditions in terms of application reusability efficiency (RE_{app}). RBH has better RE_{app} compared to other heuristics.

number of tasks that can be mapped onto the FPGA is low. Consequently, these tasks are mapped onto the GPP. This leads to a poor application performance.

(ii) More resources (in case of moderate/higher FPGAs) imply more tasks mapped onto the FPGA. Consequently, we can obtain better application performance.

(iii) Runtime mapping provides better performance in case of dynamic application/architecture conditions. If the application behavior is well known in advance, design time mapping can give equal performance.

(iv) Mapping all the tasks onto the FPGA gives better performance, but it consumes more hardware resources. Runtime mapping performs task mapping based on the runtime system conditions. As a matter of fact, with the runtime mapping, a tradeoff can be obtained in terms of performance and resources.

(v) Comparing different heuristics, in case of limited resources conditions (small FPGAs), the adhoc task mapping of AMAP performs better compared to CBH, IBH, and RBH. Due to limited resources, the careful task selection with RBH, CHB, and IBH cannot be fully exploited in such cases.

(vi) The reuse of hardware configurations is better in case of sufficient resource conditions (medium-to-large FPGAs). As a result, the configuration reuse can be well exploited. Additionally, the RBH provides better application performance than AMAP and CBH.

(vii) In case of abundant resource conditions (very large FPGAs), the performance saturates due to application constraints. Under such scenarios, all the heuristics have similar performances.

Acknowledgments

This research is partially supported by Artemisia iFEST Project (Grant 100203), Artemisia SMECY (Grant 100230), FP7 Reflect (Grant 248976).

References

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

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

[3] S. Hauck, "The roles of FPGA's in reprogrammable systems," *Proceedings of the IEEE*, vol. 86, no. 4, pp. 615–638, 1998.

[4] R. J. F. Stephen, D. Brown, and J. Rose, *Field-Programmable Gate Arrays*, vol. 180 of *The Springer International Series in Engineering and Computer Science*, Kluwer Academic Publishers, 1992.

[5] Xilinx Corporation, *Coolrunner-II CPLDs Family Overview*, September 2008.

[6] R. W. Hartenstein and R. Kress, "A datapath synthesis system for thereconfigurable datapath architecture," in *Proceedings of the Asia andSouth Pacific Design Automation Conference (ASP-DAC '95)*, pp. 479–484, September 1995.

[7] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*, vol. 16, Springer, Berlin, Germany, 2007.

[8] N. S. Voros and and K. Masselos, *System-Level Design of Reconfigurable Systems-on-Chip*, Springer, Berlin, Germany, 1st edition, 2005.

[9] K. Sigdel, M. Thompson, C. Galuzzi, A. D. Pimentel, and K. Bertels, "rSesame —a generic system-level runtime simulation framework for reconfigurable architectures," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '09)*, pp. 460–464, 2009.

[10] K. Sigdel, M. Thompson, C. Galuzzi, A. D. Pimentel, and K. Bertels, "Evaluation of runtime task mapping heuristics with rSesame—a case study," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '10)*, pp. 831–836, deu, March 2010.

[11] K. Sigdel, C. Galuzzi, K. Bertels, M. Thompson, and A. D. Pimentel, "Runtime task mapping based on hardware configuration reuse," in *Proceedingsof the International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 25–30, 2010.

[12] P. V. Knudsen and J. Madsen, "PACE: a dynamic programming algorithm for hardware/software partitioning," in *Proceedings of the 4th International Workshop on Hardware/Software Co-Design (Codes/CASHE '96)*, pp. 85–92, March 1996.

[13] M. Kaul and R. Vemuri, "Design-space exploration for block-processing based temporal partitioning of run-time reconfigurable systems," *Journal of VLSI Signal Processing Systems for*

Signal, Image, and Video Technology, vol. 24, no. 2, pp. 181–209, 2000.

[14] B. Miramond and J. M. Delosme, "Design space exploration for dynamically reconflgurable architectures," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, pp. 366–371, March 2005.

[15] B. Miramond and J. M. Delosme, "Decision guide environment for design space exploration," in *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '05)*, pp. 881–888, September 2005.

[16] L. Y. Li and M. Shi, "Software-hardware partitioning strategy using hybrid genetic and Tabu search," in *Proceedings of the International Conference on Computer Science and Software Engineering (CSSE '08)*, vol. 4, pp. 83–86, 2008.

[17] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of the Annual Workshop on Circuits, Systemsand Signal Processing (ProRISC '00)*, pp. 1–8, November 2000.

[18] C. Haubelt, S. Otto, C. Grabbe, and J. Teich, "A system-level approachto hardware reconfigurable systems," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '05)*, pp. 298–301, 2005.

[19] G. Wang, W. Gong, and R. Kastner, "Application partitioning on programmable platforms using the ant colony optimization," *Embedded Computing*, vol. 2, no. 1, pp. 119–136, 2006.

[20] G. Still, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: a first approach," in *Proceedings of the Design AutomationConference (DAC '03)*, 2003.

[21] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal, "Run-time management of a MPSoC containing FPGA fabric tiles," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 24–33, 2008.

[22] C. Huang and F. Vahid, "Dynamic coprocessor management for FPGA-enhanced compute platforms," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '08)*, pp. 71–78, 2008.

[23] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pp. 149–158, April 2005.

[24] F. Ghaffari, M. Auguin, M. Abid, and M. B. Jemaa, "Dynamic and on-line design space exploration for reconfigurable architectures," *Transactions on High-Performance Embedded Architectures and Compilers*, vol. 4050, pp. 179–193, 2007.

[25] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun, "Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTIMEDIA '06)*, pp. 33–38, October 2006.

[26] O. Moreira, J. J. D. Mol, and M. Bekooij, "Online resource management in a multiprocessor with a network-on-chip," in *Proceedings of the ACM Symposium on Applied Computing (SAC '07)*, pp. 1557–1564, March 2007.

[27] L. T. Smit, J. L. Hurink, and G. J. M. Smit, "Run-time mapping of applications to a heterogeneous SoC," in *Proceedings of the International Symposium on System-on-Chip (SoC '05)*, pp. 78–81, November 2005.

[28] L. T. Smit, G. J. M. Smit, J. L. Hurink, H. Broersma, D. Paulusma, and P. T. Wolkotte, "Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 421–424, December 2004.

[29] P. K. F. Hölzenspies, G. J. M. Smit, and J. Kuper, "Mapping streaming applications on a reconfigurable MPSoC platform at run-time," in *Proceedings of the International Symposium on System-on-Chip (SOC '07)*, pp. 74–77, November 2007.

[30] M. A. A. Faruque, R. Krist, and J. Henkel, "ADAM: run-time agent-based distributed application mapping for on-chip communication," in *Proceedings of the 45th Design Automation Conference (DAC '08)*, pp. 760–765, June 2008.

[31] C. Ykman-Couvreur, E. Brockmeyer, V. Nollet, T. Marescaux, F. Catthoor, and H. Corporaal, "Design-time application exploration for MPSoC customized runtime management," in *Proceesings of InternationalSymposium on System-on-Chip (SOC '05)*, pp. 66–69, 2005.

[32] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–111, 2006.

[33] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, vol. 74, 1974.

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

[35] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. M. Panainte, "The Molen programming paradigm," in *Proceeding of the International workshopon Systems, Architectures, Modeling and Simulation (SAMOS '03)*, pp. 1–30, July 2003.

[36] Xilinx Corporation, "Virtex-4 family overview (V3.0)".

[37] Xilinx DS86, "LogiCORE IP XPS HWICAP (v5.00a)," 2010.

[38] H. Nikolov, M. Thompson, T. Stefanov et al., "Daedalus: toward composable multimedia MPSoC design," in *Proceedings of the 45th annual Design Automation Conference (DAC '08)*, pp. 574–579, 2008.

[39] A. D. Pimentel, M. Thompson, S. Polstra, and C. Erbas, "Calibration of abstract performance models for system-level design space exploration," *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 99–114, 2008.

[40] R. Meeuws, Y. Yankova, K. Bertels, G. Gaydadjiev, and S. Vassiliadis, "A quantitative prediction model for hardware/software partitioning," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 735–739, August 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 168961, 13 pages
doi:10.1155/2012/168961

Research Article

Implementation of Ring-Oscillators-Based Physical Unclonable Functions with Independent Bits in the Response

Florent Bernard, Viktor Fischer, Crina Costea, and Robert Fouquet

Laboratoire Hubert Curien, CNRS, UMR5516, Université de Lyon, 42000 Saint-Etienne, France

Correspondence should be addressed to Florent Bernard, florent.bernard@univ-st-etienne.fr

Received 1 May 2011; Revised 21 September 2011; Accepted 14 January 2012

Academic Editor: Kris Gaj

The paper analyzes and proposes some enhancements of Ring-Oscillators-based Physical Unclonable Functions (PUFs). PUFs are used to extract a unique signature of an integrated circuit in order to authenticate a device and/or to generate a key. We show that designers of RO PUFs implemented in FPGAs need a precise control of placement and routing and an appropriate selection of ROs pairs to get independent bits in the PUF response. We provide a method to identify which comparisons are suitable when selecting pairs of ROs. Dealing with power consumption, we propose a simple improvement that reduces the consumption of the PUF published by Suh et al. in 2007 by up to 96.6%. Last but not least, we point out that ring oscillators significantly influence one another and can even be locked. This questions the reliability of the PUF and should be taken into account during the design.

1. Introduction

Security in integrated circuits (ICs) became a very important problem due to high information security requirements. In order to assure authenticity and confidentiality, cryptographic keys are used to encrypt the information. Several solutions were proposed for key generation, each with their upsides and downsides.

Confidential keys can be generated using True Random Number Generators (TRNGs) and stored in volatile or nonvolatile memories. Saving the confidential key in a nonvolatile memory inside the device ensures that the key will never be lost and that it will not be disclosed in case of passive attacks. On the other hand, nonvolatile memories are easy targets for invasive attacks [1]. Volatile memories are typical for Field Programmable Gate Arrays (FPGAs). Storing the confidential key in a volatile memory permits to erase the memory contents in case of invasive attack detection. This implies the use of a communication channel to transmit the key after device configuration [1]. Communication channels are usually easy to corrupt and information can be easily intercepted. The confidentiality and authenticity of designs are therefore compromised. A solution is backing up the embedded volatile memory block with a battery. However, it was proved that battery-backed RAMs content can be read after a long period of storage [2–5] even if the memory is not powered any more. Thus, the need of generating secret keys inside the IC became obvious.

An alternative to TRNG for key generation is the Physical Unclonable Function (PUF). PUFs are functions that extract a unique signature of an IC, based on randomness during the manufacturing process. This signature can be used as device-dependent key or device identification code. The main advantage of this principle introduced by Pappu et al. in [6, 7] is the fact that the key does not need to be stored in the device and it is thus harder to disclose. Based on intrinsic physical characteristics of circuits obtained during the manufacturing process, the extracted signature is impossible to reproduce by a different IC or by an attacker. PUFs work on challenge-response pairs. The challenge is usually a stimulus sent from outside the device, and the response is the signature of the circuit.

The quality of a PUF is determined mainly by its uniqueness and its reliability. To quantify these properties of a PUF, two types of response variations: intra- (for reliability)

and inter- (for uniqueness) chip variations [8] are used. The intrachip variation refers to the responses of the same PUF (the same device) at the same challenge, regardless of environmental changes (e.g., temperature, voltage). In the ideal case, this variation should be 0. This means that the response of the PUF for a given challenge should always be the same. The intrachip variation measures the reproducibility of the response. The function must be able to reproduce the same response over and over again, especially in the case of reconfigurable devices.

The interchip variation refers to the responses of different PUFs (different devices) at the same challenge. Ideally, this variation should be of 50%, meaning that every bit is equally likely to be zero or one. If this variation is close to 50% then the uniqueness of the responses is guaranteed.

In this paper, we focus on PUF implementation issues in reconfigurable devices and on the independency of bits in the response. Reconfigurable devices are intensively used for implementing cryptographic algorithms on hardware due to the "reconfigurable" property of such circuits. Thus we have to deal with two objectives: to keep the reconfigurable property of FPGAs and to guarantee the uniqueness and reliability of a PUF. In other words, if the PUF response changes when the device is reconfigured, the uniqueness and reliability of a PUF are questionable. We analyze and propose some enhancements of the concept introduced in 2007 by Suh and Devadas [8]. This principle is a ring-oscillators-based PUF (RO-PUF). It was chosen for our experiments because it is one of the most suitable for implementation in FPGAs, independently from the technology. The PUF uses a relatively high number of ring oscillators in order to emphasize the intrinsic characteristics of ICs and extract the signature. The principle is based on the fact that the frequency of ROs depends on gate and routing delays determined partially in an uncontrolled way by the manufacturing process.

In the first part of our work, we had to deal with implementation issues related to the mapping of the PUF to various FPGA technologies. We found out that, contrary to what original authors stated [8], the placement and routing constraints play a very important role (even when ROs are identically laid out) in the design of the function, especially if one wants to obtain sufficient inter-chip variability. The precise control of the initial phase of ROs and careful design of frequency comparators is another important issue that determines the precision of the function and thus reduces intrachip variations. This was not discussed before. Furthermore, in the response there are bits that are dependent on each others. We propose a method justified by mathematical means in order to identify which pairs of ROs we have to select to ensure independency of bits in the response. The main disadvantage of the original design is the high power consumption. We propose a simple modification enabling significant power economy. Finally, during our experiments we observed a very important phenomenon that has a significant impact on the generated results and that was completely neglected in the original design: the existence of a mutual dependence between the ROs can lead sometimes to their mutual locking in FPGAs. It is essential to take into account this unavoidable behavior of ROs in the PUF design.

The paper is organized as follows. In Section 2 we present related work on PUFs implementation and metrics used to measure the quality of a PUF. Section 3 deals with PUF design issues and with the first problem stated: the need of manual placement and routing of the design. Then we remark that some bits in the response might be dependent due to an inappropriate selection of ROs. An example of such a situation and a model of RO pair selection is proposed in Section 4 and a method to identify pairs that will give independent bit in the response is provided. Section 5 presents results of implementation of the RO PUF in main FPGA technologies and analyzes the quality of the PUF in relationship to the selected technology and the quality of the evaluation board. It also evaluates the impact of the mutual dependence of rings on the reliability of the PUF. Section 6 proposes some important enhancement of the function and finally, Section 7 concludes the paper.

2. PUF Background

2.1. Source of Noise in Electronic Devices. From its manufacturing to its usage, an electronic device is faced with many sources of noise coming from different processes and having different signification from one to another. We can distinguish at least three classes of sources of randomness.

(i) In manufacturing process: this noise is due to variation in the silicon layers during the manufacturing process. Once the device is manufactured, it contains these informations which are specific to each integrated circuit. An ideal PUF should be built to extract the maximum amount of this manufacturing noise in order to identify a circuit.

(ii) Local noise: this noise appears when the circuit is working. It is due to the random thermal motion of charge carriers. This noise is very suitable for random number generation but inappropriate for PUF. It should be reduced compared to manufacturing noise to decrease the intrachip variation.

(iii) Global environmental noise: this noise comes from environmental condition (e.g., global temperature and voltage) when the circuit is working. This noise can disrupt the PUF response and increase the intrachip variation making a circuit identification more difficult to perform. Furthermore, this source of noise can be easily manipulated from outside. Therefore, PUFs must be developed in order to reduce the influence of this global environmental noise.

2.2. Related Works and PUF Evaluation. Several concepts of PUF and implementation in reconfigurable devices have already been introduced until now. In [9], the random initialization of SRAM cells in FPGAs is used to generate a specific signature. But in recent FPGAs, manufacturers tend to initialize SRAM cells to a known value that make SRAM cells-based PUF difficult to use. A similar idea is used on FPGA flip-flops and is based on their initial unstable states

[10]. Other PUFs are based on differences in the silicon layers of the device leading to differences between delay paths [8, 11, 12]. The main difficulty in these last designs is to guarantee a perfect symmetry on delay paths in order to exploit the slight differences due to the manufacturing process. Furthermore the placement and routing must be done carefully to exploit the noise due to manufacturing process.

In most of these PUFs, delay paths are implemented with ROs (RO-PUFs). In [13], a new approach is studied in order to use RO-PUF. The so-called *Compensation Method* permits to reduce the influence of unsuitable source of noise on the PUF response. It is realized with Configurable ROs (CROs). One disadvantage pointed by the authors is the reduction of the maximum number of independent bits that can be extracted from such a PUF leading to an increase in the number of ROs that should be used.

As mentioned in introduction, PUF quality is evaluated by its uniqueness and its reliability. In [13], authors proposed two metrics. These metrics cannot directly give the characterization of interdie variation process which can only be estimated based on PUF responses. Thus it depends greatly on how the PUF is implemented to extract the maximum amount of manufacturing noise.

Let (i, j) be a pair of chips with $i \neq j$ and R_i (resp., R_j) the n-bit response of chip i (resp., chip j). The first metric is the *average interdie Hamming Distance (HD)* among a group of k chips and is defined as

$$\overline{\text{Inter} - d_{\text{HD}}}(k) = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{\text{HD}\left(R_i, R_j\right)}{n} \times 100\%. \tag{1}$$

This distance should converge to 50% in the case of an ideal PUF.

The second metric introduced by the authors of [13] is used to ensure the reliability of a PUF. An n-bits response is extracted from chip i (R_i) at normal operating conditions. Then at a different operating condition (different temperature or different voltage), x samples $(R'_{i,y})_{y \in \{1,\ldots,x\}}$ of the response of the same PUF at this operating conditions are extracted. The *average intradie HD* over x samples for the chip i is defined as

$$\overline{\text{Intra} - d_{\text{HD}}}(x, i) = \frac{1}{x} \sum_{y=1}^{x} \frac{\text{HD}\left(R_i, R'_{i,y}\right)}{n} \times 100\%. \tag{2}$$

This distance should be close to 0% to ensure reliable responses from the PUF in a given chip at various operating conditions.

In [14], a deeper analysis on a special PUF (the Arbiter PUF that was originally proposed in [12]) evaluation lead the author to consider 4 indicators on the evaluation of the intrachip variation (randomness, steadiness, correctness, and diffuseness). For the inter-chip variation the metric used is also the uniqueness (expressed differently than in [13]). Even if measurements of uniqueness and reliability seem sufficient to qualify a PUF, it can be interesting to go further (especially in the case of an unexpected high intrachip variation).

3. PUF Design Issues

3.1. Principle of the PUF and Its Implementation in FPGA. In the principle of the PUF published in [8] that was selected for our experiments, N identically laid-out ROs are placed on the IC. Slight differences between their frequencies will appear because of the unavoidable differences in the silicon layers of the semiconductor device caused by the manufacturing process. Pairs of oscillators are chosen one after another and their frequencies compared. The response of the PUF is equal to 1 if the first RO is faster and 0 otherwise.

The RO PUF in Figure 1, as not many details were given by the authors in [8], is realized using 32 ROs controlled by an enable signal for all selected technologies (ALTERA, XILINX, and ACTEL). Two counters are used to define the winner of the race by counting N periods of the two generated clock signals: if one of the two counters (the winner of the race) reaches a predefined value N, the arbiter stops the race and saves its result in the shift register. The principle of the race arbiter is depicted in Figure 2. Once one of counters reached the maximum value N, it sets its output signal "finished" to 1. This value causes that the "done" signal of the winning counter is also set to 1 and it blocks the "done" signal of the second counter, which cannot be set any more. This means that the race can have only one winner, pointed out by the OR gate (0 for counter 2, 1 for counter 1). Once the race result was obtained and saved, the oscillation and counter could restart using the same control signal (enable).

When compared with the original principle published in [8], the proposed principle is more precise; it can recognize differences that are smaller than 1-bit, so the counting period can be shorter and the PUF response faster. In our experiments, we used 10-bit counters and the most significant bit was used as the output signal (signal "finished") of the counter.

The output of the PUF presented in Figure 1 is 1-bit wide. In order to obtain a wider response, we use a shift register with a 16-bit output. To get more responses at once, the challenge generator is included in the design. It is a simple 8-bit counter-incremented after each race. For each value of the challenge generator, two different oscillators are chosen for comparison. They are separated in two groups of 16 (group A and group B). The output of the counter is divided in two parts: 4 Least Significant Bits (LSB) selecting one of 16 ROs in group A and 4 Most Significant Bits (MSB) selecting one of 16 ROs in group B. Thus, every oscillator in group A is compared to all oscillators in group B. This way, we obtain $16 \times 16 = 256$ different challenges thus 256 responses of 1 bit for each device. For simplicity, we consider that each IC delivers 256-bit responses. The generated bit streams are sent to the PC using a USB interface. For this reason, a small USB module featuring a Cypress EZ-USB device was connected to the evaluation board containing

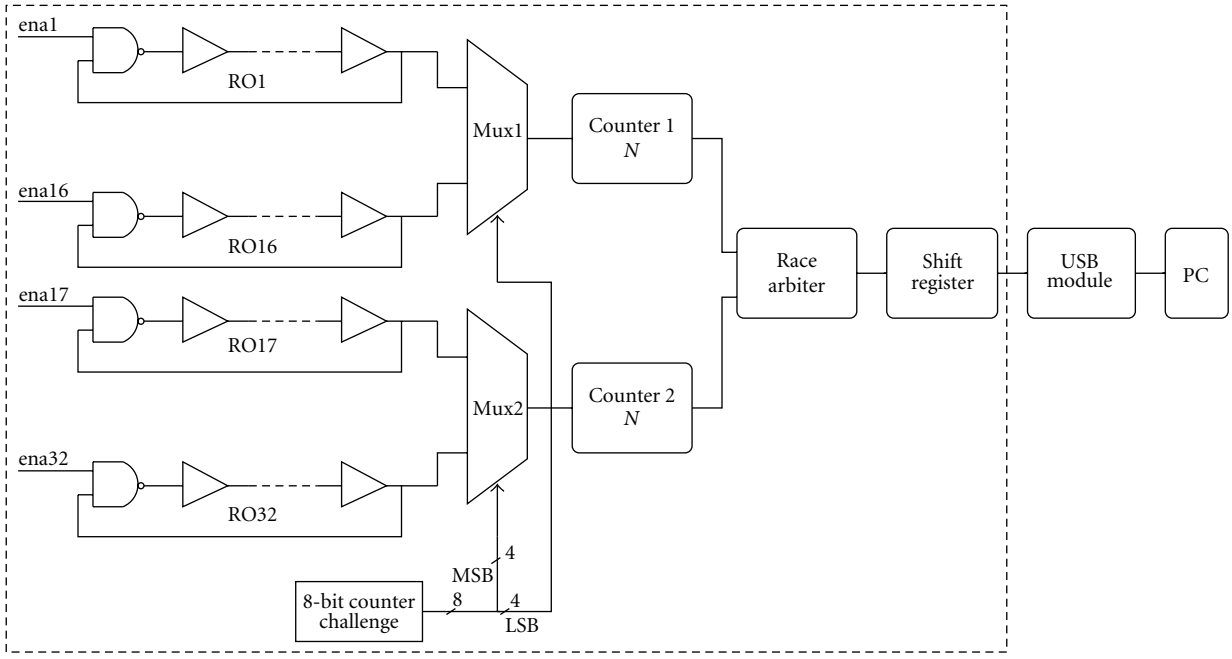

Figure 1: RO PUF Scheme.

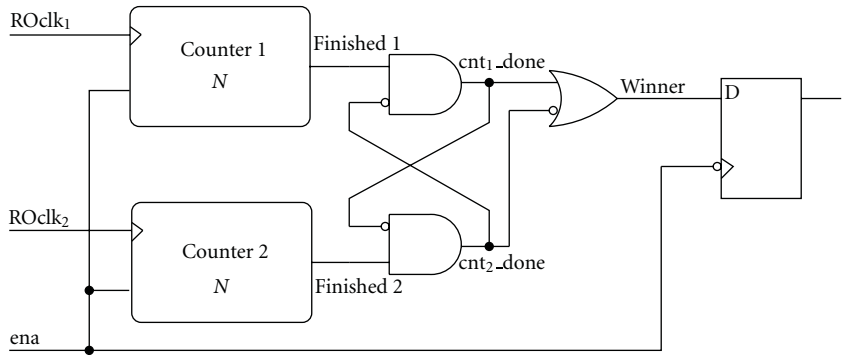

Figure 2: Principle of the race arbiter.

FPGA. A 16-bit communication interface with this module was implemented inside the FPGA. A Visual C++ application running on the PC reads the USB peripheral and writes data into a text file. For both ALTERA and XILINX technology, delay elements of the ROs are implemented using Look-Up Tables (LUTs). Finally, one NAND2 gate that is necessary to obtain oscillations, close the loop, and provide the structure with an enable signal. This configuration allows the use of either an odd or even number of delay elements. Thus, the ROs used in the design are made of 7 delay elements and one NAND2 gate in order to fit the ring into one LAB. In ACTEL technology, the oscillators employ 7 AND2 gates as delay elements and a NAND2 gate as a control gate.

3.2. Implementation Results and Tests. As resources in FPGAs have increased in volume and performances, integrated development environments (IDEs) are charged with automatic placement and routing. This is very convenient in common applications. The design can be translated into a significant number of logic elements, thus with automatic placement and routing, the user gains time and the surface of the IC is used at its maximum capacity. The compiler used by these environments calculates the optimal disposition of logic cells.

The RO PUF presented in [8] exploits, as any other PUF, the intrinsic characteristics of an IC. By definition, the position of the PUF on the IC determines the set of challenge-response pairs. RO placed in LAB A will most probably not oscillate at the same frequency as RO placed in LAB B. In order to use this PUF for the authentication process or as a secret key generator, one must be sure that the response of the PUF will be the same under any environmental conditions and even more important, after each reconfiguration of the device.

The first tests were conducted on ALTERA DE1 boards including Cyclone II EP2C20F484C7N FPGA. We used the PUF to authenticate the devices available in the laboratory. It allowed us to identify a given IC between the 13 available.

Table 1: RO PUF responses for three projects implemented in the same device (ROs were constrained but not the race arbiter).

Response no. 1 (only the PUF)	031f031f031f0005573f031f031f471f
	573f031f011fd73ff7bf431f0117031f
Response no. 2 (PUF and counter)	010000004df20000000045a04de245e0
	45a0fff70000000045e0fff745e045e0
Response no. 3 (only the PUF and an extra output)	0007600f7eef7eef200fffff724f704f
	600f0007600f7eef0005000772ef0007

Table 2: RO PUF responses for three projects implemented in the same device (ROs and race arbiter were constrained).

Response no. 1 (only the PUF)	ea09ebf9ea09ebfbea09e**b**59ebfbeb79
	ea09ebfbfffffbffeb79ea49ebfb0001
Response no. 2 (PUF and counter)	ea09ebf9ea09ebfbea09ea**6**9ebfbeb79
	ea09ebfbfffffbffeb79ea49ebfb0001
Response no. 3 (only the PUF and an extra output)	ea09ebf9ea09ebfbea09ea59ebfbeb79
	ea09ebfbfffffbffeb79ea**19f**bfb0001

As expected, we were able to perform this operation on all the devices: both inter- and intrachip variations were in normal ranges. However, authors insisted that the design needed no further placement and routing constraints in [8], page 3: "[· · ·] there is no need for careful layout and routing. For example, the paths from oscillator outputs to counters do not need to be symmetric." While it is clear that ROs must be identically laidout (which is achieved thanks to a macro in [8]), it is questionable that extra logic around the ROs (e.g., the race arbiter) needs no placement and routing constraints. We can imagine the next scenario: in order to control royalties, the IP vendor wants to use a PUF for linking the IP licence to a concrete FPGA device. However, a while later, he needs to make an upgrade of his IP function, still related to the same device and the same PUF response. This means that he needs that the response of the PUF block will be independent of the rest of the design.

We evaluated the impact of the changes in surrounding logic on the PUF response. First, we added a new counter-block module. Since it was independent of the PUF, it should not change the PUF response. Contrary to all expectations, instead of obtaining almost the same response of the PUF (i.e., obtaining small intrachip variation), the device gave completely different response so that presumably low intrachip variation after addition of the additional logic was almost as high as an ideal inter-chip variation: 48,8% of the response bits changed. We have to stress here, that the placement and routing of ROs was constrained, but not the structure of the arbiter.

In the next experiment, we kept initial PUF and sent its output, as before, towards the USB module and additionally towards a 7-segment display available on ALTERA DE1 board. While we were still maintaining the placement and routing for ROs, we got another (the third) response of the PUF for the same device! The responses for three projects including the same PUF in the same device are depicted in Table 1.

3.3. Imposing Placement and Routing Solution for Response Stabilization. The above-mentioned results show that the optimization performed by the compiler implies different placement and routing for the race arbiter after each recompilation.

As we need to have the minimal intrachip variation even for project upgrades, this is of extreme importance. Different placement of the race arbiter implies different PUF and different intrinsic characteristics that are explored. In both targeted processes (authentication and key generation) this is not acceptable. Therefore, imposing placement and routing constraints on the whole PUF block is mandatory in order to obtain a response independent from architecture modifications in a reconfigurable device.

Once placement and routing constraints were applied on both ROs and arbiter structure, the PUF provided excellent and expected responses. Table 2 presents results of three projects including the same PUF in the same device. Only few differences exist (printed in bold).

4. Bit Dependency in the PUF Response

In Section 3.1, we presented the way we selected pairs of oscillators to obtain a 256-bit response. We have to evaluate how many bits are independent in the PUF response. For example, consider 2 ROs a, b in group 1 and two other ROs c, d in group 2. The comparisons are (a, c), (a, d), (b, c), and (b, d), giving 4 possible bits in the response. But if $a > c$, $c > b$, $b > d$, then we can predict $a > d$, so there are only 3 bits of information instead of 4 in this example.

In the following, we propose to compute how many bits in the response are independent. This could help RO-PUF designers to select pairs of ROs.

4.1. Generalization. Let $(a_1, \ldots, a_n)$ be the first group of ROs and $(b_1, \ldots, b_n)$ be the second group of ROs. We use the relation $x > y$ when RO x is faster than RO y. In most of cases, ROs in group 1 (resp., in group 2) are not sorted thanks to the relation $>$. However it exists a permutation $\sigma \in \mathcal{S}_n$ (resp., $\sigma' \in \mathcal{S}_n$) such as $a_{\sigma(1)} > a_{\sigma(2)} > \cdots > a_{\sigma(n)}$ (resp., $b_{\sigma'(1)} > b_{\sigma'(2)} > \cdots > b_{\sigma'(n)}$).

We define the matrix of all comparison results between one RO in group 1 and one RO in group 2. Matrix rows are indexed by $(a_{\sigma(1)}, \ldots, a_{\sigma(n)})$ and matrix columns are indexed by $(b_{\sigma'(1)}, \ldots, b_{\sigma'(n)})$

$$\mathrm{Comp}_n = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{pmatrix}, \tag{3}$$

$$\text{where } c_{i,j} = \begin{cases} 0 & \text{when } a_{\sigma(i)} \leq b_{\sigma'(j)} \\ 1 & \text{when } a_{\sigma(i)} > b_{\sigma'(j)}. \end{cases}$$

Because $(a_{\sigma(i)})_i$ and $(b_{\sigma'(j)})_j$ are sorted, if $a_{\sigma(i)} \leq b_{\sigma'(j)}$, then for all $k \geq i$, $a_{\sigma(k)} \leq b_{\sigma'(j)}$ because $a_{\sigma(i)} > a_{\sigma(k)}$. In other words if $c_{i,j} = 0$ then for all $k \geq i$, $c_{k,j} = 0$.

Similarly, if $a_{\sigma(i)} > b_{\sigma'(j)}$, then for all $l \geq j$, $a_{\sigma(i)} > b_{\sigma'(l)}$ because $b_{\sigma(j)} > b_{\sigma(l)}$. In other words, if $c_{i,j} = 1$ then for each $l \geq j$, $c_{i,l} = 1$.

Among the 2^{n^2} possible matrices, only matrices with general term $c_{i,j}$ following the two rules,

(1) if $c_{i,j} = 0$ then for all $k \geq i$, $c_{k,j} = 0$,

(2) if $c_{i,j} = 1$ then for each $l \geq j$, $c_{i,l} = 1$,

can be obtained when comparing pairs of ROs. The others denote antagonist comparisons that cannot appear (e.g., $a > c$, $c > b$, $b > d$ and $a < d$ which is impossible).

4.2. Number of Possible Matrices. Let $M_{m,n}$ be the number of possible matrices $\mathcal{M}_{m,n}$ with m rows and n columns. By symmetry of the role played by $(a_i)_i$ and $(b_j)_j$ it is obvious that $M_{m,n} = M_{n,m}$.

By convention, we set for all $n \in \mathbb{N}$, $M_{0,n} = 1 = M_{n,0}$, thus we have the following recursive relation giving $M_{m,n}$ $(m \leq n)$:

$$M_{m,n} = \sum_{i=0}^{m} M_{i,n-i} \times M_{m-i,i}. \tag{4}$$

To prove this relation, we write the matrix with four blocks for i from 0 to m:

$$\mathcal{M}_{m,n} = \left(\begin{array}{c|c} A_{i,n-i} & B_{i,i} \\ \hline C_{m-i,n-i} & D_{m-i,i} \end{array}\right) \tag{5}$$

with block $B_{i,i}$ necessarily filled with 1s and block $C_{m-i,n-i}$ necessarily filled with 0 s. This can be explained by building first matrices.

For example, for $i = 0$ we have

$$\mathcal{M}_{m,n} = (C_{m,n}) = \begin{pmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{pmatrix}. \tag{6}$$

Then the only location to set a one in the matrix is the upper right corner. Indeed, if we set a 0 instead then the last column will be filled with 0 s according to rule (1). In this case, it is impossible to set a 1 elsewhere in a line of the matrix because according to rule (2), it would force a 1 in the last column in the same line which is impossible. For $i = 1$ we have

$$\mathcal{M}_{m,n} = \left(\begin{array}{ccc|c} & A_{1,n-1} & & 1 \\ \hline 0 & \dots & 0 & \\ \vdots & \ddots & \vdots & D_{m-1,1} \\ 0 & \dots & 0 & \end{array}\right) \tag{7}$$

with $M_{1,n-1}$ matrices for $A_{1,n-1}$ and $M_{m-1,1}$ for $D_{m-1,1}$ for a total of $M_{1,n-1} \times M_{m-1,1}$ possible matrices in this configuration. The next configuration is obtained with a 1 in the upper right corner of block $C_{m-1,n-1}$ and following rules (1) and (2) the 2×2 block $B_{2,2}$ is necessarily filled with 1 s. So the next configuration will be the study of all possible matrices of the shape

$$\mathcal{M}_{m,n} = \left(\begin{array}{ccc|cc} & A_{2,n-2} & & 1 & 1 \\ & & & 1 & 1 \\ \hline 0 & \dots & 0 & & \\ \vdots & \ddots & \vdots & D_{m-2,2} & \\ 0 & \dots & 0 & & \end{array}\right) \tag{8}$$

then for $i = 3$ possible matrices with a 1 in the upper right corner of block $C_{m-2,n-2}$, and so on until $i = m$ where the last possible matrice has the form:

$$\mathcal{M}_{m,n} = \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix}. \tag{9}$$

Thus for i from 0 to m we count all the possible matrices $A_{i,n-i}$ and $D_{m-i,i}$ following the two previously mentioned rules. For each i, there are $M_{i,n-i}$ possible matrices for $A_{i,n-i}$ and $M_{m-i,i}$ matrices for $D_{m-i,i}$. Thus, there are $M_{i,n-i} \times M_{m-i,i}$ possible matrices for a given i. The sum over i gives the formula in (4).

Using this formula in our context (two groups with the same number n of ROs), there are $M_{n,n}$ possible matrices. Then $\log_2(M_{n,n})$ gives the number of independent comparisons we can perform to get independent bits in the response.

For $n = 16$ ROs in each group, we got (only) $M_{16,16} = 601080390$ authorized matrices among the 2^{256} possible. Then only a mean of $\log_2(M_{16,16}) \approx 29$ comparisons leads to 29 independent bits in the PUF response (instead of 256). We can deduce that we should have $n = 135$ ROs in each group to get a PUF response with 256 independent bits.

4.3. Example. In the next example, we use the response in Table 2. Due to intrachip variation, some bits change in the response. To have only one representation of the response, we use a mean over 64 PUF responses with the same challenge and we obtain ea09ebf9ea09ebfbea09eb59ebfbeb79ea09ebf-bfffffb ffeb79ea49ebfb0001. If we analyze the response, we can see repetitive patterns (e.g., ea09, ebf9, etc.) meaning that there are dependent bits in the response.

We can be more precise and give, in this configuration, the number of bits that are independent in the PUF response.

We rewrite the PUF response column by column in a 16×16 matrix, with rows from top to bottom $a_1, \ldots, a_{16}$ and columns from left to right $b_1, \ldots, b_{16}$

$$\begin{pmatrix}
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
0&1&0&1&0&1&1&1&0&1&1&1&1&0&1&0\\
0&1&0&1&0&0&1&0&0&1&1&1&0&0&1&0\\
0&1&0&1&0&1&1&1&0&1&1&1&1&1&1&0\\
0&1&0&1&0&0&1&1&0&1&1&1&1&0&1&0\\
0&1&0&1&0&1&1&1&0&1&1&1&1&0&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0\\
0&0&0&1&0&0&1&0&0&1&1&1&0&0&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1
\end{pmatrix}. \tag{10}$$

Then we have to transform this matrix with respect to the two previous rules. We count the Hamming weight of each line. Then we permute lines to obtain the Hamming weight of line $a_{\sigma(i)}$ greater than or equal to the one of line $a_{\sigma(i+1)}$ for each i see Table 5.

We obtain

$$\begin{pmatrix}
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0\\
0&1&0&1&0&1&1&1&0&1&1&1&1&1&1&0\\
0&1&0&1&0&1&1&1&0&1&1&1&1&0&1&0\\
0&1&0&1&0&1&1&1&0&1&1&1&1&0&1&0\\
0&1&0&1&0&0&1&1&0&1&1&1&1&0&1&0\\
0&1&0&1&0&0&1&0&0&1&1&1&0&0&1&0\\
0&0&0&1&0&0&1&0&0&1&1&1&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0
\end{pmatrix}. \tag{11}$$

The same work is done with columns (Hamming weight of column $b_{\sigma'(j)}$ is less or equal than the one of column $b_{\sigma'(j+1)}$) see Table 6:

Finally, we obtain the following matrix:

$$\begin{pmatrix}
\boxed{1}&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
\boxed{0}&\boxed{1}&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
0&\boxed{1}&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
0&\boxed{1}&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
0&\boxed{1}&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
0&\boxed{1}&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
0&\boxed{1}&1&1&1&1&1&1&1&1&1&1&1&1&1&1\\
0&\boxed{0}&\boxed{0}&\boxed{0}&\boxed{0}&\boxed{1}&1&1&1&1&1&1&1&1&1&1\\
0&0&0&0&0&\boxed{0}&\boxed{1}&1&1&1&1&1&1&1&1&1\\
0&0&0&0&0&0&\boxed{1}&1&1&1&1&1&1&1&1&1\\
0&0&0&0&0&0&\boxed{0}&\boxed{1}&1&1&1&1&1&1&1&1\\
0&0&0&0&0&0&0&\boxed{0}&\boxed{0}&\boxed{1}&1&1&1&1&1&1\\
0&0&0&0&0&0&0&0&0&\boxed{0}&\boxed{1}&1&1&1&1&1\\
0&0&0&0&0&0&0&0&0&0&\boxed{0}&\boxed{0}&\boxed{0}&\boxed{0}&\boxed{1}&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&\boxed{1}&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&\boxed{0}&\boxed{1}
\end{pmatrix} \tag{12}$$

with $\sigma = \begin{pmatrix} 1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\ 16&1&2&3&5&7&13&10&8&12&11&9&15&4&14&6 \end{pmatrix}$ which is the permutation of rows and $\sigma' = \begin{pmatrix} 1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16\\ 16&1&3&5&9&14&6&8&13&2&4&7&10&15&12&11 \end{pmatrix}$ which is the permutation of columns. Thus, $a_{\sigma(1)} = a_{16} > b_{\sigma'(1)} = b_{16}$ (upper left corner in the matrix) gives a one. Then other comparisons between $a_{\sigma(1)}$ and $b_{\sigma'(j)}$ for $j > 1$ give no additional information because we know that they will give a 1. The next comparison is between $a_{\sigma(2)} = a_1$ and $b_{\sigma'(1)} = b_{16}$ which is a zero, meaning that $a_1 < b_{16}$. Then other comparisons between $b_{\sigma'(1)}$ and $a_{\sigma(i)}$ for $i > 1$ are useless because we know that they will give a 0. In this way, we can identify which comparisons are giving information (they are boxed in the matrix). This also gives indexes of ROs that have to be compared. In this special case, they are ($a_{\sigma(1)} = a_{16}$ and $b_{\sigma'(1)} = b_{16}$, $a_{\sigma(2)} = a_1$ and $b_{\sigma'(1)} = b_{16}$, $a_{\sigma(2)} = a_1$ and $b_{\sigma'(2)} = b_1$,...). In our case, we have 31 suitable comparisons and so 31 bits in the response that are independent (close to the theoretical mean of 29 computed).

Table 3: Intrachip variation for different devices in nominal conditions.

Device	Cyclone III	Spartan 3	Virtex 5	Fusion
Intrachip variation	0.92%	0.81%	3.38%	13.5%

Table 4: Locking of ROs depending on voltage.

Voltage (5)	Number of RO locked (over 16 ROs)
0.95–1.15	0
1.20–1.25	2
1.30	4
1.35–1.40	8

4.4. Comments on This Method. This method is useful to know how many bits are independent in the PUF response. In particular, when the PUF is used for cryptographic key generation, it indicates how many bits of entropy you can expect in the response.

However the response has still 256 bits (including dependencies) and can be used to compute inter-chip variation between many devices of the same family. Due to intrinsic parameters of each device, permutations of ring oscillators will be different from one device to another, giving different response and contributing to inter-chip variation.

For intrachip variation, it is different. Permutations of ring oscillators are related to the device and will obviously change from one device to another. The number of independent bits in the response and their positions will depend on each device and our method permits to know precisely how many independent bits are there and what are their positions. The intrachip variation must be computed on these bits.

The proposed method is used to analyze the PUF response and not to change it. It has been implemented in software in order to estimate the entropy of the generated sequence. A hardware implementation could be possible and useful for improving PUF response. This aspect was not studied in this paper.

5. Observation of the PUF in Various Technologies and Environmental Conditions

5.1. Observing the PUF in ALTERA, XILINX, and ACTEL Technologies. In order to compare different FPGA technologies, we would need a huge number of devices for all of tested families. Unfortunately, we had only cards with thirteen Altera Cyclone II and four Cyclone III devices, five Xilinx Spartan 3, three Xilinx Virtex 5 chips, and five Actel Fusion FPGAs at our disposal. For this reason, we used the biggest group of thirteen Cyclone II FPGAs to verify the inter-chip variation. We used results obtained in Section 4 (i.e., a PUF response of 31 bits in this case). The obtained value computed using (1) was 48.57% in average, which is close to the ideal value of 50%. The intrachip variation of the PUF was tested on four ALTERA Cyclone III EP3C25F256C8N ICs. These experiments were conducted under variable temperature and voltage conditions. Results have been prevailed for a temperature range from 30 to 80° Celsius (see Figure 3) and a voltage range from 0.9 to 1.3 V for the nominal voltage of 1.2 V (see Figure 4).

In these two figures, the distribution of the number of bits (x-axis) that changed between two different responses from the same PUF is shown as a histogram. The doted line presents the binomial distribution $\mathcal{B}(n, p)$, where $n = 31$ is the number of bits of the response using the methodology presented in Section 4 and $p = \overline{\text{Intra} - d_{\text{HD}}}(64, i)$ is the average intradie Hamming distance over 64 samples for the board tested.

Experiments show that intrachip variation increases when temperature increases. Furthermore, the behavior of the PUF drifts from the binomial distribution. This is probably caused by the influence of thermal noise which is more important as temperature increases and superposes a normal distribution on the binomial distribution.

The lowest intrachip variation is obtained in the normal operating conditions, both in voltage (1.2 V) and temperature (30° Celsius).

In comparison to our previous results in [15], intrachip variations were underestimated because the mean was computed on a 256-bit response ignoring dependency between bits. According to our method, we identify 31 bits of information in the response. Thus the intrachip variation must be computed on these bits. This explains why the ratio

$$\frac{\text{intrachip variation in [15]}}{\text{intrachip variation in this paper}} \approx \frac{256}{31}. \quad (13)$$

The PUF was also tested on XILINX Spartan 3 XC3S700ANn, on XILINX Virtex 5 XC5VLX30T, and on ACTEL Fusion M7AFS600 FPGA devices. Experimental results confirm the fact that placement constraints are mandatory. Intrachip variation for these devices are presented in Table 3.

For ACTEL technology, the tests were performed on ACTEL Fusion M7AFS600 FPGA. The intrachip variation reaches 13.5%! This technology presents the highest intrachip variation which is unexploitable for IC authentication. One of the reasons for which we think the intrachip variation is higher for these boards is the fact that they present more noise than the other ones. We observed a peak at 20 MHz in the core voltage spectrum, caused probably by some internal oscillator embedded in ACTEL FPGA. Similar peak was not detected in other technologies. These results show that the quality of this PUF is strongly related to the quality of the device and the board. In this precise case, the intrinsic characteristics of the IC are overwhelmed by the noise and the results are far from being ideal.

5.2. PUF and Mutual Relationship between Rings. While studying properties of ROs, we observed that ROs influence one another sometimes to an unexpected extent. If the ROs are identically laid out, their oscillating frequencies are almost the same. The differences are caused by the intrinsic characteristics of the IC as well as by the noise. If the frequencies are so close that the current peaks caused by rising and falling edges overlap, the ROs can lock and oscillate at the same frequency, either in phase or with a phase shift.

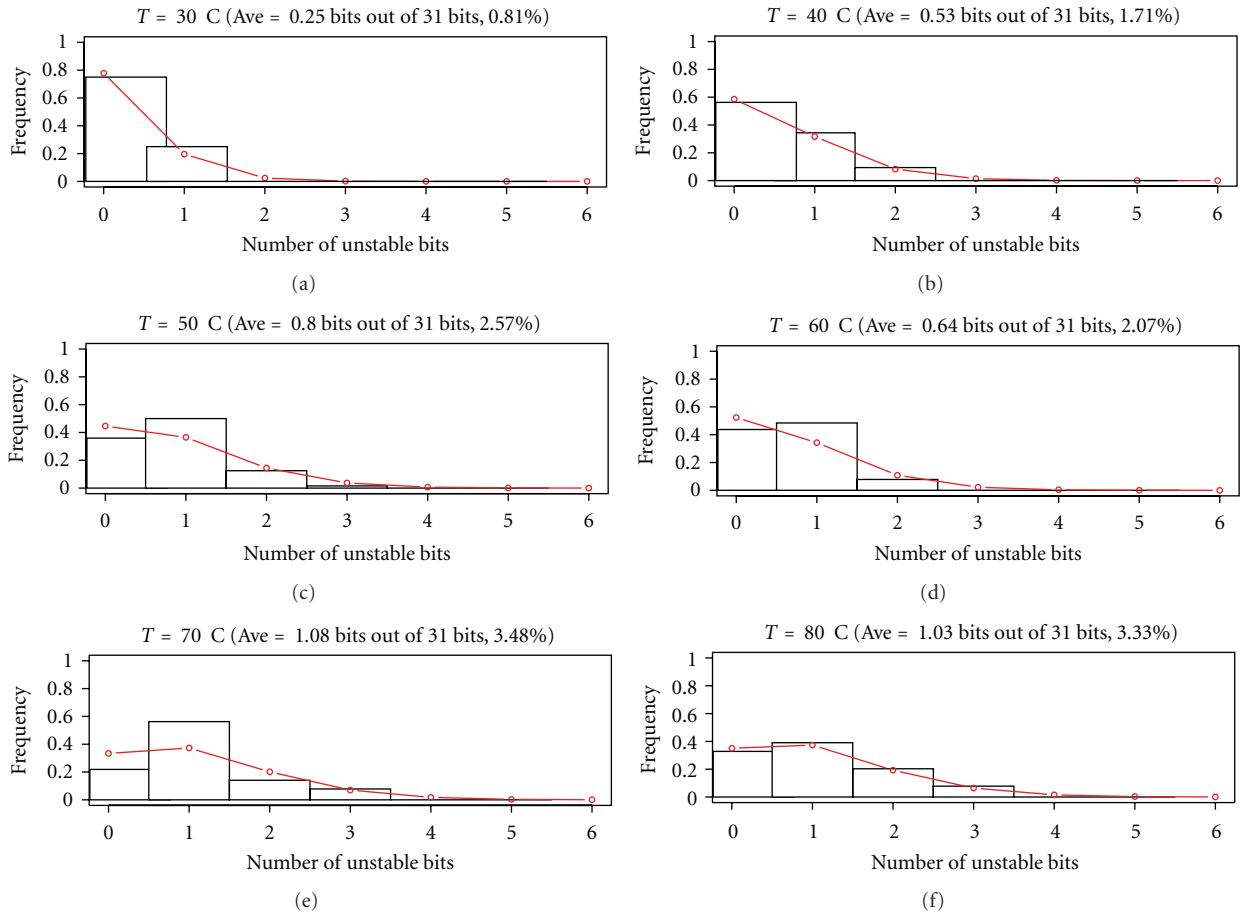

Figure 3: Intrachip variation on the same Cyclone III EP3C25F256C8N FPGA for various temperatures.

Table 5

Old line index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Number of 1	15	15	15	2	15	1	15	10	7	11	9	10	15	2	6	16
New index	2	3	4	14	5	16	6	9	12	8	11	10	7	15	13	1

Table 6

Column index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Number of 1	7	12	7	13	7	10	13	11	7	13	16	15	11	8	13	1
New index	2	10	3	11	4	7	12	8	5	13	16	15	9	6	14	1

Figure 5 shows output waveforms of two ROs that are locked (both waveforms are visible) and Figure 6 shows ROs that are not locked (the second waveform is not observable). Note that the oscilloscope was synchronized on the first waveform.

One can argue that the mutual dependence of rings could be caused by the FPGA input/output circuitry. In order to avoid influencing the results by outputting the signals from FPGA, we used simple circuitry permitting to detect the locking. The signals delivered by the two ROs were fed into the D flip-flop: one of them to the data input and the other to the clock input. If the output of the flip-flop is constant ("1" or "0") then the oscillators are locked.

The observation of numerous rings confirmed the fact that the mutual dependence of oscillators is big enough for them to lock and oscillate at the same frequency. We could also observe that independent oscillators at moment t_0 can become locked at moment t_1 if external conditions (temperature, voltage) present even slight changes.

If the challenge sent to the PUF selects a pair of oscillators that are locked, then the response is no longer based on intrinsic characteristics of the IC. Frequencies are identical, therefore the bit should not be valid. This depends however on the method employed for frequency measurement. In our design, if the ROs are locked with a phase shift, the rising edge of the RO with an advance will always be counted

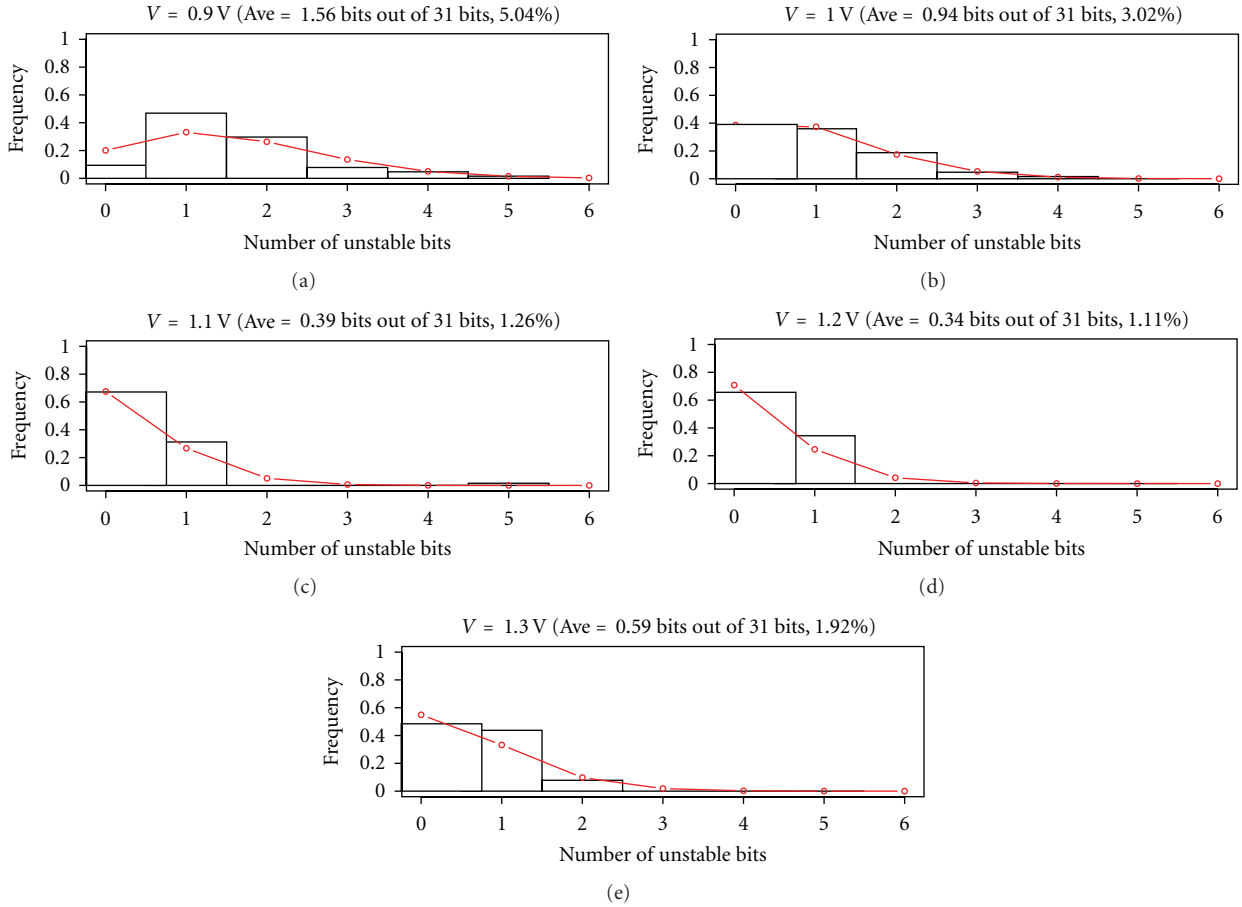

Figure 4: Intrachip variation on the same Cyclone III EP3C25F256C8N FPGA for various voltages.

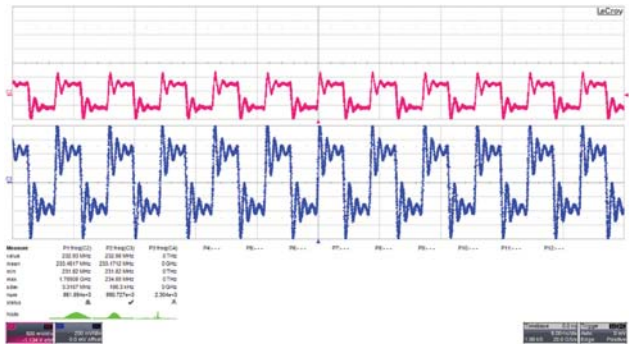

Figure 5: Locked ring oscillators. Trigger on top signal.

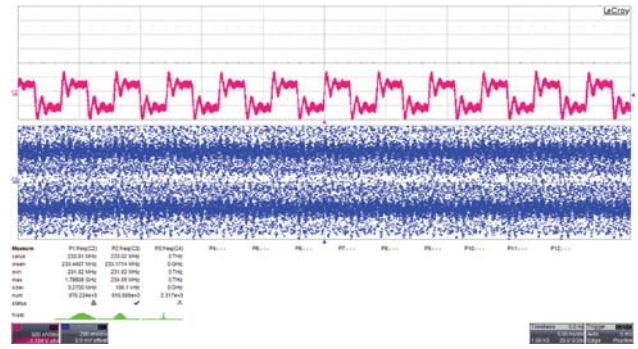

Figure 6: Unlocked ring oscillators. Trigger on top signal.

before the rising edge of the second RO. Thus, the result of the evaluation will always show that this RO has a greater oscillating frequency. If the oscillators are locked without a phase shift, the two counters will finish at the same time and the bit will be declared not valid.

This raises an important question on the quality of the response delivered by the PUF. If the oscillators are locked at the moment we compare their frequencies, the response is deterministic and no longer based on intrinsic characteristics of the device.

Identically laid-out oscillators request manual placement of the delay elements, as argued earlier. This means that the user will impose the position of the ROs on the device. Experimental results on the PUF showed that in certain configurations the distribution of "1"s and "0"s in the response was not uniform at all. In other configurations, the response presented a better distribution of values. Thus, we studied the locking phenomenon for ROs in certain configurations occupying the smallest area possible. These configurations were chosen because the surface of the PUF

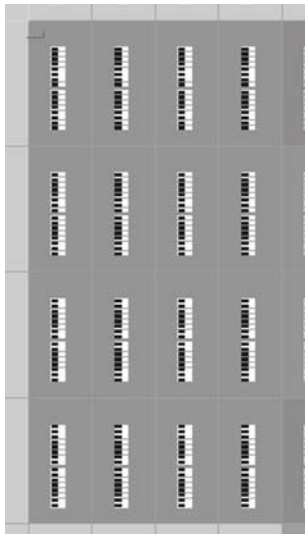

Figure 7: Configuration of ROs in a compact block.

should be relatively small comparing to the rest of the logic implemented in the device. Moreover, the PUF needs to be implemented in an isolated zone so that additional logic has only minimum influence on the response.

We tested two particular configurations, ROs grouped in a compact block, as presented in Figure 7 and ROs placed on two columns as presented in Figure 8. In this case one column represents one group of ROs.

Experimental results show that in the first configuration, there are more chances to have locked ROs. The most probable explanation for this phenomenon is that ROs placed close to each others are powered by the same wires. This fact has a great influence on the behavior of the oscillators. In Table 4 we present the influence of the voltage on the locking of the ROs on Cyclone III IC. Considering these experimental results, we cannot determine with precision the conditions under which ROs lock. We only observed that pairs of ROs can lock or unlock if environmental conditions change. Thus, questions rise on the reliability of the PUF and as manual placement is required, the configuration of the oscillators and the placement and routing of both ROs and arbiter must be carefully studied. In a recent publication by Maiti and Schaumont [13], the "Compensation method" used to select pairs of ROs in order to have good PUF properties indicate to choose ROs as close as possible from one to each others. Even if more investigation should be done on the locking of RO, such a method seems to be exposed to this phenomenon.

6. Further Enhancements of the PUF

Next, we propose some modifications of the PUF in order to enhance its characteristics.

6.1. Reduction of Intrachip Variations. As we observed in Session 5, changing environmental conditions (namely, voltage and temperature) increases the intrachip variation. This is due to the fact that identically laid-out ROs have very close oscillation frequencies. Since all ROs do not have exactly the same dependence on environmental conditions, some ROs can be more affected than others, and differences in frequencies can invert. While for laboratory temperatures the intrachip variation did not exceeded 4 bits, for temperature ranges from 30 to 80° Celsius, up to 15 bits out of 256 were unstable. For device authentication this is not a problem if the inter-chip variation remains high. For key generation this fact is not acceptable. We must guarantee the uniqueness of the key generated by the device. Therefore, as proposed by Suh and Devadas, an error correcting code can be used to correct errors due to the intrachip variation.

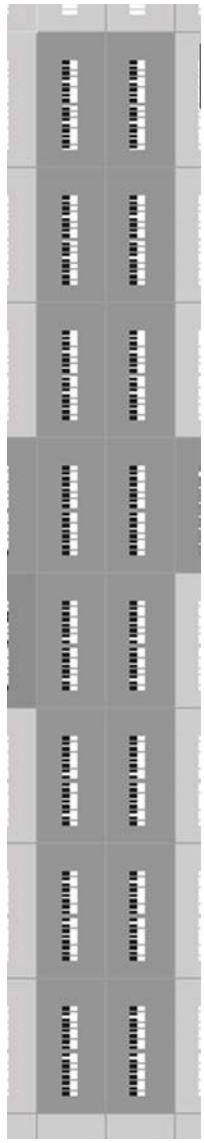

Figure 8: Configuration of ROs in 2 columns.

As usual, the response should not be used directly as a key even after correcting the errors. On one hand, there are weak

and periodic patterns in the response. On the other hand, after this error correcting process, the entropy of the "key" will probably be reduced. A hash function (e.g., Whirlpool [16] based on a modified AES) can be used to remove weak patterns but unfortunately it cannot solve the problem of entropy reduction.

6.2. Reduction of the Power Consumption. When dealing with the power consumption of the PUF, we used small dedicated modules made in our laboratory featuring ALTERA Cyclone III EP3C25F256C8N FPGA. We measured the static current consumption of the module and obtained 4 mA. Then we measured the consumption of the PUF using the 32 ROs and a PLL delivering the clock signal. The module consumed 24.7 mA, which is indeed considerable for a background function such as PUF. However, the PUF employs each time only two out of N ROs in order to obtain one bit of the response. Thus, we propose to stop all the $N - 2$ oscillators (30 in our case) that are currently not used for the response bit. The ROs are enabled and stopped using the enable input of the structure (Figure 1). When only two ROs were running, we measured a 13.4 mA current consumption. This is a reduction of consumption by approximately 51%.

In the next paragraph, we estimate the approximate power reduction that can be obtained in the design proposed in [8]. The total power consumption represents the sum of the static consumption (S), the consumption of the logic which is independent of the number of ROs (i.e., PLL, counters, comparators, etc.) (L) and the consumption of the logic which depends almost linearly on the number of ROs (multiplexers and ROs) denoted by $R(N) = \lambda \times N$ where N is the number of ROs and λ a constant float. We can make a simple calculus and show that the improved model would probably reduce considerably the consumption of the board.

In [8], Suh and Devadas used 1024 ROs in their design. Then, if we shut down unused ROs for each comparison, we should obtain a consumption of approximately $S + L + \lambda \times 2 = 13.4$ mA instead of $S + L + \lambda \times 1024 = 397.6$ mA. With our improved PUF control, we obtain a current consumption reduction of $1 - 13.4/397.6 = 96.6\%$. The PUF's power consumption thus becomes much more suitable for practical implementations.

Such an idea for reducing self-heating has been proposed in [17] but was not considered by Suh and Devadas in their design. However, in this paper only one ring is selected at a time. This idea cannot be used in our proposal to save more power consumption. If we want to use only one RO, we have to count its number of raising edges during one enable time, record this number, and repeat this measurement by selecting another RO during the same enable time. The main problem in such a case is the influence of the global deterministic part of the jitter on the frequency of one-ring oscillator [18]. This influence will not be the same from one measurement to another. Thus the comparison between the number of raising edges of two ROs will be suitable only if they are influenced by the same global deterministic part of the jitter in the same time.

7. Conclusions

The concept introduced in [8] is very simple, with a differential structure that presents an excellent behavior as long as the IC is not reconfigured. As this structure (the PUF) is useless if implemented alone in an IC, we analyze the influence of additional logic upon the response of the PUF. Our work proves that placement and routing constraints are required in order to maintain the quality of the PUF in FPGAs. Without any constraints, additional logic creates a completely different PUF and implicitly a completely different response. Instead of a small and acceptable intrachip variation after the IC reconfiguration, we obtained the variation 48.8% that was comparable in size to an ideal inter-chip variation (50%).

We also showed that bits in the response are dependent and propose a method to select pairs of ROs to have independent bits. Unfortunately this shortens the PUF response. The huge current consumption obtained by Suh and Devadas was also of our concern. We improved the design in order to considerably reduce the consumption. For a small PUF, (e.g., the one described in our experimental conditions with 32 ROs) the consumption was reduced by 51%. For a greater PUF, our improvement leads to an even more important reduction: we reduced the consumption of the PUF described in [8] by 96.6%.

Moreover, we showed that there are other phenomena that influence and jeopardize the integrity of the PUF. We argued why the "locking" phenomenon is affecting the response of the PUF and it is very important to notice that not all challenges can be used at any moment. Apart from the locking, our experimental results show that noisy motherboards can increase the intrachip variation for the PUF.

Acknowledgments

The work presented in this paper was realized in the frame of the SecReSoC Project n. ANR-09-SEGI-013, supported by the French National Research Agency (ANR). The work was partially supported also by the Rhones-Alpes Region and Saint-Etienne Metropole, France. The authors would like to thank Malick Boureima and Nathalie Bochard for their help with numerous experiments.

References

[1] S. Drimer, "Volatile FPGA design security–a survey," in *IEEE Computer Society Annual Volume*, pp. 292–297, 2008.

[2] R. Anderson and M. Kuhn, "Low cost attacks on tamper resistant devices," in *Security Protocols*, pp. 125–136, Springer, 1998.

[3] R. Anderson and M. Kuhn, "Tamper resistance: a cautionary note," in *2nd USENIX Workshop on Electronic Commerce*, vol. 2, 1996.

[4] S.P. Skorobogatov, "Semi-invasive attacks-a new approach to hardware security analysis," Tech. Rep., University of Cambridge, Computer Laboratory, 2005.

[5] J. A. Halderman, S. D. Schoen, N. Heninger et al., "Lest we remember: cold boot attacks on encryption keys," in *USENIX Security Symposium*, P.C. van Oorschot, Ed., pp. 45–60, 2008.

[6] R. Pappu, *Physical one-way functions*, Ph.D. thesis, Massachusetts Institute of Technology, 2001.

[7] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, no. 5589, pp. 2026–2030, 2002.

[8] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *44th ACM/IEEE Design Automation Conference (DAC '07)*, pp. 9–14, June 2007.

[9] J. Guajardo, S. Kumar, G.J. Schrijen, and P. Tuyls, "FPGA intrinsic PUFs and their use for IP protection," in *Cryptographic Hardware and Embedded Systems (CHES '07)*, pp. 63–80, 2007.

[10] S. Kumar, J. Guajardo, R. Maes, G. J. Schrijen, and P. Tuyls, "Extended abstract: the butterfly PUF protecting IP on every FPGA," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST '08)*, pp. 67–70, 2008.

[11] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 148–160, November 2002.

[12] D. Lim, J. W. Lee, B. Gassend, G. E. Suh, M. Van Dijk, and S. Devadas, "Extracting secret keys from integrated circuits," *IEEE Transactions on Very Large Scale Integration*, vol. 13, no. 10, pp. 1200–1205, 2005.

[13] A. Maiti and P. Schaumont, "Improved ring oscillator PUF: an FPGA-friendly secure primitive," *Journal of Cryptology*, pp. 1–23, 2010.

[14] Y. Hori, T. Yoshida, T. Katashita, and A. Satoh, "Quantitative and statistical performance evaluation of arbiter physical unclonable functions on FPGAs," in *International Conference on Reconfigurable Computing and FPGAs*, pp. 298–303, 2010.

[15] C. Costea, F. Bernard, V. Fischer, and R. Fouquet, "Analysis and enhancement of ring oscillators based physical unclonable functions in FPGAs," in *International Conference on Reconfigurable Computing and FPGAs*, pp. 262–267, 2010.

[16] P. Barreto and V. Rijmen, "The Whirlpool hashing function," in *1st Open NESSIE Workshop*, vol. 13, p. 14, Leuven, Belgium, 2000.

[17] P. Sedcole and P. Y. K. Cheung, "Within-die delay variability in 90nm FPGAs and beyond," in *IEEE International Conference on Field Programmable Technology (FPT '06)*, pp. 97–104, December 2006.

[18] V. Fischer, F. Bernard, N. Bochard, and M. Varchola, "Enhancing security of ring oscillator-based RNG implemented in FPGA," in *Proceedings of the Field- Programable Logic and Applications (FPL '08)*, pp. 245–250, September 2008.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 360242, 9 pages
doi:10.1155/2012/360242

Research Article

Blind Cartography for Side Channel Attacks: Cross-Correlation Cartography

Laurent Sauvage, Sylvain Guilley, Florent Flament, Jean-Luc Danger, and Yves Mathieu

Télécom ParisTech, Institut Télécom CNRS LTCI, 46 rue Barrault, F-75634 Paris Cedex 13, France

Correspondence should be addressed to Laurent Sauvage, laurent.sauvage@telecom-paristech.fr

Received 12 July 2011; Revised 20 October 2011; Accepted 27 December 2011

Academic Editor: Kris Gaj

Side channel and fault injection attacks are major threats to cryptographic applications of embedded systems. Best performances for these attacks are achieved by focusing sensors or injectors on the sensible parts of the application, by means of dedicated methods to localise them. Few methods have been proposed in the past, and all of them aim at pinpointing the cryptoprocessor. However it could be interesting to exploit the activity of other parts of the application, in order to increase the attack's efficiency or to bypass its countermeasures. In this paper, we present a localisation method based on cross-correlation, which issues a list of areas of interest within the attacked device. It realizes an exhaustive analysis, since it may localise any module of the device, and not only those which perform cryptographic operations. Moreover, it also does not require a preliminary knowledge about the implementation, whereas some previous cartography methods require that the attacker could choose the cryptoprocessor inputs, which is not always possible. The method is experimentally validated using observations of the electromagnetic near field distribution over a Xilinx Virtex 5 FPGA. The matching between areas of interest and the application layout in the FPGA floorplan is confirmed by correlation analysis.

1. Introduction

Side channel attacks (SCA) and fault injection attacks (FIA) are very efficient techniques to retrieve secret data stored in cryptographic devices such as smartcards. First attacks have been performed globally, for instance by measuring the power consumption (power analysis: PA) [1] of a device under analysis (DUA) or by quickly changing the nominal voltage of its power supply [2]. But best results have then been achieved locally, by using a small EM probe just over the cryptoprocessor (electromagnetic analysis: EMA) [3] or by shooting at it with a laser beam [4, 5]. Indeed, for SCA, such locality permits to solely collect the activity of the cryptoprocessor, instead of gathering the activity of the whole DUA. In the case of FIA, depending on the technology process of the integrated circuit, only one bit of the implementation can be affected. However, the efficiency of these attacks relies on localisation methods which have to pinpoint as accurately as possible the DUA-sensitive areas. Using these localisation methods is mandatory in the case of a cryptographic application embedded in a field Programmable gate array (FPGA) as its regular structure prevents the localisation of sensible modules by optical or electron microscopy. Indeed, the task is easier for most ASICs, where the functional modules stand clearly out from a visual inspection of the layout as rectangular shapes. Some methods have been proposed in the past, illustrated using as observations the near electromagnetic (EM) field radiated by the DUA: an EM probe is moved over the DUA from a position to another one, and for each of them, the temporal variation of the EM field is measured with an oscilloscope. We use once again such cartography procedure in this paper. Furthermore, we note that all previously published localisation methods along with the one in this paper can deal with other physical phenomenons, such as photons emitted by transistors while they commute [6, 7].

Up to now, two strategies have been deployed to locate cryptographic modules within a DUA. They consist in iden-

tifying areas where the physical observations vary according to:

(1) the data processed during an encryption [8, 9] or

(2) the operations performed by the cryptographic module [10], even if this latter is protected against SCA [11].

With the first strategy, two observations are collected for two different plaintexts p_1 and p_2, and their fluctuations are assessed either by looking for the maximum difference in their temporal domain [8] or by calculating the incoherence of the frequency spectrum [9]. The larger the difference is or the lower the incoherence is, the closer to the cryptoprocessor the EM probe is. To improve the accuracy of the method, a third observation but of the same plaintext p_1 can be acquired, with a view to reject the measurement noise [8]. These approaches seem to be the most suitable because statistical tools which are then used, the CPA [12] for example, exploit plaintexts or ciphertexts as well. However, they would be optimal only if and only if the differences in the observations are maximal, which requires that all of the transistors making the datapath of the cryptoprocessor commute. This can happen only if the attacker knows the secret data, obviously the key but also the mask values when the DUA is protected by masking [13, 14], which is possible in the framework of an evaluation but not with a real-world application.

Instead of focusing on the same time slot, that of the encryption, and collecting two observations, the techniques of the second strategy need only one plaintext and take advantage of two or more time slots of a single observation. Typically, if the cryptoprocessor is in idle state, none of its transistors commute and the corresponding activity is at a low level. On the contrary, during an encryption, the activity is expected to be at a high level. Thus, the localisation of the cryptoprocessor can be achieved in the temporal domain by evaluating the difference between these activities [10]. To diminish the impact of the measurement noise, the localisation can be performed in the frequency domain: indeed, the succession of these low and high activity levels yields a special signature in the frequency spectrum [10]. As this succession still occurs for some countermeasures, the second strategy remains valid to localise protected cryptoprocessors. This fact is all the more true with protections using dual-rail with precharge logic (DPL [15]): as they alternate between two phases, namely, precharge and evaluation phases, they "oscillate" at the half of the master clock frequency, a frequential component of great interest for the frequency analysis [11].

Both previous strategies require to identify the time slot of some sensible operations, such as the encryption. This information can be extracted from the implementation netlist or by using simple power analysis (SPA) [1] or simple electromagnetic analysis (SEMA). Nonetheless, exploring the full implementation appears to be complex, at least time consuming and forces it to be partial, focused on few targets. In this paper, we propose a method to *exhaustively* locate the information sources of a DUA, *without preliminary knowledge* about it. This method is described in Section 2. Then, its ability to identify areas of interest, and in particular cryptographic modules, is evaluated in Section 3. Finally, Section 4 concludes this paper and presents some future works and perspectives.

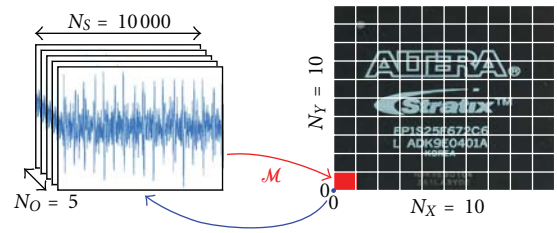

Figure 1: Cartography principle commonly used.

2. Cross-Correlation Cartography

Cartography formally consists in monitoring one or more physical phenomenons at N_P positions over a DUA. Generally, these positions form a 2D grid composed of N_X and N_Y points over, respectively, the X and Y axes. N_X and N_Y could have the same value, for instance 10 as in Figure 1. For each of the $N_P = N_X \times N_Y$ positions, identified by their coordinates (x, y), N_O observations $O^0_{(x,y)}$ of N_S samples n are achieved. They constitute an observations set $\mathcal{O}_{(x,y)} = \{O^0_{(x,y)}, O^1_{(x,y)}, \ldots, O^{N_O-1}_{(x,y)}\}$. In the example of Figures 1 and 4 observations of 10,000 samples are collected per position. To build the final 2D map, each set $\mathcal{O}_{(x,y)}$ has to be reduced to a unique scalar $m_{(x,y)}$. The common usage is to apply a function $\mathcal{M}$ to the corresponding observations $O^0_{(x,y)}, O^1_{(x,y)}, \ldots, O^{N_O-1}_{(x,y)}$. From a "graphical" standpoint, the $m_{(x,y)}$ values, real numbers, are then mapped to colors according to a user-defined scale.

The localisation method we propose in this paper is motivated by the fact that the observation of a physical phenomenon depends on the time and on the space. In the SCA topic, the physical phenomenon we consider is the emanation of a digital integrated circuit. As the state of this circuit changes from a synchronization clock cycle to another one, the observations are made of successive peaks. The amplitude of each one of these peaks

(i) varies in the time according to the data manipulated by its source (see for instance Figure 7);

(ii) decreases when the distance between the source and the observation point increases.

In consequence, sources carrying distinct information generate physical phenomenons whose temporal variations look completely different, that is, *uncorrelated*. At the opposite, observations gathered at positions close to each other, and in particular at positions which are themselves close to a source, look very alike. Thus, to locate these sources, we collect a single observation $O^0_{(x,y)}$ per (x, y) position, then estimate the similarity level between all of these observations. While the methods presented in Section 1 consider (through the $\mathcal{M}$ function, see Figure 1) each observations set $\mathcal{O}_{(x,y)}$ independently from the other, we use conjointly all of them.

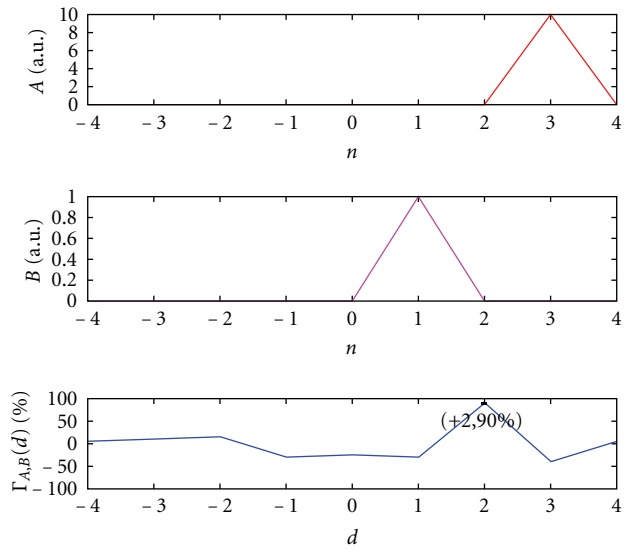

Figure 2: From top to bottom: one reference signal A, the same B but with a lower amplitude and delayed by two samples, and their normalized cross-correlation function $\Gamma_{A,B}(d)$.

The first step of our technique consists in taking each observation $O^0_{(x,y)}$ as a reference, then looking for the maximum of normalized cross-correlation (abridged NXC) between this reference and the other observations. The NXC function is defined as

$$\Gamma_{A,B}(d) = \frac{\operatorname{cov}(A, B_d)}{\sigma_A \cdot \sigma_B} = \frac{\sum_{n=d}^{d+\inf(N_A,N_B)-1} \left(A(n) - \overline{A(n)}\right) \cdot \left(B(n-d) - \overline{B(n)}\right)}{\sqrt{\sum_{n=0}^{N_A-1} \left(A(n) - \overline{A(n)}\right)^2} \cdot \sqrt{\sum_{n=0}^{N_B-1} \left(B(n-d) - \overline{B(n)}\right)^2}}. \quad (1)$$

In (1), $\operatorname{cov}(\cdot,\cdot)$ stands for the covariance, A and B are two observations (at two different points), which, respectively, have N_A and N_B temporal samples, whose mean values are $\overline{A(n)}$ and $\overline{B(n)}$, and whose standard deviations are σ_A and σ_B. B_d means that a delay d belonging to the interval from $-(N_B - 1)$ to $N_A - 1$ is applied to B. From a "graphical" standpoint, the waveform of B is shifted to the right along the temporal X-axis, or in other words the origin $n = 0$ of B is moved to $n = m$. We simply abridge A_0 as A and note that $\sigma_{B_d} = \sigma_B$ because the standard deviation is considered over the complete waveform (and is thus independent of the offset d). Figure 2 shows the variations of the NXC function $\Gamma_{A,B}(d)$ according to the d values, when A and B are two signals with an identical shape, but distinct amplitudes, are delayed by two samples. The maximum value of $\Gamma_{A,B}(d)$ indicates that A and B are 90% similar, while the index of this maximum, 2, provides the sample delay between A and B. The value of $\Gamma_{A,B}(d)$ does not reach 100% because of computational side-effects: the X-axis is not infinite but bounds to $[-4; +4]$.

To briefly illustrate the result of such computation, we provide two NXC maps on Figure 3. The first one, (a), has been obtained considering the center of the map as the reference point. Maximum correlation values are at a very low level, lightly positive in red on the map, or negative in blue, except for the reference point, which by definition is 100% correlated to itself and takes the yellow point in the center. It does not identify a source of interest, because it is insulated. On the second map, (b) of Figure 3, an area of high correlation (in yellow) stands out around the reference point located at $X = -7.90$ mm and $Y = 12.0$ mm and is marked with a white cross. This zone has a size greater than the actual active logic in the FPGA and notably extends outside the silicon chip's boundary, depicted in a white dashed line. The diffusion of the EM field accounts for this extension of a couple of millimeters around the radiating logic. Indeed, in our setup, the distance between the loop sensor and the silicon surface is roughly speaking equal to 2 mm. Above, a second area emerges in blue, with a negative correlation value, as the observations correspond to EM field measurements, these ones, and in turn the correlation values, may be of opposite sign. But in reality, this second blue area contains the same information as the first yellow area does. Therefore, to prevent such artifact, we now consider the *absolute* maximum values of the normalized cross-correlation function.

Each observation gathered at a position of the 2D grid become in turn a reference observation, we finally collect N_P NXC maps. Most of them are alike, as computed at neighbouring points, close to physical sources. The second step of our technique aims at grouping them. For this purpose, we need once again a correlation estimator, but this time, as we manipulate maps, this one should take into account the two dimensions x and y. This bidimensional estimator, namely, $\Gamma^{2D}_{M,N}(p,q)$, is defined as:

$$\Gamma^{2D}_{M,N}(p,q) = \frac{\operatorname{cov}\left(M, N_{(p,q)}\right)}{\sigma_M \cdot \sigma_N}, \quad (2)$$

where

$$\operatorname{cov}\left(M, N_{(p,q)}\right) = \sum_{x=p}^{x_{\max}} \sum_{y=q}^{y_{\max}} \left(M(x,y) - \overline{M(x,y)}\right) \cdot \left(N(x-p, y-q) - \overline{N(x,y)}\right),$$

$$\text{with } x_{\max} = p + \inf\left(N_{X_M}, N_{X_N}\right) - 1$$

$$\text{and } y_{\max} = q + \inf\left(N_{Y_M}, N_{Y_N}\right) - 1,$$

$$\sigma_M = \sqrt{\sum_{x=0}^{N_{X_M}} \sum_{y=0}^{N_{Y_M}} \left(M(x,y) - \overline{M(x,y)}\right)^2},$$

$$\sigma_N = \sqrt{\sum_{x=0}^{N_{X_N}} \sum_{y=0}^{N_{Y_N}} \left(N(x-p, y-q) - \overline{N(x,y)}\right)^2}. \quad (3)$$

In this equation, M and N are two maps, of N_{X_M} and N_{X_N} points on x, N_{Y_M} and N_{Y_N} points on y, whose mean values

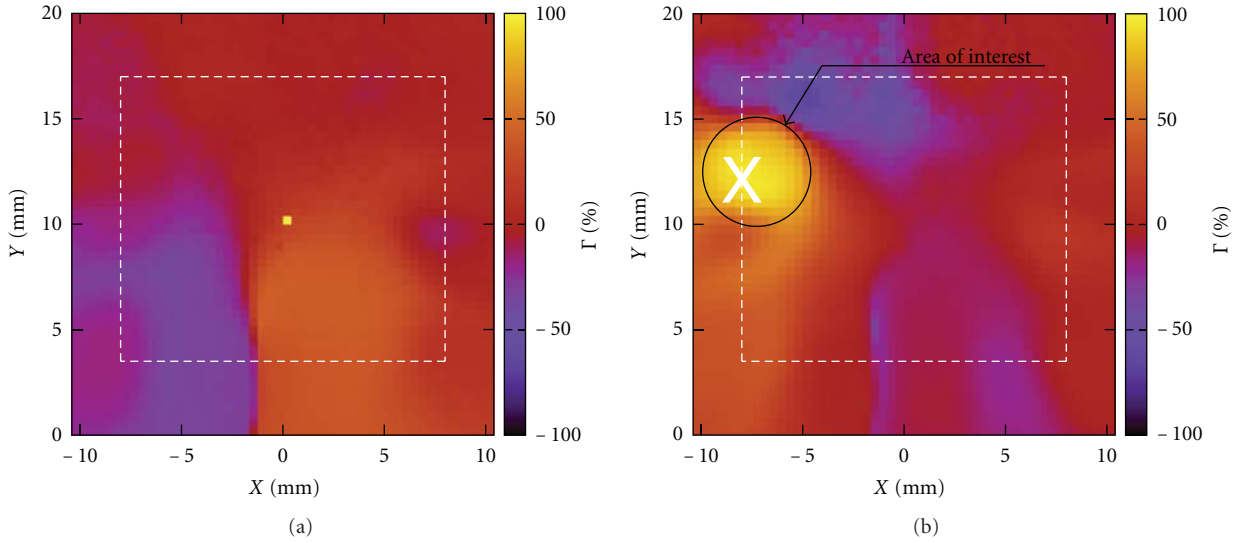

Figure 3: Normalized cross-correlation maps obtained when the reference point, marked with a white cross, is useless (a) or of interest (b).

are $\overline{M(x,y)}$ and $\overline{N(x,y)}$ and standard deviations σ_M and σ_N. $N_{(p,q)}$ means that a spacial offset is applied to the map N, so that its origin point $(0,0)$ is moved to (p,q). This offset is useful when the objective is to find the location of a small pattern within a reference map. In this paper, as we compare maps with identical size, p and q are set to zero. As previously, we fix a reference map, then we look for the maximum of the absolute value of $\Gamma^{2D}_{M,N}(p,q)$. If this maximum is greater than a user-defined threshold, maps are considered as identical and grouped in the same list. Every list is called an area of interest.

To finish the analysis, one map per list has to be extracted. It could be randomly chosen, but we suggest to select the map for which the number of points with a value above under, respectively, a user-defined threshold is the greatest. The corresponding map is the one with the widest, nearest area. The full method is summarized by the Algorithm 1. In this algorithm, the selection of areas of interest is represented by a function called "*extract*."

3. Experimental Results

To evaluate the efficiency of our method, we have used it against an FPGA-based cryptoprocessor performing the simple and triple Data Encryption Standard (3DES) [16], and protected by first-order Boolean masking [13, 14]. In practice, we have implemented the same masking scheme as in [17]. We concur this design is obsolete for at least two reasons. First of all, DES has been replaced by the Advanced Encryption Standard (AES) [18] since the year 2001. Second, the employed masking scheme is not robust against High-Order Side Channel Attack (HO-SCA) [17, 19]. Nonetheless, the objective of this section is not to come up with a new attack to break a still considered invulnerable countermeasure, but to experimentally prove that our method identifies areas of interest, and in particular two sensible 64-bit registers, LR and MASK, carrying respectively the masked value and the mask itself.

To make the experiment easier, we have constrained their placement so that they may fit in rectangular areas, themselves placed at the opposite sides of the FPGA. As depicted by Figure 4, MASK is at the top left hand corner, while LR is at the bottom right hand corner. Splitting these registers in such a way has spread the routing of the 3DES cryptoprocessor datapath all over the FPGA. Therefore, in a view to keep the other components of our implementation visible, for the 3DES datapath, only its logic cells are displayed. They appear as black dots in the upper half part of the floorplan. The KEY scheduling block is at the bottom right of Figure 4, in salmon, while the 3DES CONTROLler, in green, is in the middle, on the left. Close to this latter, we find a 6502 CISC CPU in olive and an UART in turquoise. All previous components share a VCI bus along with its memories, in gold in Figure 4. This real-life application has been programmed into a Xilinx [20] Virtex 5 FPGA, whose metallic lid has been removed with a cutter, as shown in Figure 5. This way, not only we can reduce the analysis area strictly to the FPGA silicon die, but the signal to noise ratio is also greatly improved.

Observations have been acquired using a 2 mm diameter EM probe, a 3 GHz bandwidth 30 dB gain preamplifier, and an Agilent [21] Infiniium 54854 oscilloscope, whose bandwidth and sampling rate have been set up to, respectively, 3 GHz and 10 GSa/s. The EM probe has been moved following a 25×25 points grid, per step of 480 μm along the X-axis, and 400 μm along the Y-axis. The grid is rather rectangular, since we covered the whole silicon die of the Virtex 5 (refer to Figure 5): 12 mm wide and 10 mm high. Then, maps have been grouped together according to a threshold of 90%, that is, two maps whose 2D cross-correlation coefficient is greater than 90% are considered as identical and gathered in the same list. Finally, we have counted for each map the number of points with a correlation level above 90%. From each list,

Require: One observation per point
Ensure: List of identical maps
  For each 2D grid point (x, y) **do** {Fixed reference point A}
    {/ Looping over all fixed reference points A /}
    $x\,ref \leftarrow x$
    $y\,ref \leftarrow y$
    for each 2D grid point (x, y) **do**
      {/ Looping over all mobile points B /}
      $m(x, y) \leftarrow \max_d(|\Gamma_{O(x\,ref, y\,ref), O(x,y)}(d)|)$
    end for
    $map(x\,ref, y\,ref) \leftarrow m$
  end for
  $i \leftarrow 0$ {/ Index of areas of interest /}
  for each 2D grid point (x, y) **do**
    {/ Looping over all fixed reference points A /}
    $x\,ref \leftarrow x$
    $y\,ref \leftarrow y$
    $list[i] \leftarrow map(x\,ref, y\,ref)$
    for each 2D grid point (x, y) **do**
      {/ Looping over all mobile points B /}
      $M_r \leftarrow map(x\,ref, y\,ref)$
      $M_c \leftarrow map(x, y)$
      if M_c / *list* and $\max(|\Gamma^{2D}_{M_r, M_c}(0, 0)|) > threshold$ **then**
        $list[i] \leftarrow map(x, y)$
      end if
    end for
    $i \leftarrow i + 1$
  end for
  for $j = 0$ to i **do**
    $area(j) \leftarrow extract\ (list[j])$
  end for

Algorithm 1: Algorithm for grouping maps per area of interest.

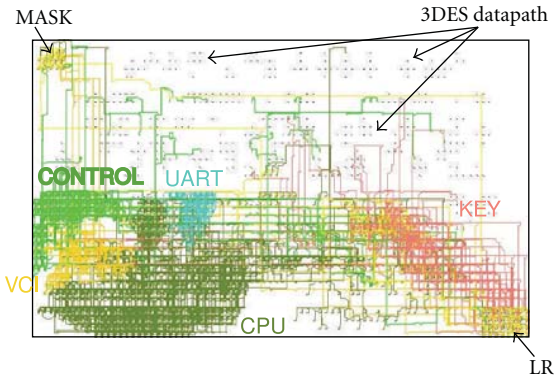

Figure 4: Floorplan of the cryptographic application under ISE floorplanner.

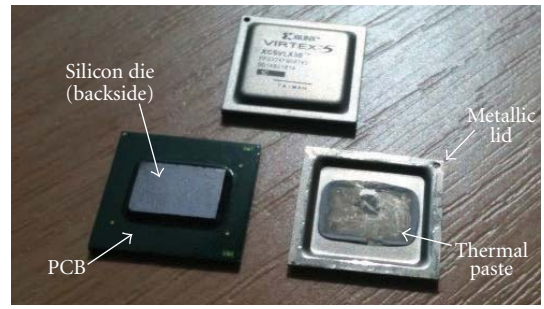

Figure 5: Photograph of the FPGA chip whose cryptographic modules are to be located by cross-correlation cartography.

we have extracted only the map with the greatest number of such points.

Proceeding this way, we have obtained eleven areas of interest. The nine most significant ones are reported on Figure 6, with a disposition in the page that reflects their location within the FPGA. As in Figure 3, reference points are marked with a white cross. The maps (a) and (i) pinpoint two areas in the top left hand and bottom right hand corners. At first sight, they correspond to the two sensible registers LR and MASK. To confirm this, we have conducted large acquisition campaigns of 1,000 observations per point, then computed for each of them the CEMA factor, that is, CPA (Correlation Power Analysis) with electromagnetic waves. We denote by ρ this CEMA factor, to distinguish it from Γ, the NXC coefficient defined in (1). Note that we use data

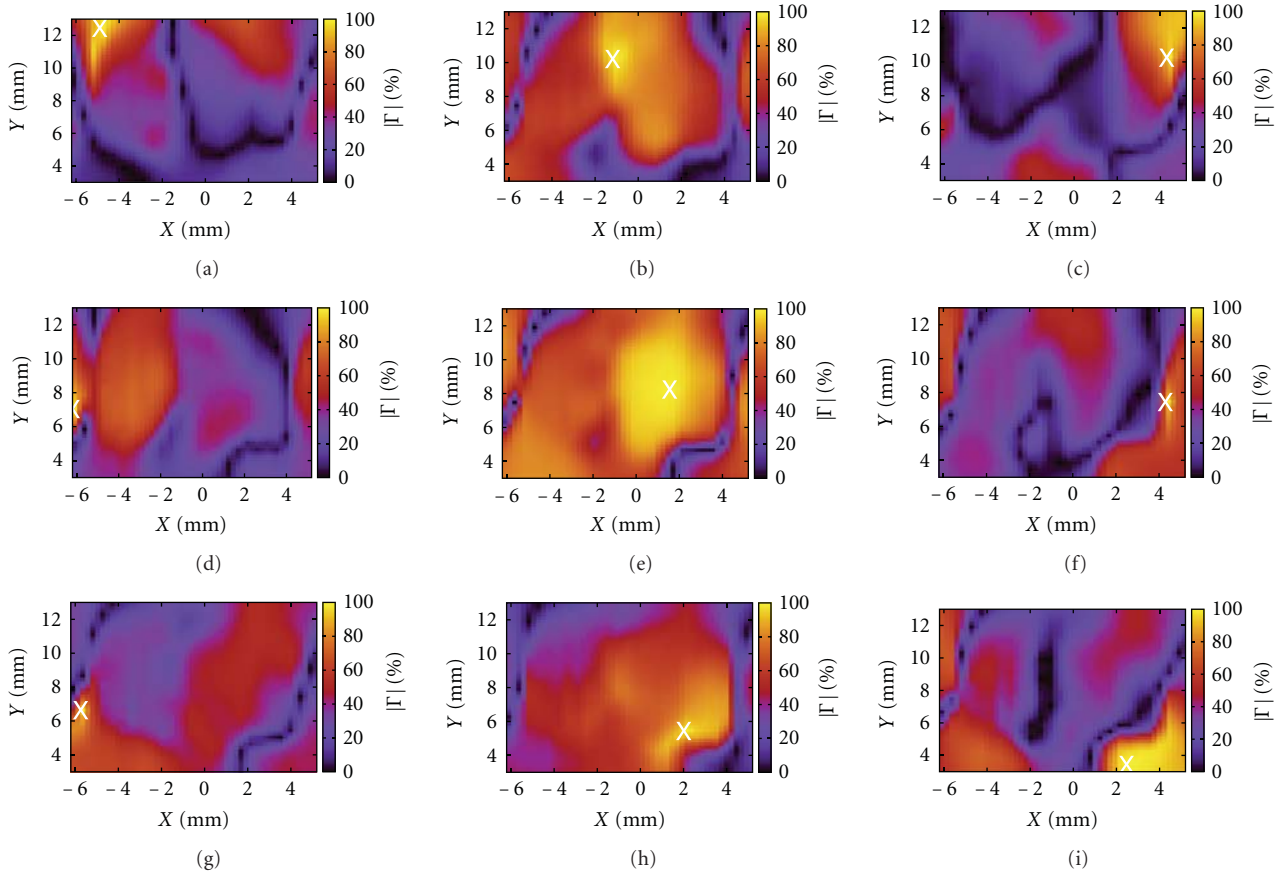

Figure 6: Some maps obtained by cross-correlation. The white cross indicates the point of greatest Γ.

normally not accessible to an attacker such as the mask's value: this is, however, possible in an evaluation context. The resulting maps for the MASK and LR registers are depicted by Figure 8. The CEMA clearly identifies that area of interest (a) is correlated with the mask and that area (i) is correlated with the masked data. In these two CEMA maps, the point with the maximum ρ correlation is marked with a white cross. This location coincides almost exactly with that of maximum Γ in the NXC maps. Hence, the proof that the methodology succeeds in insulating areas of consistent activity. Therefore, our main objective has been successfully reached. This result is very precious to continue with a HO-SCA (second order) taking advantage of this spatial diversity: LR is leaking more about the masked data, whereas MASK discloses information more related to the mask. HO-SCA such as those based on correlation reviewed in [22] or the one based on information theory in [23] would advantageously combine observations over these points. Identifying the other areas is not trivial as their shapes do not fit the arrangement of Figure 4. Indeed, EM radiations are generally more likely due to the power grid and the clock tree of the FPGA [10] than to its logic cells and routing paths.

To complement the analysis, Figure 7 delivers the output voltage of the EM probe when this latter is just over the points of interest. Except for the map (f), the 16 rounds of the DES encryption are neatly visible in the right hand part of the observations. The 16 peaks amplitude varies in time, but not in the same way from a position to another one, which confirms that we observe the activity of distinct elements. We guess that the observation that coincides with the locations:

(i) (e) and/or (h) may be due to the key scheduling;

(ii) (d) and/or (g) to some reads/writes on the VCI bus;

(iii) (b) and/or (c) and/or (f) to some combinatorial functions in the 3DES datapath, such as exclusive logical OR.

We insist that our blind cartography method does not actually distinguish cryptographic blocks from the others. But still, the method has the following interests.

(i) It highlights "equivalent areas" for EMA. Once those areas of interest are localized, the attacker can focus her measurements on them. In our example, this reduces the number of positions from $25 \times 25 = 625$ to only 11.

(ii) Applied to the second-order attack of a first-order masking scheme, the number of combinations to be tested to match the mask and the masked data activity is only $\binom{11}{2} = 55$. Without NXC, the number of couples to test would be equal to $\binom{25 \times 25}{2}$ 200,000, which is deterrent for an attacker, but the computational workload is too high.

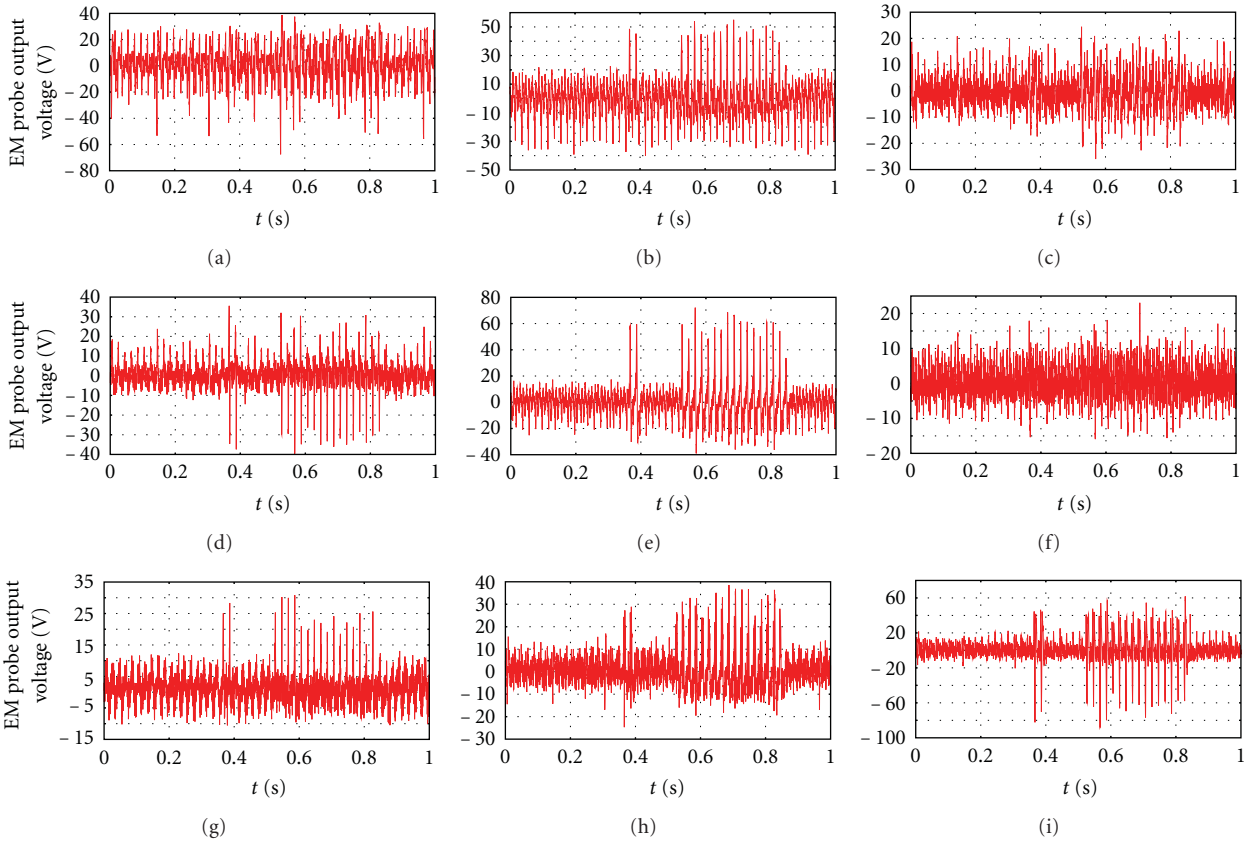

Figure 7: Output voltage of the EM probe when positioned over the points of interest.

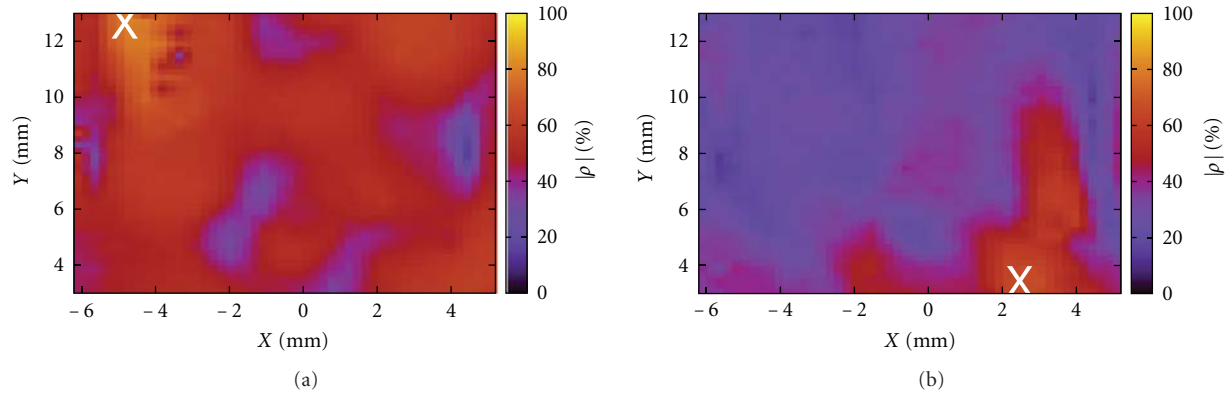

Figure 8: CEMA maps obtained knowing the activity of the MASK (a) and LR (b) registers. The white cross indicates the point of greatest ρ.

4. Conclusion and Future Works

Many implementation-level attacks can be enhanced if the floorplan of the application is known by the attacker. For instance, side-channel measurements can be made less noisy if focused on the most leaking zone, and fault injection attacks (by electromagnetic waves or laser shots) have indeed more chance to succeed in perturbing the adequate resource if positioned well in a vicinity of the zone of influence. As far as ASICs are concerned, the location of each module can be guessed by an optical analysis of chip photographs. Modern ASICs (such as modern smartcards) have their logic dissolved so as to make its analysis intractable. Now, regarding FPGAs, the problem is the same, since the fabric is extremely regular and does not show the location of the user design. In addition, FPGA chips are wider than ASICs, thus the research for sensitive regions is *a priori* more complex.

In this paper, we introduce a novel location method based on cross-correlation of electromagnetic cartographies. It is indeed able to reveal the position of blocks. This shows that the structure of the floorplan shall not be considered confidential in FPGAs, even if the bitstream is confidential

(e.g., encrypted). Then, we experimentally demonstrate that the cross-correlation location method is efficient to pinpoint areas of interest in the context of a protected cryptographic application. This methodology illustrates a new aspect of the wealth of the information carried out by the electromagnetic field leaked by electronic devices. The floorplan reverse-engineering method presented in this paper is an algorithm-agnostic preliminary step that enables the further realization of well-focused electromagnetic analysis attacks aiming this time at extracting secrets. We have exemplified this method with the successful localization of the registers that hold the mask and the masked data that are manipulated concomitantly. Being able to record traces from both locations allows for second-order attacks by combination of the twain measurements [24]. Also, the same method could be used to record traces selectively from one half of separable dual-rail logic styles (such as SDDL [25, Section 3.1], DWDDL [26], divided backend duplication [27], partial DDL [28], or PA-DDL [29]) thereby defeating the complementation property of those "hiding" countermeasures.

References

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the 19th Annual International Cryptology Conference Advances in Cryptology (CRYPTO '99)*, vol. 1666 of *Lecture Notes in Computer Science*, pp. 388–397, Springer, Santa Barbara, Cali, USA, 1999.

[2] R. Anderson and M. Kuhn, "Tamper resistance—a cautionary note," in *Proceedings of the 2nd USENIX Workshop on Electronic Commerce (WOEC'96)*, pp. 1–11, USENIX Association, Berkeley, Calif, USA, 1996.

[3] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: concrete results," in *Proceedings of the 3rd International Workshop Cryptographic Hardware and Embedded Systems (CHES'01)*, C. K. Koc, D. Naccache, and C. Paar, Eds., vol. 2162 of *Lecture Notes in Computer Science*, pp. 251–261, Springer, Paris, France, 2001.

[4] M. Agoyan, J.-M. Dutertre, A.-P. Mirbaha, D. Naccache, A.-L. Ribotta, and A. Tria, "Single-bit DFA using multiple-byte laser fault injection," in *Proceedings of the IEEE International Conference on Technologies for Homeland Security (HST'10)*, pp. 113–119, 2010.

[5] G. Canivet, J. Clédière, J. B. Ferron, F. Valette, M. Renaudin, and R. Leveugle, "Detailed analyses of single laser shot effects in the configuration of a Virtex-II FPGA," in *14th IEEE International On-Line Testing Symposium, (IOLTS '08)*, pp. 289–294, Rhodes, Greece, 2008.

[6] S. P. Skorobogatov, "Using optical emission analysis for estimating contribution to power analysis," in *Proceedings of the 6th International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC '09)*, pp. 111–119, IEEE Computer Society, Lausanne, Switzerland, 2009.

[7] J. Di-Battista, J.-C. Courrège, B. Rouzeyre, L. Torres, and P. Perdu, "When failure analysis meets side-channel attacks," in *Proceedings of the 12th International Workshop Cryptographic Hardware and Embedded Systems (CHES '10)*, Santa Barbara, Calif, USA, 2010.

[8] D. Réal, F. Valette, and M. Drissi, "Enhancing correlation electromagnetic attack using planar near-field cartography," in *Proceedings of the Design, Automation and Test in Europe, (DATE '09)*, pp. 628–633, IEEE, Nice, France, April 2009.

[9] A. Dehbaoui, V. Lomne, P. Maurine, and L. Torres, "Magnitude squared incoherence em analysis for integrated cryptographic module localisation," *Electronics Letters*, vol. 45, no. 15, pp. 778–780, 2009.

[10] L. Sauvage, S. Guilley, and Y. Mathieu, "ElectroMagnetic radiations of FPGAs: high spatial resolution cartography and attack of a cryptographic module," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, no. 1, pp. 1–24, 2009.

[11] L. Sauvage, S. Guilley, J.-L. Danger, Y. Mathieu, and M. Nassar, "Successful attack on an FPGA-based WDDL DES cryptoprocessor without place and route constraints," in *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, pp. 640–645, IEEE, Nice, France, April, 2009.

[12] É. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Proceedings of the 6th International Workshop Cryptographic Hardware and Embedded Systems (CHES'04)*, vol. 3156 of *Lecture Notes in Computer Science*, pp. 16–29, Springer, Cambridge, Mass, USA, August, 2004.

[13] L. Goubin and J. Patarin, "DES and differential power analysis (The "Duplication" Method)," in *Proceedings of the 1st International Workshop Cryptographic Hardware and Embedded Systems (CHES'99)*, vol. 1717 of *Lecture Notes in Computer Science*, pp. 158–172, Worcester, Mass, USA, August, 1999.

[14] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Proceedings of the 19th Annual International Cryptology Conference Advances in Cryptology (CRYPTO '99)*, vol. 1666 of *Lecture Notes in Computer Science*, pp. 398–412, Springer, Santa Barbara, Calif, USA, August, 1999.

[15] J.-L. Danger, S. Guilley, S. Bhasin, and M. Nassar, "Overview of dual rail with precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors," in *Proceedings of the 3rd International Conference on Signals, Circuits and Systems (SCS'09)*, pp. 1–8, IEEE, Jerba, Tunisia, November 2009.

[16] National Institute of Standards and Technology, "Data Encryption Standard (DES): FIPS PUB 46-3," 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf. .

[17] E. Peeters, F.-X. Standaert, N. Donckers, and J.-J. Quisquater, "Improved higher-order side-channel attacks with FPGA experiments," in *Proceedings of the 7th International Workshop Cryptographic Hardware and Embedded Systems (CHES '05)*, vol. 3659 of *Lecture Notes in Computer Science*, pp. 309–323, Springer, Edinburgh, UK, 2005.

[18] National Institute of Standards and Technology, "Advanced Encryption Standard (AES): FIPS PUB 197," 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[19] T. S. Messerges, "Using second-order power analysis to attack DPA resistant software," in *Proceedings of the 2nd International Workshop Cryptographic Hardware and Embedded Systems (CHES'00)*, vol. 1965 of *Lecture Notes in Computer Science*, pp. 238–251, Springer, Worcester, Mass, USA, August, 2000.

[20] "Xilinx FPGA designer," http://www.xilinx.com/.

[21] "Agilent Technologies," http://www.home.agilent.com/.

[22] E. Prouff, M. Rivain, and R. Bévan, "Statistical analysis of second order differential power analysis," *IEEE Transactions on Computers*, vol. 58, no. 6, pp. 799–811, 2009.

[23] B. Gierlichs, L. Batina, B. Preneel, and I. Verbauwhede, "Revisiting higher-order DPA attacks: multivariate mutual information analysis," in *Proceedings of the The Cryptographer's Track at RSA Conference (CT-RSA'10)*, vol. 5985 of *Lecture Notes in Computer Science*, pp. 221–234, Springer, San Francisco, Calif, USA, March 2010.

[24] E. Prouff, M. Rivain, and R. Bévan, "Statistical analysis of second order differential power analysis," *IEEE Transactions on Computers*, vol. 58, no. 6, pp. 799–811, 2009.

[25] K. Tiri and I. Verbauwhede, "A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, pp. 246–251, IEEE Computer Society, Paris, France, February 2004.

[26] P. Yu and P. Schaumont, "Secure FPGA circuits using controlled placement and routing," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS'07)*, pp. 45–50, ACM, New York, NY, USA, 2007.

[27] K. Baddam and M. Zwolinski, "Divided Backend duplication methodology for balanced dual rail routing," in *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES '08)*, vol. 5154 of *Lecture Notes in Computer Science*, pp. 396–410, Springer, Washington, DC, USA, 2008.

[28] J.-P. Kaps and R. Velegalati, "DPA Resistant AES on FPGA Using Partial DDL," in *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machine (FCCM'10)*, pp. 273–280, IEEE Computer Society, Charlotte, NC, USA, May, 2010.

[29] W. He, E. D. L. Torre, and T. Riesgo, "A precharge-absorbed DPL logic for reducing early propagation effects on FPGA implementations," in *Proceedings of the ReConFig*, IEEE Computer Society, Quintana Roo, México, 2011.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 675130, 11 pages
doi:10.1155/2012/675130

Research Article

A Hardware Efficient Random Number Generator for Nonuniform Distributions with Arbitrary Precision

Christian de Schryver,[1] Daniel Schmidt,[1] Norbert Wehn,[1] Elke Korn,[2] Henning Marxen,[2] Anton Kostiuk,[2] and Ralf Korn[2]

[1] *Microelectronic Systems Design Research Group, University of Kaiserslautern, Erwin-Schroedinger-Straße, 67663 Kaiserslautern, Germany*
[2] *Stochastic Control and Financial Mathematics Group, University of Kaiserslautern, Erwin-Schroedinger-Straße, 67663 Kaiserslautern, Germany*

Correspondence should be addressed to Christian de Schryver, schryver@eit.uni-kl.de

Received 30 April 2011; Accepted 23 November 2011

Academic Editor: Ron Sass

Nonuniform random numbers are key for many technical applications, and designing efficient hardware implementations of nonuniform random number generators is a very active research field. However, most state-of-the-art architectures are either tailored to specific distributions or use up a lot of hardware resources. At ReConFig 2010, we have presented a new design that saves up to 48% of area compared to state-of-the-art inversion-based implementation, usable for arbitrary distributions and precision. In this paper, we introduce a more flexible version together with a refined segmentation scheme that allows to further reduce the approximation error significantly. We provide a free software tool allowing users to implement their own distributions easily, and we have tested our random number generator thoroughly by statistic analysis and two application tests.

1. Introduction

The fast generation of random numbers is essential for many tasks. One of the major fields of application are Monte Carlo simulation, for example widely used in the areas of financial mathematics and communication technology.

Although many simulations are still performed on high-performance CPU or general-purpose graphics processing unit (GPGPU) clusters, using reconfigurable hardware accelerators based on field programmable gate arrays (FPGAs) can save up to at least one order of magnitude of power consumption if the random number generator (RNG) is located on the accelerator. As an example, we have implemented the generation of normally distributed random numbers on the three mentioned architectures. The results for the achieved throughput and the consumed energy are given in Table 1. Since one single instance of our proposed hardware design (together with a uniform random number generator) consumes less than 1% of the area on the used Xilinx Virtex-5 FPGA, we have introduced a line with the extrapolated values for 100 instances to highlight the enormous potential of hardware accelerators with respect to the achievable throughput per energy.

In this paper, we present a refined version of the floating point-based nonuniform random number generator already shown at ReConFig 2010 [1]. The modifications allow a higher precision while having an even lower area consumption compared to the previous results. This is due to a refined synthesis. The main benefits of the proposed hardware architecture are the following:

(i) The area saving is even higher than the formerly presented 48% compared to the state-of-the-art FPGA implementation of Cheung et al. from 2007 [2].

(ii) The precision of the random number generator can be adjusted and is mainly independent of the output resolution of the auxiliary uniform RNG.

Table 1: Normal random number generator architecture comparison.

Implementation	Architecture	Power consumption	Throughput [M samples/s]	Energy per sample
Fast Mersenne Twister, optimized for SIMD	Intel Core 2 Duo PC 2.0 GHz, 3 GB RAM, one core only	~100 W	600	166.67 pJ
Nvidia Mersenne Twister + Box-Muller CUDA	Nvidia GeForce 9800 GT	~105 W	1510	69.54 pJ
Nvidia Mersenne Twister + Box-Muller OpenCL			1463	71.77 pJ
Proposed architecture, only one instance [1]	Xilinx FPGA Virtex-5FX70T-3 380 MHz	~1.3 W	397	3.43 pJ
Proposed architecture, 100 instances		~1.9 W	39700	0.05 pJ

(iii) Our design is exhaustively tested by statistical and application tests to ensure the high quality of our implementation.

(iv) For the convenience of the user, we provide a free tool that creates the lookup table (LUT) entries for any desired nonuniform distribution with a user-defined precision.

The rest of the paper is organized as follows. In Section 2, we give an overview about current techniques to obtain uniform (pseudo-)random numbers and to transform them to nonuniform random numbers. Section 3 shows state-of-the-art inversion-based FPGA nonuniform random number generators, as well as a detailed description of the newly introduced implementation. It also presents the LUT creator tool needed for creating the lookup table entries. How floating point representation can help to reduce hardware complexity is explained in Section 4. Section 5 shows detailed synthesis results of the original and the improved implementation and elaborates on the excessive quality tests that we have applied. Finally, Section 6 concludes the paper.

2. Related Work

The efficient implementation of random number generators in hardware has been a very active research field for many years now. Basically, the available implementations can be divided into two main groups, that are

(i) random number generators for uniform distributions,

(ii) circuits that transform uniformly distributed random numbers into different target distributions.

Both areas of research can, however, be treated as nearly distinct. We will give an overview of available solutions out of both groups.

2.1. Uniform Random Number Generators. Many highly elaborate implementations for uniform RNGs have been published over the last decades. The main common characteristic of all is that they produce a bit vector with n bits that represent (if interpreted as an unsigned binary-coded integer and divided by $2^n - 1$) values between 0 and 1. The set of all results that the generator produces should be as uniformly as possible distributed over the range $(0,\ 1)$.

A lot of fundamental research on uniform random number generation has already been made before 1994. A comprehensive overview of the work done until that point in time has been given by L'Ecuyer [3] who summarized the main concepts of uniform RNG construction and their mathematical backgrounds. He also highlights the difficulties of evaluating the quality of a uniform RNG, since in the vast majority of the cases, we are dealing not with truly random sequences (as, e.g., Bochard et al. [4]), but with pseudorandom or quasirandom sequences. The latter ones are based on deterministic algorithms. *Pseudorandomness* means that the output of the RNG looks to an observer like a truly random number sequence if only a limited period of time is considered. *Quasirandom* sequences, however, do not aim to look very random at all, but rather try to cover a certain bounded range in a best even way. One major field of application for quasirandom numbers is to generate a suitable test point set for Monte Carlo simulations, in order to increase the performance compared to pseudorandom number input [5, 6].

One of the best investigated high-quality uniform RNGs is the Mersenne Twister as presented by Matsumoto and Nishimura in 1998 [7]. It is used in many technical applications and commercial products, as well as in the RNG research domain. Well-evaluated and optimized software programs are available on their website [8]. Nvidia has adapted the Mersenne Twister to their GPUs in 2007 [9].

A high-performance hardware architecture for the Mersenne Twister has been presented in 2008 by Chandrasekaran and Amira [10]. It produces 22 millions of samples per second, running at 24 MHz. Banks et al. have compared their Mersenne Twister FPGA design to two multiplier pseudo-RNGs in 2008 [11], especially for the use in financial mathematics computations. They also clearly show that the random number quality can be directly traded off against the consumed hardware resources.

Tian and Benkrid have presented an optimized hardware implementation of the Mersenne Twister in 2009 [12], where they showed that an FPGA implementation can outperform a state-of-the-art multicore CPU by a factor of about 25, and a GPU by a factor of about 9 with respect to the throughput. The benefit for energy saving is even higher.

We will not go further into details here since we concentrate on obtaining nonuniform distributions. Nevertheless, it is worth mentioning that quality testing has been a big issue for uniform RNG designs right from the beginning [3]. L'Ecuyer and Simard invented a comprehensive test suite named *TestU01* [13] that is written in C (the most recent version is 1.2.3 from August, 2009). This suite combines a lot of various tests in one single program, aimed to ensure the quality of specific RNGs. For users without detailed knowledge about the meaning of each single test, the TestU01 suite contains three test batteries that are predefined selections of several tests:

(i) *Small Crush*: 10 tests,

(ii) *Crush*: 96 tests,

(iii) *Big Crush*: 106 tests.

TestU01 includes and is based on the tests from the other test suites that have been used before, for example, the Diehard Test Suite by Marsaglia from 1995 [14] or the fundamental considerations made by Knuth in 1997 [15].

For the application field financial mathematics (what is also our main area of research), McCullough has strongly recommended the use of TestU01 in 2006 [16]. He comments on the importance of random number quality and the need of excessive testing of RNGs in general.

More recent test suites are the very comprehensive Statistical Test Suite (STS) from the US National Institute of Standards and Technology (NIST) [17] revised in August, 2010, and the Dieharder suite from Robert that was just updated in March, 2011 [18].

2.2. Obtaining Nonuniform Distributions. In general, nonuniform distributions are generated out of uniformly distributed random numbers by applying appropriate conversion methods. A very good overview of the state-of-the-art approaches has been given by Thomas et al. in 2007 [19]. Although they are mainly concentrating on the normal distribution, they show that all applied conversion methods are based on one of the four underlying mechanisms:

(i) transformation,

(ii) rejection sampling,

(iii) inversion,

(iv) recursion.

Transformation uses mathematical functions that provide a relation between the uniform and the desired target distribution. A very popular example for normally distributed random numbers is the Box-Muller method from 1958 [20]. It is based on trigonometric functions and transforms a pair of uniformly distributed into a pair of normally distributed random numbers. Its advantage is that it provides a pair of random numbers for each call deterministically. The Box-Muller method is prevalent nowadays and mainly used for CPU and GPU implementations. A drawback for hardware implementations is the high demand of resources needed to accurately evaluate the trigonometric functions [21, 22].

Rejection sampling can provide a very high accuracy for arbitrary distributions. It only accepts input values if they are within specific predefined ranges and discards others. This behavior may lead to problems if quasirandom number input sequences are used, and (especially important for hardware implementations) unpredictable stalling might be necessary. For the normal distribution, the Ziggurat method [23] is the most common example of rejection sampling and is implemented in many software products nowadays. Some optimized high-throughput FPGA implementations exist, for example, by Zhang et al. from 2005 [24] who generated 169 millions of samples per second on a Xilinx Virtex-2 device running at 170 MHz. Edrees et al. have proposed a scalable architecture in 2009 [25] that achieves up to 240 Msamples on a Virtex-4 at 240 MHz. By increasing the parallelism of their architecture, they predicted to achieve even 400 Msamples for a clock frequency of around 200 MHz.

The *inversion method* applies the inverse cumulative distribution function (ICDF) of the target distribution to uniformly distributed random numbers. The ICDF converts a uniformly distributed random number $x \in (0, 1)$ to one output $y = \text{icdf}(x)$ with the desired distribution. Since our proposed architecture is based on the inversion method, we go more into details in Section 3.

The so far published hardware implementations of inversion-based converters are based on piecewise polynomial approximation of the ICDF. They use lookup tables (LUTs) to store the coefficients for various sampling points. Woods and Court have presented an ICDF-based random number generator in 2008 [26] that is used to perform Monte Carlo simulations in financial mathematics. They use a nonequidistant hierarchical segmentation scheme with smaller segments in the steeper parts of the ICDF, what reduces the LUT storage requirements significantly without losing precision. Cheung et al. have shown a very elaborate multilevel segmentation approach in 2007 [2].

The *recursion method* introduced by Wallace in 1996 [27] uses linear combinations of originally normally distributed random numbers to obtain further ones. He provides the source code of his implementation for free [28]. Lee et al. have shown a hardware implementation in 2005 [29] that produces 155 millions of samples per second on a Xilinx Virtex-2 FPGA running at 155 MHz.

3. The Inversion Method

The most genuine way to obtain nonuniform random numbers is the *inversion method*, as it preserves the properties of the originally sampled sequence [30]. It uses the ICDF of the desired distribution to transform every input $x \in (0, 1)$ from a uniform distribution into the output sample $y = \text{icdf}(x)$ of the desired one. In case of a continuous and strictly monotone cumulative distribution (CDF) function F, we have

$$F_{\text{out}}(\alpha) = \mathbb{P}(\text{icdf}(U) \leq \alpha) = \mathbb{P}(U \leq F(\alpha)) = F(\alpha). \quad (1)$$

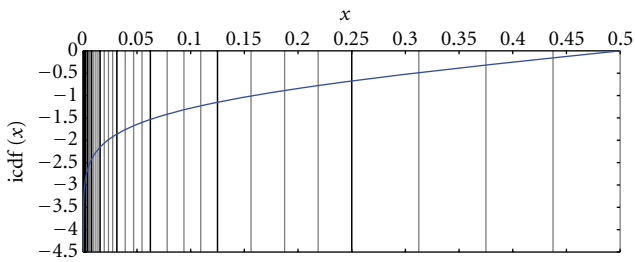

Figure 1: Segmentation of the first half of the Gaussian ICDF.

Identical CDFs always imply the equality of the corresponding distributions. For further details, we refer to the works of Korn et al. [30] or Devroye [31].

Due to the above mechanism, the inversion method is applicable to transform also quasirandom sequences. In addition to that, it is completable with variance reduction techniques, for example, antithetic variates [26]. Inversion-based methods in general can be used to obtain any desired distribution using memory-based lookup tables. This is especially advantageous for hardware implementations, since for many distributions, no closed-form expressions for the ICDF exist, and approximations have to be used. The most common approximations for the Gaussian ICDF (see Peter [32] and Moro [33]) are, however, based on higher-grade rational polynomials, but, for that reason, they cannot be efficiently used for a hardware implementation.

3.1. State-of-the-Art Architectures. In 2007, Cheung et al. proposed to implement the inversion using the piecewise polynomial approximation [2]. It is based on a fixed point representation and uses a hierarchical segmentation scheme that provides a good trade-off between hardware resources and accuracy. For the normal distribution (as well as any other symmetric distribution), it is also common to use the following simplification: due to the symmetry of the normal ICDF around $x = 0.5$, its approximation is implemented only for values $x \in (0,\ 0.5)$, and one additional random bit is used to cover the full range. For the Gaussian ICDF, Cheung et al. suggest to divide the range $(0,\ 0.5)$ into nonequidistant segments with doubling segment sizes from the beginning to the end of the interval. Each of these segments should then be subdivided into inner segments of equal size. Thus, the steeper regions of the ICDF close to 0 are covered by more smaller segments than the regions close to 0.5, where the ICDF is almost linear. This segmentation of the Gaussian ICDF is shown in Figure 1. By using a polynomial approximation of a fixed degree within each segment, this approach allows to obtain an almost constant maximal absolute error over all segments. The inversion algorithm first determines in which segment the input x is contained, then retrieves the coefficients c_i of the polynomial for this segment from a LUT, and evaluates the output as $y = \sum c_i \cdot x^i$ afterwards.

Figure 2 explains how, for a given fixed point input x, the coefficients of the polynomial are retrieved from the lookup table (that means how the address of the corresponding

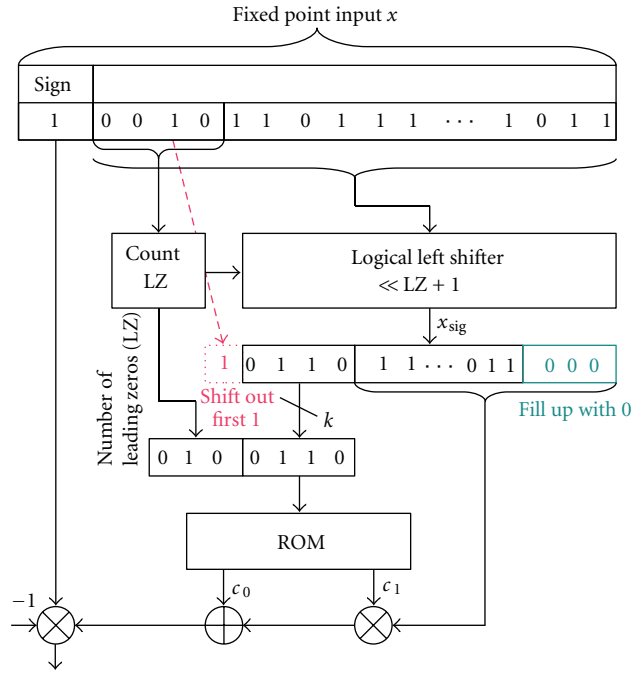

Figure 2: State-of-the-art architecture.

segment in the LUT is generated). It starts with counting the number of leading zeros (LZ) in the binary representation of x. It uses a bisection technique to locate the segment of the first level: that means numbers with the most significant bit (MSB) 1 lie in the segment $[0.25,\ 0.5)$ and those with 0 correspondingly in $(0,\ 0.25)$, numbers with second MSB 1 (i.e., $x = 01\ldots$) lie in the segment $[0.125,\ 0.25)$ and those with 0 (i.e., $x = 00\ldots$) in $(0,\ 0.125)$, and so forth. Then the input x is shifted left by $LZ + 1$ bits, such that x_{sig} is the bit sequence following the most significant 1-bit in x. The k MSBs of x_{sig} determine the subsegments of the second level (the equally sized ones). Thus, the LUT address is the concatenation of LZ and $MSB_k(x_{\text{sig}})$. The inverted value equals the approximating polynomial for the ICDF in that segment evaluated on the remaining bits of x_{sig}. The architecture for the case of linear interpolation [2] is presented in Figure 2. It approximates the inversion with a maximum absolute error of $0.3 \cdot 2^{-11}$.

The works of Lee et al. [34, 35] are also based on this segmentation/LUT approach. They use the same technique to create generators for the log-normal and the exponential distributions, with only slight changes in the segmentation scheme. For the exponential distribution, the largest segment starts near 0, sequentially followed by the twice smaller segments towards 1. For the log-normal distribution, neighboring segments double in size starting from 0 until 0.5 and halve in size towards 1.

But this approach has a number of drawbacks as follows.

(i) *Two uniform RNGs needed for a large output range*: due to the fixed point implementation, the output range is limited by a number of input bits. The smallest positive value that can be represented by an

m bit fixed point number is 2^{-m}, what in the case of a 32-bit input value leads to the largest inverted value of $\text{icdf}(2^{-32}) = 6.33\sigma$. To obtain a larger range of normal random variable up to 8.21σ, the authors of [2] concatenate the input of two 32-bit uniform RNGs and pass a 53-bit fixed point number into the inversion unit, at the cost of one additional uniform RNG. The large number of input bits results in the increased size of the LZ counter and shifter unit, that dominate the hardware usage of the design.

(ii) *A large number of input bits is wasted:* as a multiplier with a 53-bit input requires a large amount of hardware resources, the input is quantified to 20 significant bits before the polynomial evaluation. Thus, in the region close to the 0.5, a large amount of the generated input bits is wasted.

(iii) *Low resolution in the tail region:* for the tail region (close to 0), there are much less than 20 significant bits left after shifting over the LZ. This limits the resolution in the tail of the desired distribution. In addition, as there are no values between 2^{-53} and 2^{-52} in this fixed point representation, the proposed RNG does not generate output samples between $\text{icdf}(2^{-52}) = 8.13\sigma$ and $\text{icdf}(2^{-53}) = 8.21\sigma$.

3.2. Floating Point-Based Inversion. The drawbacks mentioned before result from the fixed point interpretation of the input random numbers. We therefore propose to use a floating point representation.

First of all, we do not use any floating point arithmetics in our implementation. Our design does not contain any arithmetic components like full adders or multipliers that usually blow up a hardware architecture. We just exploit the representation of a floating point number consisting of an exponent and a mantissa part. We also do not use IEEE 754 [36] compliant representations, but have introduced our own optimized interpretation of the floating point encoded bit vector.

3.2.1. Hardware Architecture. We have enhanced our formerly architecture presented at ReConFig 2010 [1] with a second *part* bit that is used to split the encoded half of the ICDF into two parts. The additionally necessary hardware is just one multiplexer and an adder with one constant input, that is, the offset for the address range of the LUT memory where the coefficients for the second half are located.

Figure 3 shows the structure of our proposed ICDF lookup unit. Compared to our former design, we have renamed the *sign_half* bit to *symmetry* bit. This term is more appropriate now since we use this bit to identify in which half of a symmetrical ICDF the output value is located. In this case, we also only encode one half and use the symmetry bit to generate a symmetrical coverage of the range (0, 1) (see Section 3.1).

Each part itself is divided further into *octaves* (formerly *segments*), that are halved in size by moving towards the outer borders of the parts (compare with Section 3.1). One exception is that the both very smallest octaves are equally

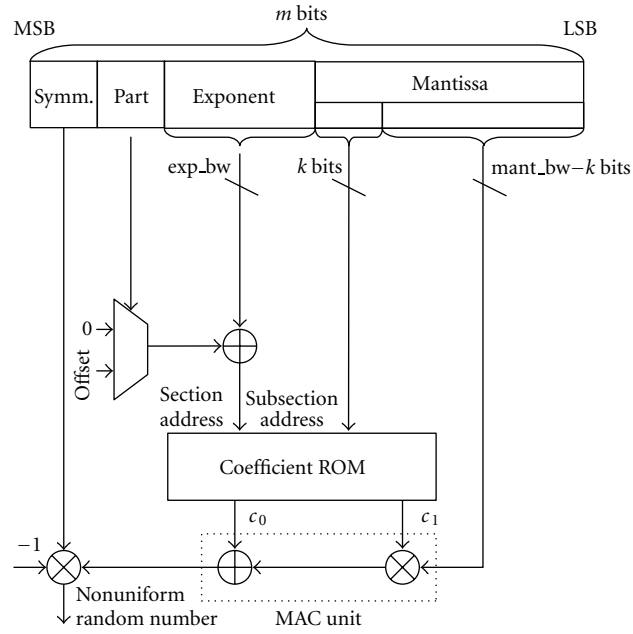

Figure 3: ICDF lookup structure for linear approximation.

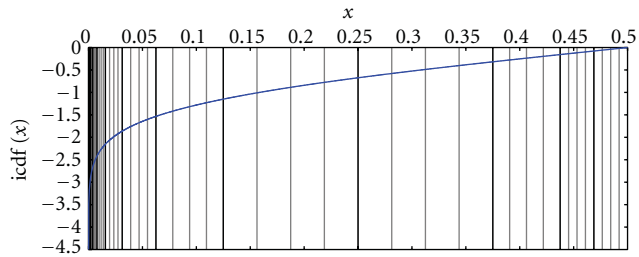

Figure 4: Double segmentation refinement for the normal ICDF.

sized. In general, the number of octaves for each part can be different. As an example, Figure 4 shows the left half of the Gaussian ICDF with a nonequal number of octaves in both parts.

Each octave is again divided into 2^k equally sized *subsections*, where k is the number of bits taken from the mantissa part in Figure 3. k therefore has the same value for both parts, but is not necessarily limited to powers of 2.

The input address for the coefficient ROM is now generated in the following way.

(i) The offset is exactly the number of subsections in part 0, that means all subsections in the range from 0 to 0.25 for a symmetric ICDF:

$$\text{offset} = 2^k \cdot \text{number of octaves in part 0}. \quad (2)$$

(ii) In part 0, the address is the concatenation of the exponent (giving the number of the octave) and the k dedicated mantissa bits (for the subsection).

(iii) In part 1, the address is the concatenation of (exponent + offset) and the k mantissa bits.

Table 2: Selected tool configuration for provided error values.

Parameter	Value
Growing octaves	54
Diminishing octaves	4
Subsection bits (k)	3
Mantissa bits ($mant\_bw - k$)	18
Output precision bits	42

This floating point-based addressing scheme efficiently exploits the LUT memory in a hardware friendly way since no additional logic for the address generation is needed compared to other state-of-the-art implementations (see Sections 2.2 and 3.1). The necessary LUT entries can easily be generated with our freely available tool presented in Section 3.2.2.

3.2.2. The LUT Creator Tool. For the convenience of the users who like to make use of our proposed architecture, we have developed a flexible C++ class package that creates the LUT entries for any desired distribution function. The tool has been rewritten from scratch, compared to the one presented at ReConFig 2010 [1]. It is freely available for download on our website (http://ems.eit.uni-kl.de/).

Most of the detailed documentation is included in the tool package itself. It uses Chebyshev approximation, as provided by the GNU Scientific Library (GSL) [37]. The main characteristics of the new tool are as follows.

(i) It allows any function defined on the range (0, 1) to be approximated. However, the GSL already provides a large number of ICDFs that may be used conveniently.

(ii) It provides configurable segmentation schemes with respect to

  (i) symmetry,
  (ii) one or two parts,
  (iii) independently configurable number of octaves per part,
  (iv) number of subsections per octave.

(iii) The output quantization is configurable by the user.

(iv) The degree of the polynomial approximation is arbitrary.

Our LUT creator tool also has a built-in error estimation that directly calculates the maximum errors between the provided optimal function and the approximated version. For linear approximation and the configuration shown in Table 2, we present a selection of maximum errors in Table 3. For optimized parameter sets that take the specific characteristics of the distributions into account, we expect even lower errors.

Table 3: Maximum approximation errors for different distributions.

Distribution	Symmetry	Maximum absolute error
Normal	Point	0.000383397
Log-normal (0, 1)	None	0.00233966
Gamma (0, 1)	None	0.00787368
Laplace (1)	Point	0.000901326
Exponential (1)	None	0.000787368
Rayleigh (1)	None	0.000300666

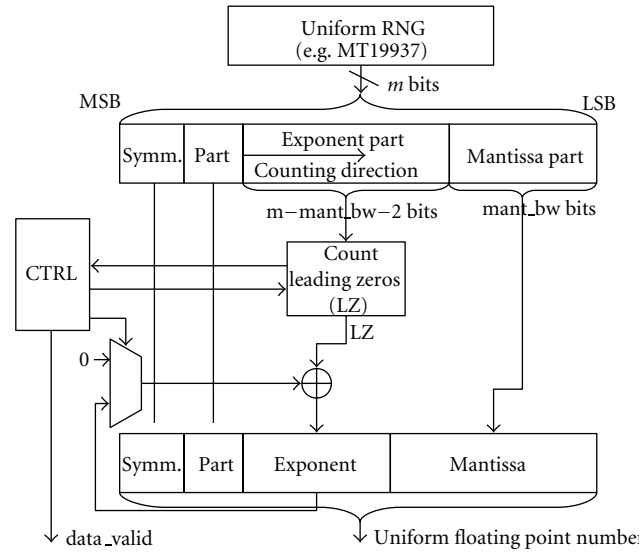

Figure 5: Architecture of the proposed floating point RNG.

4. Generating Floating Point Random Numbers

Our proposed LUT-based inversion unit shown in Section 3.2.1 requires dedicated floating point encoded numbers as inputs. In this section, we present an efficient hardware architecture for generating these numbers. Our design consumes an arbitrary-sized bit vector from any uniform random number generator and transforms it into the floating point representation with adjustable precisions. In Section 5.2, we show that our floating point converter maintains the properties of the uniform random numbers provided by the input RNG.

Figure 5 shows the structure of our unit and how it maps the incoming bit vector to the floating point parts. Compared to our architecture presented on ReConFig 2010 [1], we have enhanced our converter unit with an additional *part* bit. It provides the information if we use the first or the second segmentation refinement of the ICDF approximation (see Section 3.2).

For each floating point random number that is to be generated, we extract the symmetry and the part bit in the first clock cycle, as well as the mantissa part that is just mapped to the output one to one. The mantissa part in our case is encoded with a hidden bit, for a bit width of

mant_bw bits, it can therefore represent the values 1, $1 + (1/2^{mant\_bw})$, $1 + (2/2^{mant\_bw}), \ldots, 2 - (1/2^{mant\_bw})$.

The exponent in our floating point encoding represents the number of leading zeros (LZs) that we count from the exponent part of the incoming random number bit vector. We can use this exponent value directly as the segment address in our ICDF lookup unit described in Section 3.2.1. In the hardware architecture, the leading zeros computation is, for efficiency reasons, implemented as a comparator tree.

However, if we would only consider one random number available at the input of our converter, the maximum value for the floating point exponent would be $m - mant\_bw - 2$, with all the bits in the input exponent part being zero. To overcome this issue, we have introduced a parameter determining the maximum value of the output floating point exponent, max_exp. If now all bits in the input exponent part are detected to be zero, we store the value of already counted leading zeros and consume a second random number where we continue counting. For the case that we have again only zeros, we consume a third number and continue if either one is detected in the input part or the predefined maximum of the floating point exponent, max_exp, is reached. In this case, we set the *data_valid* signal to 1 and continue with generating the next floating point random number.

For the reason that we have to wait for further input random numbers to generate one floating point result, we need a stalling mechanism for all subsequent units of the converter. Nevertheless, depending on size of the exponent part in the input bit vector that is arbitrary, the probability for necessary stalling can be decreased significantly. A second random number is needed with the probability of $P_2 = 1/2^{m-mant\_bw-2}$, a third with $P_3 = 1/2^{2\cdot(m-mant\_bw-2)}$, and so on. For an input exponent part with the size of 10 bits, for example, $P_2 = 1/2^{10} = 0.976 \cdot 10^{-3}$, which means that on average one additional input random number has to be consumed for generating about 1,000 floating point results.

We have already presented pseudocode for our converter unit at the ReConFig 2010 [1] that we have enhanced now for our modified design by storing two sign bits. The modified version is shown in Algorithm 1.

```
rn ← get_random_number();
symmetry ← rn.get_symmetry();
part ← rn.get_part();
mant ← rn.get_mantissa();
exp ← rn.get_exponent();
LZ ← exp.count_leading_zeros();
while (exp == 0) and (LZ < max_exp) do
    rn ← get_random_number();
    exp ← rn.get_exponent();
    LZ ← LZ + exp.count_leading_zeros();
end
LZ ← min(LZ, max_exp);
return symmetry, part, mant, LZ
```

Algorithm 1: Floating point generation algorithm.

Table 4: ReConFig 2010 [1]: optimized for speed.

	Slices	FFs	LUTs	BRAMs	DSP48E
Floating point converter	30	62	40	—	—
LUT evaluator	12	47	—	1	1
Complete design	40	108	39	1	1

Table 5: Proposed design: optimized for speed.

	Slices	FFs	LUTs	BRAMs	DSP48E
Floating point converter	30	62	40	—	—
LUT evaluator	18	47	7	1	1
Complete design	42	109	46	1	1

Table 6: ReConFig 2010 [1]: optimized for area.

	Slices	FFs	LUTs	BRAMs	DSP48E
Floating point converter	13	11	26	—	—
LUT evaluator	12	47	—	1	1
Complete design	26	84	26	1	1

5. Synthesis Results and Quality Test

In addition to our conference paper presented at ReConFig 2010 [1], we provide detailed synthesis results in this section on a Xilinx Virtex-5 device, for both speed and area optimization. Furthermore, we show quality tests for the normal distribution.

5.1. Synthesis Results. Like for the proposed architecture from ReConFig 2010, we have optimized the bit widths to exploit the full potential of the Virtex-5 DSP48E slice that supports an $18 \cdot 25$ bit + 48 bit MAC operation. We therefore selected the same parameter values that are as follows: input bitwidth $m = 32$, $mant\_bw = 20$, $max\_exp = 54$, and $k = 3$ for subsegment addressing. The coefficient c_0 is quantized to 46 bits, and c_1 has 23 bits.

We have synthesized our proposed design and the architecture presented at ReConFig with the newer Xilinx ISE 12.4, allowing a fair comparison of the enhancement impacts. Both implementations have been optimized for area and speed, respectively. The target device is a Xilinx Virtex-5 XC5FX70T-3. All provided results are post place and route.

From Tables 4 and 5, we see that just by using the newer ISE version, we already save area of the whole nonuniform random number converter compared to the ReConFig result that was 44 slices (also optimized for speed) [1]. The maximum clock frequency is now 393 MHz compared to formerly 381 MHz.

Even with the ICDF lookup unit extension described in Section 3.2.1, the new design is two slices smaller than in the former version and can run at 398 MHz. We still consume one 36 Kb BRAM and one DSP48E slice.

The synthesis results for area optimization are given in Tables 6 and 7. The whole design now only occupies 31 slices on a Virtex-5 and still runs at 286 MHz instead of 259 MHz formerly. Compared to the ReConFig 2010 architecture, we

Table 7: Proposed design: optimized for area.

	Slices	FFs	LUTs	BRAMs	DSP48E
Floating point converter	13	11	26	—	—
LUT evaluator	18	47	7	1	1
Complete design	31	85	34	1	1

therefore consume about 20% more area by achieving a speedup of about 10% at a higher precision.

5.2. Quality Tests. Quality testing is an important part in the creation of a random number generator. Unfortunately, there are no standardized tests for nonuniform random number generators. Thus, for checking the quality of our design, we proceed in three steps: in the first step, we test the floating point uniform random number converter, and then we check the nonuniform random numbers (with a special focus on the normal distribution here). Finally, the random numbers are tested in two typical applications: an option pricing calculation with the Heston model [38] and the simulation of the bit error rate and frame error rate of a duo-binary turbo code from the WiMax standard.

5.2.1. Uniform Floating Point Generator. We have already elaborated on the widely used TestU01 suite for uniform random number generators in Section 2.1. TestU01 needs an equivalent fixed point precision of at least 30 bits, and for the big crush tests even 32 bits. The uniformly distributed floating point random numbers have been created as described in Section 4 with a mantissa of 31 bits from the output of a Mersenne Twister MT19937 [7].

The three test batteries small crush, crush, and big crush have been used to test the quality of the floating point random number generator. The Mersenne Twister itself is known to successfully complete all except two tests. These two tests are linear complexity tests that all linear feedback shift-register and generalized feedback shift-register-based random number generators fail (see [13] for more details). Our floating point transform of Mersenne random numbers also completes all but the specific two tests successfully. Thus, we conclude that our floating point uniform random number generator preserves the properties of the input generator and shows the same excellent structural properties.

For computational complexity reasons, for the following tests, we have restricted the output bit width of the floating point converter software implementation to 23 bits. The resolution is lower than the fixed point input in some regions, whereas in other regions a higher resolution is achieved. Due to the floating point representation, the regions with higher resolutions are located close to zero. Figure 6 shows a zoomed two-dimensional plot of random vectors produced by our design close to zero. It is important to notice that no patterns, clusters, or big holes are visible here.

Besides the TestU01 suite, the equidistribution of our random numbers has also been tested with several variants of the frequency test mentioned by Knuth [15]. While checking the uniform distribution of the random numbers up to 12 bits, no extreme P value could be observed.

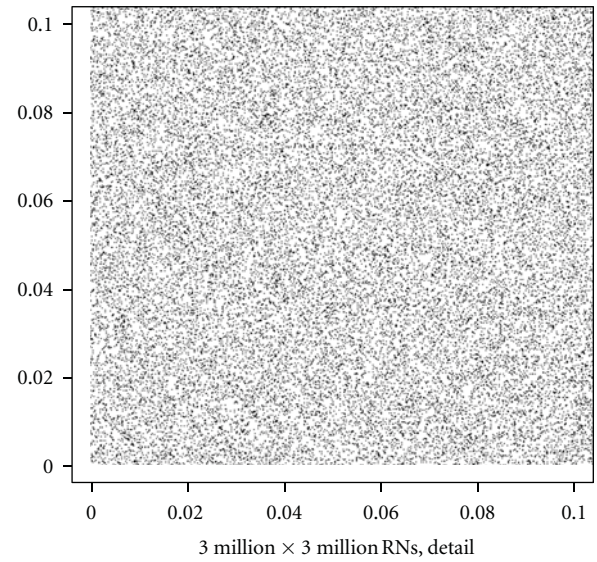

Figure 6: Detail of uniform 2D vectors around 0.

5.2.2. Nonuniform Random Number Generator. For the nonuniform random number generator, we have selected a specific set of commonly applied tests to examine and ensure the quality of the produced random numbers. In this paper, we focus on the tests performed for normally distributed random numbers, since those are most commonly used in many different fields of applications. Also the application tests presented below use normally distributed random numbers.

As a first step, we have run various χ^2-tests. In these tests, the empirical number of observations in several groups is compared with the theoretical number of observations. Test results that would only occur with a very low probability indicate a poor quality of the random numbers. This may be the case if either the structure of the random numbers does not fit to the normal distribution or if the numbers show more regularity than expected from a random sequence. The batch of random numbers in Figure 7 shows that the distribution is well approximated. The corresponding χ^2-test with 100 categories had a P value of 0.4.

The Kolmogorov-Smirnov test compares the empirical and the theoretical cumulative distribution function. Nearly all tests with different batch sizes were perfectly passed. Those not passed did not reveal an extraordinary P value. A refined version of the test, as described in Knuth [15] on page 51, sometimes had low P values. This is likely to be attributed to the lower precision in some regions of our random numbers, as the continuous CDF can not be perfectly approximated with random numbers that have fixed gaps. Other normality tests were perfectly passed, including the Shapiro-Wilk [39] test. Stephens [40] argues that the latter one is more suitable

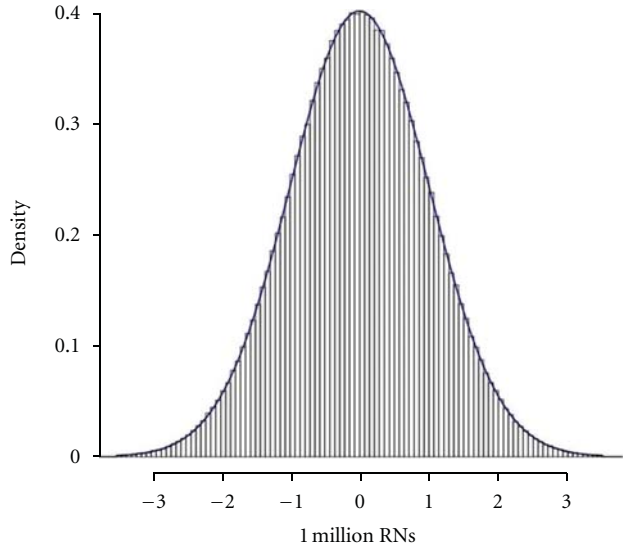

Figure 7: Histogram of Gaussian random numbers.

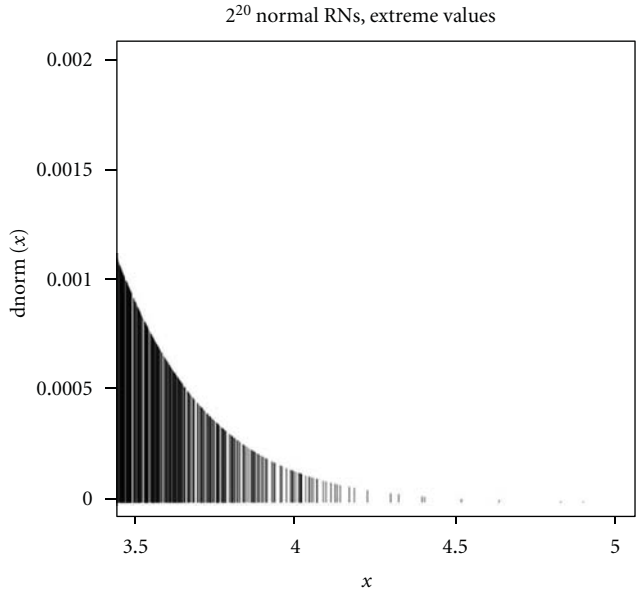

Figure 8: Tail of the empirical distribution function.

for testing the normality than the Kolmogorov-Smirnov test. The test showed no deviation from normality.

We not only compared our random numbers with the theoretical properties, but also with those taken from the well-established normal random number generator of the *R* language. It is based on a Mersenne Twister as well. Again, we used the Kolmogorov-Smirnov test, but no difference in distribution could be seen. Comparing the mean with the *t*-test and the variance with the *F*-test gave no suspicious results. The random numbers of our generator seem to have the same distribution as standard random numbers, with an exception of the reduced precision in the central region and an improved precion in the extreme values. This difference can be seen in Figures 8 and 9. Both depict the empirical results of a draw of 2^{20} random numbers, the first with the presented algorithm and the second with the RNG of *R*.

The tail distribution of the random numbers of the presented algorithm seems to be better in the employed test set. The area of extreme values is fitted without large gaps in contrast to the *R* random numbers. The smallest value from our floating point-based random number generator is $1 \cdot 2^{-54}$, compared to $1 \cdot 2^{-32}$ in standard RNGs, thus values of -8.37σ and 8.37σ can be produced. Our approximation of the inverse cumulative distribution function has an absolute error of less than $0.4 \cdot 2^{-11}$ in the achievable interval. Thus, the good structural properties of the uniform random numbers can be preserved. Due to the good properties of our random number generator, we expect it to perform well in the case of a long and detailed approximation, where rare extreme events can have a huge impact (consider risk simulations for insurances, e.g.).

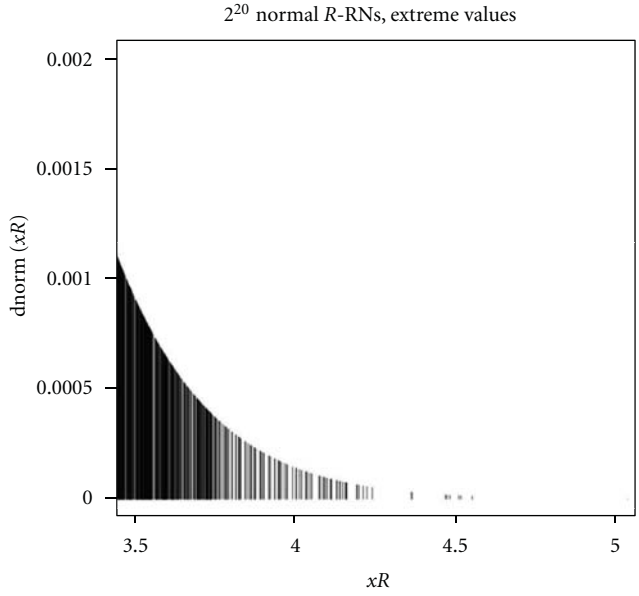

Figure 9: Tail of the empirical distribution function for the *R* RNG.

5.2.3. Application Tests. Random number generators are always embedded in a strongly connected application environment. We have tested the applicability of our normal RNG in two scenarios: first, we have calculated an option price with the Heston model [38]. This calculation was done using the Monte Carlo simulation written in Octave. The provided RNG of Octave *randn*() has been replaced by a bit true model of our presented hardware design. For the whole benchmark set, we could not observe any peculiarities with respect to the calculated results and the convergence behavior of the Monte Carlo simulation. For the second application, we have produced a vast set of simulations of a wireless communications system. For comparison to our RNG, a Mersenne Twister and inversion using the Moro approximation [33] has been used. Also in this test,

no significant differences between the results from both generators could be observed.

6. Conclusion

In this paper, we present a new refined hardware architecture of a nonuniform random number generator for arbitrary distributions and precision. As input, a freely selectable uniform random number generator can be used. Our unit transforms the input bit vector into a floating point notation before converting it with an inversion-based method to the desired distribution. This refined method provides more accurate random numbers than the previous implementation presented at ReConFig 2010 [1], while occupying roughly the same amount of hardware resources.

This approach has several benefits. Our new implementation saves now more than 48% of the area on an FPGA compared to state-of-the-art implementations, while even achieving a higher output precision. The design can run at up to 398 MHz on a Xilinx Virtex-5 FPGA. The precision itself can be adjusted to the users' needs and is mainly independent of the output resolution of the uniform RNG. We provide a free tool allowing to create the necessary look-up table entries for any desired distribution and precision.

For both components, the floating point converter and the ICDF lookup unit, we have presented our hardware architecture in detail. Furthermore, we have provided exhaustive synthesis results for a Xilinx Virtex-5 FPGA. The high quality of the random numbers generated by our design has been ensured by applying extensive mathematical and application tests.

Acknowledgment

The authors gratefully acknowledge the partial financial support from the Center for Mathematical and Computational Modeling $(CM)^2$ of the University of Kaiserslautern.

References

[1] C. de Schryver, D. Schmidt, N. Wehn et al., "A new hardware ecient inversion based random number generator for non-uniform distributions," in *Proceedings of the International Conference on Recongurable Computing and FPGAs (ReConFig '10)*, pp. 190–195, December 2010.

[2] R. C. C. Cheung, D.-U. Lee, W. Luk, and J. D. Villasenor, "Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 8, pp. 952–962, 2007.

[3] P. L'Ecuyer, "Uniform random number generation," *Annals of Operations Research*, vol. 53, no. 1, pp. 77–120, 1994.

[4] N. Bochard, F. Bernard, V. Fischer, and B. Valtchanov, "True-randomness and pseudo-randomness in ring oscillator-based true random number generators," *International Journal of Reconfigurable Computing*, vol. 2010, article 879281, 2010.

[5] H. Niederreiter, "Quasi-Monte Carlo methods and pseudo-random numbers," *American Mathematical Society*, vol. 84, no. 6, p. 957, 1978.

[6] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, Society for Industrial Mathematics, 1992.

[7] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[8] M. Matsumoto, Mersenne Twister, http://www.math.sci.hiroshima-u.ac.jp/?m-mat/MT/emt.html, 2007.

[9] V. Podlozhnyuk, Parallel Mersenne Twister, http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/MersenneTwister/doc//MersenneTwister.pdf, 2007.

[10] S. Chandrasekaran and A. Amira, "High performance FPGA implementation of the Mersenne Twister," in *Proceedings of the 4th IEEE International Symposium on Electronic Design, Test and Applications (DELTA '08)*, pp. 482–485, January 2008.

[11] S. Banks, P. Beadling, and A. Ferencz, "FPGA implementation of Pseudo Random Number generators for Monte Carlo methods in quantitative finance," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 271–276, December 2008.

[12] X. Tian and K. Benkrid, "Mersenne Twister random number generation on FPGA, CPU and GPU," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 460–464, August 2009.

[13] P. L'Ecuyer and R. Simard, "TestU01: a C library for empirical testing of random number generators," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, 22 pages, 2007.

[14] G. Marsaglia, Diehard Battery of Tests of Randomness, http://stat.fsu.edu/pub/diehard/, 1995.

[15] D. E. Knuth, *Seminumerical Algorithms, Volume 2 of The Art of Computer Programming*, Addison-Wesley, Reading, Mass, USA, 3rd edition, 1997.

[16] B. D. McCullough, "A review of TESTU01," *Journal of Applied Econometrics*, vol. 21, no. 5, pp. 677–682, 2006.

[17] A. Rukhin, J. Soto, J. Nechvatal et al., "A statistical test suite for random and pseudorandom number generators for cryptographic applications," http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/ SP800-22rev1a.pdf, Special Publication 800-22, Revision 1a, 2010.

[18] G. B. Robert, Dieharder: A Random Number Test Suite, http://www.phy.duke.edu/?rgb/General/dieharder.php, Version 3.31.0, 2011.

[19] D. B. Thomas, W. Luk, P. H. W. Leong, and J. D. Villasenor, "Gaussian random number generators," *ACM Computing Surveys*, vol. 39, no. 4, article 11, 2007.

[20] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958.

[21] A. Ghazel, E. Boutillon, J. L. Danger et al., "Design and performance analysis of a high speed AWGN communication channel emulator," in *Proceedings of the IEEE Pacific Rim Conference*, pp. 374–377, Citeseer, Victoria, BC, Canada, 2001.

[22] D.-U. Lee, J. D. Villasenor, W. Luk, and P. H. W. Leong, "A hardware Gaussian noise generator using the box-muller method and its error analysis," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 659–671, 2006.

[23] G. Marsaglia and W. W. Tsang, "The ziggurat method for generating random variables," *Journal of Statistical Software*, vol. 5, pp. 1–7, 2000.

[24] G. Zhang, P. H. W. Leong, D.-U. Lee, J. D. Villasenor, R. C. C. Cheung, and W. Luk, "Ziggurat-based hardware gaussian random number generator," in *Proceedings of the International*

Conference on Field Programmable Logic and Applications (FPL '05), pp. 275–280, August 2005.

[25] H. Edrees, B. Cheung, M. Sandora et al., "Hardware-optimized ziggurat algorithm for high-speed gaussian random number generators," in *Proceedings of the International Conference on Engineering of Recongurable Systems & Algorithms (ERSA '09)*, pp. 254–260, July 2009.

[26] N. A. Woods and T. Court, "FPGA acceleration of quasi-monte carlo in finance," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 335–340, September 2008.

[27] C. S. Wallace, "Fast pseudorandom generators for normal and exponential variates," *ACM Transactions on Mathematical Software*, vol. 22, no. 1, pp. 119–127, 1996.

[28] C. S. Wallace, MDMC Software—Random Number Generators, http://www.datamining.monash.edu.au/software/random/index.shtml, 2003.

[29] D. U. Lee, W. Luk, J. D. Villasenor, G. Zhang, and P. H. W. Leong, "A hardware Gaussian noise generator using the Wallace method," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 8, pp. 911–920, 2005.

[30] R. Korn, E. Korn, and G. Kroisandt, *Monte Carlo Methods and Models in Finance and Insurance*, Financial Mathematics Series, Chapman & Hull/CRC, Boca Raton, Fla, USA, 2010.

[31] L. Devroye, *Non-Uniform Random Variate Generation*, Springer, New York, NY, USA, 1986.

[32] J. A. Peter, An algorithm for computing the inverse normal cumulative distribution function, 2010.

[33] B. Moro, "The full Monte," *Risk Magazine*, vol. 8, no. 2, pp. 57–58, 1995.

[34] D.-U. Lee, R. C. C. Cheung, W. Luk, and J. D. Villasenor, "Hierarchical segmentation for hardware function evaluation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 103–116, 2009.

[35] D.-U. Lee, W. Luk, J. Villasenor, and P. Y. K. Cheung, "Hierarchical Segmentation Schemes for Function Evaluation," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '03)*, pp. 92–99, 2003.

[36] IEEE-SA Standards Board. IEEE 754-2008 Standard for Floating-Point Arithmetic, August 2008.

[37] Free Software Foundation Inc. GSL—GNU Scientic Library, http://www.gnu.org/software/gsl/, 2011.

[38] S. L. Heston, "A closed-form solution for options with stochastic volatility with applications to bond and currency options," *Review of Financial Studies*, vol. 6, no. 2, p. 327, 1993.

[39] S. S. Shapiro and M. B. Wilk, "An analysis-of-variance test for normality (complete samples)," *Biometrika*, vol. 52, pp. 591–611, 1965.

[40] M. A. Stephens, "EDF statistics for goodness of fit and some comparisons," *Journal of the American Statistical Association*, vol. 69, no. 347, pp. 730–737, 1974.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 615824, 10 pages
doi:10.1155/2012/615824

Research Article

Hardware Middleware for Person Tracking on Embedded Distributed Smart Cameras

Ali Akbar Zarezadeh[1] and Christophe Bobda[2]

[1] *Department of Computer Science, University of Potsdam, 14482 Potsdam, Germany*
[2] *Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR 72701, USA*

Correspondence should be addressed to Christophe Bobda, bobda@acm.org

Received 29 April 2011; Accepted 6 December 2011

Academic Editor: Ron Sass

Tracking individuals is a prominent application in such domains like surveillance or smart environments. This paper provides a development of a multiple camera setup with jointed view that observes moving persons in a site. It focuses on a geometry-based approach to establish correspondence among different views. The expensive computational parts of the tracker are hardware accelerated via a novel system-on-chip (SoC) design. In conjunction with this vision application, a hardware object request broker (ORB) middleware is presented as the underlying communication system. The hardware ORB provides a hardware/software architecture to achieve real-time intercommunication among multiple smart cameras. Via a probing mechanism, a performance analysis is performed to measure network latencies, that is, time traversing the TCP/IP stack, in both software and hardware ORB approaches on the same smart camera platform. The empirical results show that using the proposed hardware ORB as client and server in separate smart camera nodes will considerably reduce the network latency up to 100 times compared to the software ORB.

1. Introduction

Smart cameras are embedded systems for performing image analysis directly at the sensor, and thus delivering description of scene in an abstract manner [1]. Networks of multiple smart cameras can be used to efficiently implement features like detection accuracy, fault tolerance, and robustness that would not have been possible with a single camera in such applications like surveillance or smart environments. In such networks, smart camera nodes are usually embedded systems with very tight design and operation requirements. Besides complex computations that must take place in real time, communication among the cameras has to be handled as well. In embedded environment, the complex computation can efficiently be handled using a combination of hardware and software. While the hardware handles the most computational demanding tasks, the software takes care of the control parts. By using FPGA as the main computational component, complex computation can be directly implemented in hardware, while the control part can be carried out by an embedded processor. We used this approach in the design of an FPGA-based embedded smart camera platform [2]. Also, in our previous works [3, 4], we proposed a software/hardware codesign middleware approach, called hardware ORB. The motivation for the hardware ORB was to provide a low-latency intercommunication with deterministic behavior among the smart camera nodes. The middleware enables developers to program distributed applications seamlessly, without having to focus on the details of the implementation, thus reducing the development time [5].

The middleware provides designers to transparently describe distributed systems. A lightweight agent-oriented middleware for distributed smart cameras was introduced by Quaritsch in his PhD thesis [1]. His middleware implements basic networking and messaging functionality for collaborative image processing. There are some existing hardware implementations of ORB which are exploited in the area of software-defined radio (SDR) systems. The integrated circuit ORB (ICO) engine [6, 7] implements a hardware CORBA ORB. PrismTech's ICO is such an ORB in SDR applications.

The ICO engine is responsible for implementing the transfer syntax used in CORBA messages.

According to the network performance analysis, the Linux kernel must be manipulated to collect timing measurements on internals. The data streams kernel interface (DSKI) instrumentation software was made for Linux [8]. It supports gathering of system behavioral data from instrumentation points inserted in kernel and application source. Demter et al., in [9], demonstrated the performance analysis of the TCP/IP stack for the Linux kernel, and then the time traversing of packets for each layer of the Linux kernel TCP/IP stack was analyzed. Their idea to achieve network performance analysis inspired us to follow the same methodology with some modification for extraction of the network latencies.

Adequate coverage of the critical areas is not feasible with only one camera. Inevitably, multiple cameras must observe an *overlapping* scene from different views. It provides robustness against occlusion and improves accuracy [10]. The observations of an object on different captured video streams must be identified as the same object. It is of crucial importance to determine the correspondence between different views. Khan and Shah [10] introduced a pure geometry-based approach without need for camera calibration parameters. Calderara et al. [11] proposed another class of geometry-based approach called homography and epipolar-based consistent labeling for outdoor park surveillance (HECOL). As the highlighted benefit, their method identifies people in groups by leveraging the probabilistic Bayesian theory.

In [3], we proposed Common object Request Broker Architecture (CORBA) in the compact profile as a software middleware. Additionally, the hardware ORB for server side was proposed to overcome the limitations of the software ORB running on an embedded platform in real-time applications. The concept of the hardware ORB was extended in two steps in [4]: (i) a new client hardware ORB as a pair of the sever hardware ORB and (ii) performance analysis of both hardware and software approaches for systematic measurement of the latencies.

This paper adds a distributed multicamera tracker layer with jointed field of view (FoV) on top of our hardware ORB middleware. Tracking being a key part of high-level visual algorithms, a wide variety of applications, like surveillance, human motion analysis, and traffic monitoring, can benefit from the results provided in this paper. The simultaneous association process in an overlapped camera setup demands twofold constraints: (i) a real-time stand-alone tracker at each camera node and (ii) tight data exchange mechanism among multiple cameras with deterministic behavioral. The existing works mostly concentrate on the only software implementations of vision algorithms and communication modules, leaving the optimal hardware/software implementation, in particular in the embedded system open. By presenting such typical vision application, first our novel system-on-chip (SoC) architecture of a real-time stand-alone tracker based on an FPGA-based smart camera is demonstrated. It functions via segmentation process (object detection) and object association (identification) steps. Segmentation algorithm demands an impressive amount of computational power. By hardware acceleration of the segmentation, we achieve an efficient implementation which guaranties a real-time performance. Second, we prove the viability and versatility of our proposed communication infrastructure based on the hardware ORB. Considering the parallelized internal nature of our hardware ORB, it appropriately satisfies such constraints dictated by the upper application layer.

The rest of the paper is organized as follows: first, an explanation of the methodology for finding the correspondence of an object which appears simultaneously on multiple overlapping cameras is given in Section 2. In Section 3, an architectural view of the proposed system inside the smart camera is presented. Afterwards, Section 4 is dedicated to explain the protocol stack performance analysis on the basis of probing strategy implementation. Section 5 is devoted to the experimental results. It reveals the obtained empirical results. Finally, the paper is briefly concluded in Section 6.

2. Tracker System

The focus of this section is the multiperson tracking in an overlapping multi-camera setup. The tracking system classifies all the observations of the same person, grabbed from different cameras, into trajectories. In the case of failure in a camera node, the tracking process can uninterruptedly be continued with other cameras, and trajectories of persons from various views can be merged [12]. Regarding the occlusion of the target person, the trajectories still can be kept updated in all cameras via exchanging the extracted abstract features in multiple cameras. The usage of multiple cameras requires keeping consistent association among the different views of single persons. Put simply, it assigns the same label to different instances of the same person acquired by different cameras. This task is known as consistent labeling [10, 11]. We divide the whole procedure into two off-line and on-line phases. At the off-line phase by reidentification of a known person who moves in the camera FoVs, we extract the FoV lines and homography matrix by means of multiview geometry computations. Using the determined hypothesis from the off-line phase, we find the correspondence between different simultaneous observations of the same person in on-line phase.

2.1. Off-Line Phase. The off-line process aims at computing overlapping constraints among camera views to create the homography. This process applies for each pair of overlapped cameras. It performs the computation of the entry edges of field of view (EEoFoV). As the upper-left side of Figure 1 shows, two adjacent cameras C^1 and C^2 have partially overlapping view scenes. Our goal is to extract the FoV lines associated to the C^2 in C^1 and vice versa. The lower-left side of Figure 1 depicts the defined overlapping zones $Z^{1,2}$ and $Z^{2,1}$ between the cameras C^1 and C^2, respectively, assuming a single person walks through the FoVs of the C^1 and C^2 and crosses the EEoFoVs of each individual camera. Once the person appears in the FoV of a camera, an stand-alone tracker is allocated to him/her.

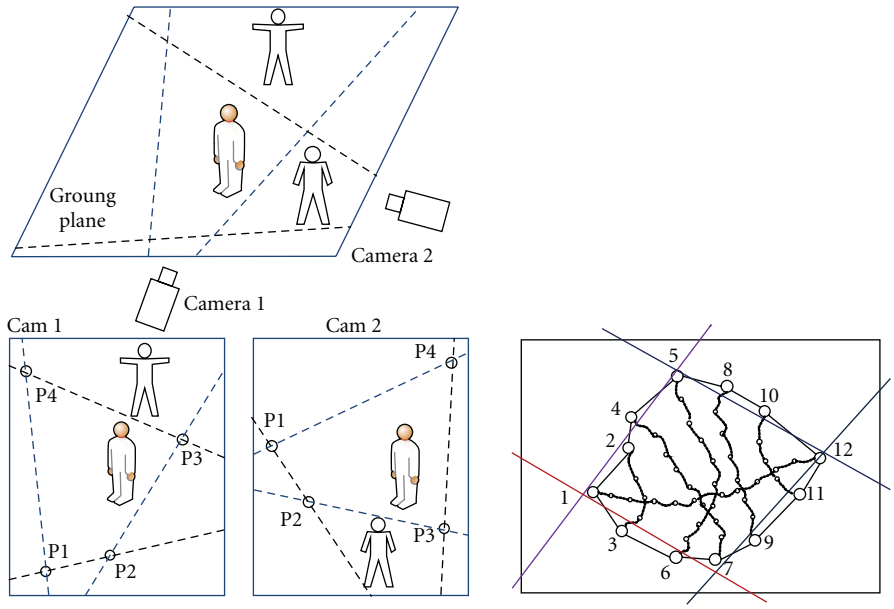

Figure 1: Detecting the overlapping zones based on FoV lines (left), and gathered person trajectories in off-line phase (right).

In each new scan of the scene, the tracker keeps the other neighbor camera updated about new position (lower support point) of the person. Integration of all the positions represents the person's trajectory.

As illustrated in the right side of Figure 1, after gathering adequate number of trajectories, we collect all points which lay on the EEoFoVs and establish a set as $\Gamma^{1,2} = \{p_1^{1,2}, p_2^{1,2}, \ldots, p_{12}^{1,2}\}$, where the subscripts indicate corresponding points in the two cameras. Afterwards, we elicit a polygon, which encompasses all points $p_i^{1,2} \in \Gamma^{1,2}$ with the biggest covered area. In order to compute this polygon, we need to find out if two points in the set $\Gamma^{1,2}$ are adjacent. Our strategy consists of sorting out this set of points based on the $p_i^{1,2}$ of the X and Y coordinates. First, we sort out all the $p_i^{1,2}$ points based on the X values and then construct a new set of points called $\Omega^{1,2}$. Now, to each element of the $\Omega^{1,2}$ a label equal to the element's position at the set is assigned. Afterwards, we sort out the $\Omega^{1,2}$ considering the elements' Y values and then form a new set of points referred to as $\Theta^{1,2}$. Figure 2 shows a 2D representation of the *EEoFoV* points where each column is the $\Theta^{1,2}$ (sorting based on Y), and each element of the column is the label assigned from sorting based on $X(\Omega^{1,2})$. Let us start from the label 1 (point with the lowest X value). At the first column from the position of label 1, we look for its nearest adjacent point based on the minimum Euclidean distance in the direction toward the bigger Y values. The point with label 3 has the minimum Euclidean distance. We continue this procedure up to the last element of the list with the highest Y value. Afterwards, we change the direction backward from the highest to the lowest Y values. After several iterations, all edges are computed.

Figure 3 shows the final computed graph where all vertexes are the EEoFoV points sorted based on the X values and

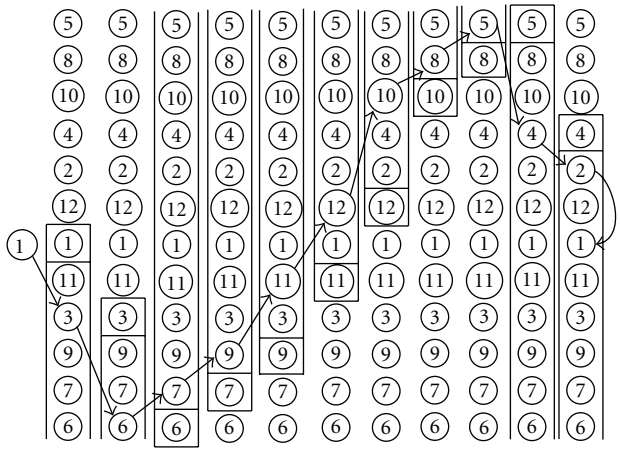

Figure 2: Arrangement of gathered *EEoFoV* points based on X and Y coordinate sorting.

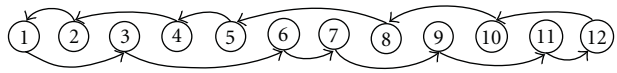

Figure 3: Final computed graph representing the extracted polygon.

all edges are results of previous iterations. Hence, the polygon is extracted.

According to the sign alteration of the slope of lines, connecting the adjacent points (the right side of Figure 1), we classify all the points into four groups. Next, by means of linear regression, the best line on each group is fitted, and thus, all FoV lines are computed. Intersection points between

the four lines in the cameras C^1 and C^2 (the four corners of the $Z^{1,2}$ and $Z^{2,1}$) are the best candidates for computation of homography matrix [11]. The homography matrix $H^{1,2}$ from camera C^1 to camera C^2 is formulated as

$$\begin{bmatrix} x_k^{1,2} \\ y_k^{1,2} \\ 1 \end{bmatrix} = \begin{bmatrix} h_{1,1}^{1,2} & h_{1,2}^{1,2} & h_{1,3}^{1,2} \\ h_{2,1}^{1,2} & h_{2,2}^{1,2} & h_{2,3}^{1,2} \\ h_{3,1}^{1,2} & h_{3,2}^{1,2} & h_{3,3}^{1,2} \end{bmatrix} \times \begin{bmatrix} x_k^{2,1} \\ y_k^{2,1} \\ 1 \end{bmatrix}. \quad (1)$$

2.2. On-Line Phase. At the on-line process, once a camera node detects a new object, it sends the person's position together with an accompanying wall clock time stamp to all overlapped neighboring nodes. Hence, each node has its own local knowledge of the detected person and also the external information received from its neighbors. The correspondence between the local and external information must be accomplished. Based on the computed FoV lines at the off-line phase, we determine whether the detected person is in the overlapping zone or not. If this condition is asserted, with using the computed ground-plane homography matrix, we convert the person's locally extracted lower support point from the local coordinate to the external coordinate corresponding to the neighbor camera node. By using the temporal information (time stamp) and minimum Euclidean distance between the spatial data (e.g., distance between position of the person in local and external observations), we infer the correspondence. Provided that the person is only in the FoV of one camera, the history of that individual is applied to determine whether he is a new person or the one who has been observed previously.

The on-line process must be performed repeatedly for every acquired new video frame at all cameras. Thus, it imposes two key constraints on the entire design: (i) a real-time stand-alone tracker at each camera node and (ii) a real-time data exchange mechanism among multiple cameras. Next Section 3 covers the architectural aspect of our implemented system inside the smart camera to tackle processing and communication tasks.

3. Smart Camera Internal Architecture

The focus of this section is the structure of hardware/software inside the FPGA, which is the computing element inside our smart cameras. To have an efficient partitioning of the hardware/software, we use different processes for implementation of the communication and computer vision tasks. Our architecture consists of two processes: *computer vision module* (CVM) and *communication module* (CM).

3.1. Computer Vision Module. The *CVM* is a framework to launch the upper intelligence layer, which contains all the required computer vision algorithms for processing the captured raw video stream. As Figure 4 shows, it consists of two threads: *serverPeer_Thread* (SPT) and *ClientPeer_Thread* (CPT).

The *CPT* listens to the receiving events from the *CM* process inside the node. Based on the initial configuration, it

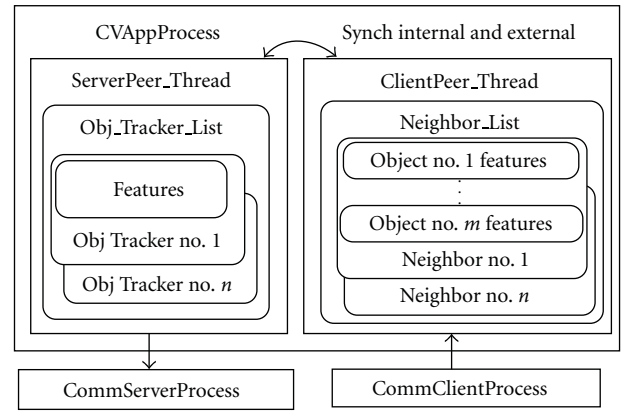

Figure 4: *Computer vision processor* internal architecture.

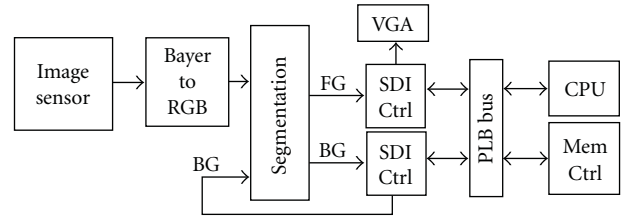

Figure 5: The hardware accelerated segmentation module.

generates a neighbor list, which includes all the node's neighbors inside the cluster. At run time, this list keeps track of persons for a particular neighbor node. The *SPT* is designed to get query from the image sensor and subsequently running upper layer computer vision algorithms cyclically. In each cycle, it runs a locally stand-alone multiperson tracker for detection and tracking purposes. Segmentation plays explicitly a basic role for labeling the pixels as foreground (FG) or background (BG) pixels. The segmentation is performed by the method proposed by Horprasert et al. [13]. To gain real-time performance in such a low level pixel processing unit, the segmentation process is totally offloaded into the hardware. Figure 5 shows the SoC for the segmentation process.

The SoC design presents a pipelined chain of IP cores from grabbing the raw image, conversion from the Bayer to RGB patterns, segmenting, and streaming data flow. The integrated algorithm inside the segmentation module compares an initially trained BG image to the current image. The decision for each pixel, whether an FG or BG, is made based on the color and brightness distortions properties. Two streaming data interface (SDI) controllers are employed for handling the streaming data flow between a given individual IP core and memory [14].

The output FG image of the hardware segmentation module is readily accessible from the software tracker in a preassigned block of memory. The tracker exploits the FG image to detect the moving persons in the scene. A new instance of the tracker is instantiated for each new person in the scene. Each instance holds the extracted features for an individual person. The *SPT* participates on finding

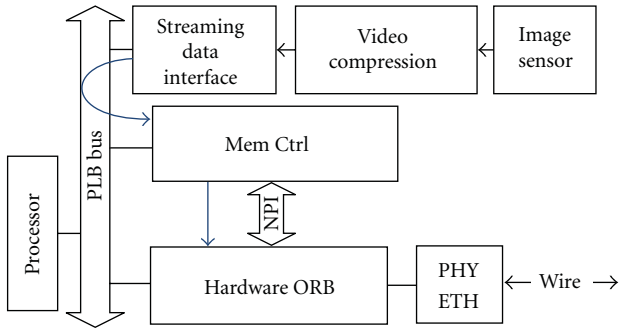

Figure 6: The usage of the hardware ORB for video streaming.

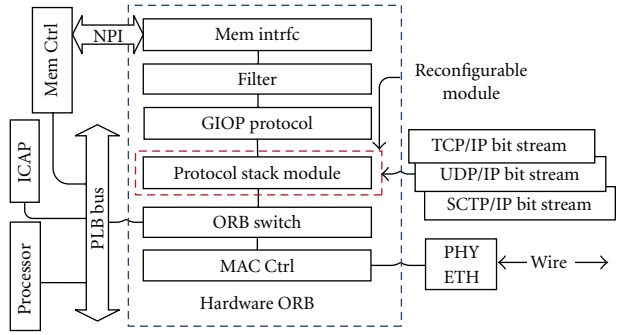

Figure 7: The general architecture of the hardware ORB IP core.

the association between its local person's features and the external person's features kept by the *CPT*. The mechanism for the discovery of such a correspondence is based on the tracking algorithm explained in Section 2.

3.2. Communication Module. The *CM* acts as the output/input gates for pushing/pulling the locally generated events by the *CVM* to the cluster of cameras. It has an interconnection with the *CVM* through some of interprocess facilities, for example, shared memory. The hardware ORB is the core engine of the *CM*. As a CORBA-based middleware IP core, the hardware ORB allows clients in a real-time manner to invoke operations on distributed objects without concern about communication protocols. The hardware ORB server is responsible for replying determined requests of the CORBA client hosted in a GPP or another smart camera platform. Hence, a predefined block of memory can be used as a CORBA servant object. Figure 6 shows the usage of the hardware ORB as a video streaming solution for the acquired video footage. In this case, it is supposed that the compressed information of video stream resides in a determined location of memory. Therefore, upon the request of a client node in another smart camera node, the hardware ORB fetches such processed data from memory and then directly serializes and transmits it through the protocol stack towards the client node. Figure 7 shows the overall anatomy of the hardware ORB IP core. The most considerable feature of such architecture is to have a redundant offloaded protocol stack in parallel operation with the embedded processor. Its main advantage is to have the possibility of using the network services on the operating system and also utilizing another software ORB running on the embedded processor. Besides the usage of the software ORB, the hardware ORB plays the role of an auxiliary deterministic and real-time performance communication engine. Considering Figure 7, the *ORB switch* shares the MAC layer between the processor and the hardware ORB. Basically, the hardware ORB can exchange the contents of blocks of the memory through general inter-ORB protocol (GIOP), without the usage of the embedded processor. It implements the lower layers of the standard TCP/IP stack and also supports the direct access to DDR memory with the native port interface (NPI) of multiport memory controller (MPMC). Figure 8 shows the internal architecture of the IP core as client/server node in more details. To initiate a new transaction in the client side, the embedded processor must activate the transmission path by writing in a specific register of the IP core. Then, based on the defined responsibilities of GIOP such as common data representation (CDR) and interoperable object reference (IOR), it constructs the *request* message format and sends it down through the protocol stack. Afterwards, there are two separate transmission buffers, one for the embedded processor and another for the hardware ORB, called *Tx buffer* and *hard ORB Tx buffer*, respectively. The *ORB switch* connects one of the two buffers to the *MAC Tx controller* for transmission.

The hardware ORB's server mechanism is semantically similar to the client's one. Once the server receives a request packet from the Rx path of the protocol stack, if the packet satisfies the filter criteria (i.e., the predefined TCP port number and CORBA interoperable object reference), the server fetches the relevant data from memory by a module, called *reader controller*. Then the hardware ORB constructs the TCP/IP packet and finally stores it inside the *hard ORB Tx buffer* for transmission via the *MAC Tx controller*.

Once the client receives the *reply* from the server, the replied packet comes up through the receiving path of the hardware ORB and finally is stored in the memory by means of the *writer controller*. Moreover, the embedded processor is connected to the TX and RX dual port buffers of the MAC layer (*Tx buffer* and *Rx buffer*) by means of the PLB slave interface, and it enables the protocol stack on Linux. Referring to Figure 8, once a new packet arrives from the *MAC Rx controller*, one copy is placed inside the *Rx buffer* of the embedded processor and routed at the same time to the Rx path of the protocol stack of the hardware ORB. The hardware ORB listens to the incoming packets and looks for the specific ones with a predefined TCP port number and CORBA IOR. If the incoming packet satisfies the filter criteria, first, with the masking of the Rx interrupt to the embedded processor, the hardware ORB blocks the interface to the embedded processor. It avoids racing between the hardware ORB and the embedded processor. Afterwards, the hardware ORB copies the received data to DDR memory by a module called *writer controller*.

4. Protocol Stack Performance Analysis

In this section, the performance analysis methodology on the basis of probing strategy is explained. We use the terminology of [15] to characterize the performance parameters

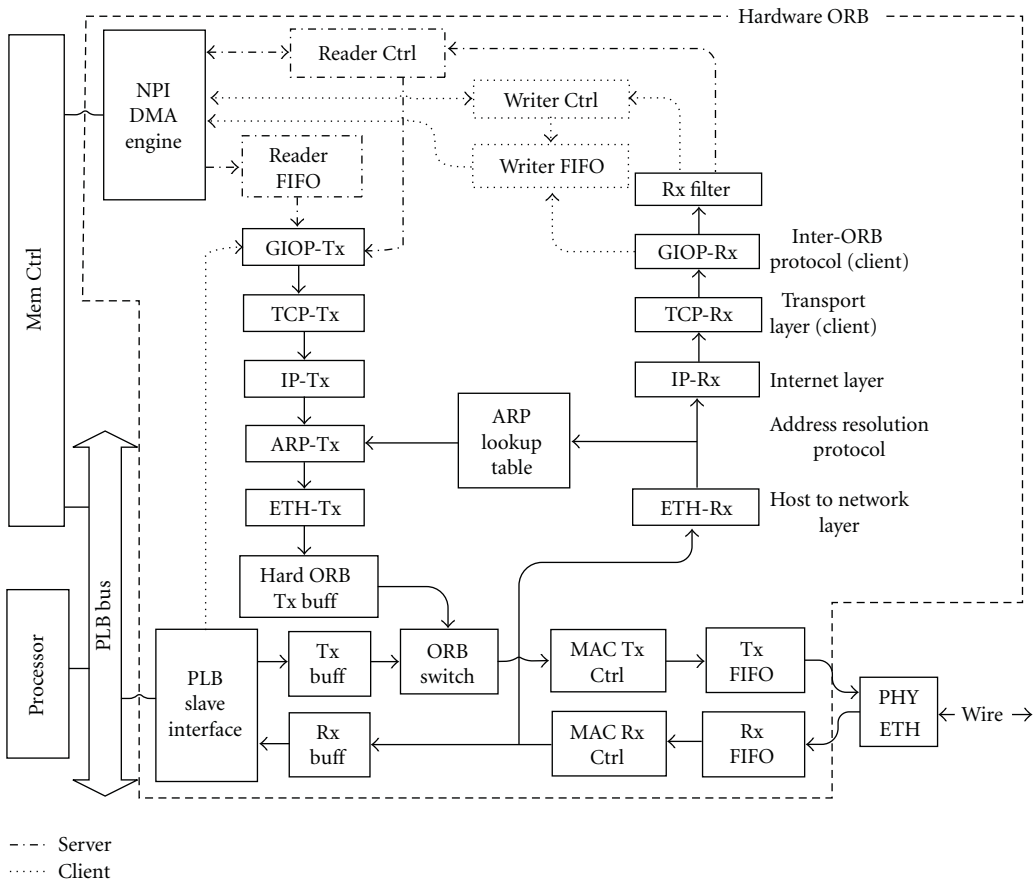

Figure 8: The architecture of the hardware ORB client/server IP core.

(i.e., time of flight, sending/receiving overheads, injection/reception, transmission time, and bandwidth). For a network with only two devices, the bandwidth for *injection* (sending) is yielded by dividing the *Packet size* upon the maximum of the *sending overhead* and *transmission time*. The similar calculation can be done for the bandwidth of *reception* by substitution of the *receiving overhead* instead of the *sending overhead*. Total data bandwidth supplied by network is defined as *aggregate bandwidth*. The fraction of aggregate bandwidth that gets delivered to the application is *effective bandwidth* or *throughput* [15],

$$\text{Eff.bandwidth} = \min\left(2 \times BW_{\text{Link Injection}}, 2 \times BW_{\text{Link Reception}}\right). \quad (2)$$

The goal of performance analysis is to determine the underlying metrics such as *sending* and *receiving overheads* and the *time of flight*. It is of particular interest to investigate the protocol stack traversal times (GIOP/TCP/IP/ETH). In Sections 4.1 and 4.2, we explain our methodology for probing the software and hardware ORB approaches, respectively.

4.1. Software ORB Probing Strategy. Figure 9 illustrates the internal mechanism for extraction of the protocol stack traversal times where two FPGA-based smart cameras are using

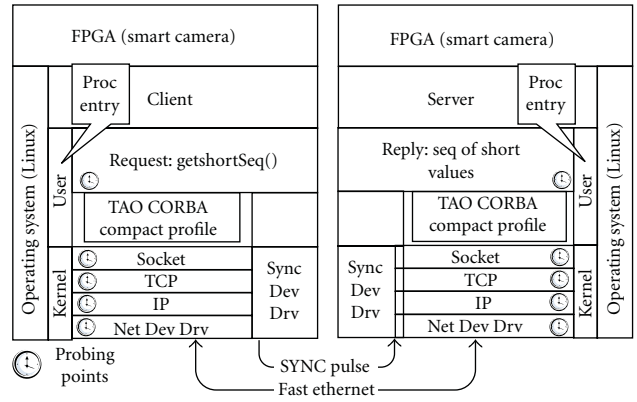

Figure 9: Software ORB experiment setup.

the software ORB. The big challenge here is how to accurately track packets traversing the Linux kernel. To measure packet traversal, an additional time stamp probing code is added to the Linux kernel code, similar to the approach in [9]. As depicted in Figure 9, a variety of probing points at several positions of the kernel's protocol stack (including network device driver, IP layer, TCP layer, and socket layer) are inserted. A unique identification (ID) code is assigned to each probing point position. Later at the postprocessing phase of

performance analysis, those ID codes are used for packet tracking across network protocol stack. The acquired time stamps relevant to each unique ID must be stored and made later accessible in the user space for further analysis. For storage and access at initialization phase, a virtual memory segment is allocated. Anytime the time stamp code inside the Kernel is called, the acquired time stamp together with ID code is written into this segment. To make the measurement results transparent in the user space, a *proc* file entry as a bridge between the kernel and user space is created. The *proc* file is a virtual file residing in the *proc* directory of Linux. This gives the possibility of reading the aforementioned kernel memory segment into the user space. Therefore, based on our definition, each line of this *proc* entry file contains an ID, which determines unique probing point and also a time stamp. Thus, after completing the test, the contents of this *proc* entry are used for further analysis. However, for time synchronization between two nodes (for measurement of *time of flight*), two particular IO pins of FPGAs on both cameras are connected by an external wire. On a certain predetermined point of transaction, client node first records a time stamp and simultaneously asserts a pulse signal to the server node to generate an interrupt there. At that point, the server records a time stamp in its relevant interrupt service routine (*ISR*). In the postprocessing analysis, on the basis of those time reference events, we reach to the same time reference in the client and server measured records. To have an access to this physical IO, another new Linux device driver on both client and server nodes is developed for handling ISR. Also, to have a seamless measurement results in the user and kernel spaces for later postprocessing; again the mentioned device driver is used. In fact, by calling the *ioctl* in the user space, some particular probing points in the kernel space are activated.

4.2. Hardware ORB Probing Strategy. The second probing strategy (Figure 10) presents a realistic interconnection of two smart camera systems with fully utilization of the hardware ORB's client and server. We perform measurements on the FPGA by monitoring some hardware probing points in the state diagram of each protocol stack layer and then starting and stopping a timer which gives us the best results. As Figure 10 shows, inside both client and server nodes, an auxiliary IP core called *measurement tool* is used. It counts precisely the number of clock cycles between the start and stop commands coming from probing points placed in several layers of TCP/IP inside the hardware ORB. Therefore, the consumed time is exactly determined. Also, for measurement of *time of flight* in the same manner like the software ORB probing, particular IO pins of FPGAs on both cameras are connected by an external wire. Once the server node sends the first bit of packet, it commands for starting of a counter inside. When the client receives this first bit, it notifies the server node to stop counting via a pulse on the wire.

5. Experiments

We mounted three video cameras apart in our lab. Two sets of our developed embedded FPGA-based smart camera systems

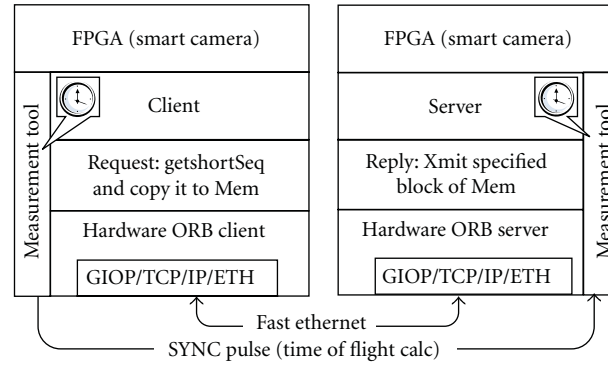

Figure 10: Hardware ORB client and server experiment setup.

Table 1: The FPGA resource usage by the SoC design.

IP Core	Flip-flop	LUT	BRAM
Segmentation	140	479	8
VGA out	73	248	
Bayer 2 RGB	64	75	2
SDI Ctrl 0	1658	1867	2
SDI Ctrl 1	1522	1731	2
Mem Ctrl	8262	7172	19
Hard ORB	3787	4265	2

(PICSy) [2] were interconnected to a desktop PC with logitech Webcam C500. PICSy features a *Virtex4 (Xilinx)* FPGA device with a *PowerPC 405* clocked at 300 MHz. OpenCV was used for implementation of computer vision algorithms. TAO [16] was applied as the middleware for exchanging the abstract information between cameras. Table 1 shows the FPGA resource usage by each IP core in the SoC design. We first program the cameras to identify multiple persons randomly walking on the site. The following sequence was conducted between the adjacent neighboring nodes. When a walking person appeared in the FOV of a camera, the camera generated a new event at each captured video frame. The payload of event was the person's extracted lower support point and the wall clock. The camera cyclically supplied its other neighboring consumer nodes with those features. It was of particular interest to measure the performance of the segmentation module on both smart camera and desktop PC. A PC with Intel Core2Duo running at 3.16 GHz needed an approximately 23 ms to compute the FG for a 640 × 480 sized image. In comparison to the PC, the superiority of the hardware-implemented module was achieved by a factor of ≈7 (≈3.8 ms). Furthermore, the entire procedure from the acquiring one video frame to putting the FG into the memory took ≈34 ms (the pixel clock of the image sensor was 48 MHz). Next, we investigated intercamera communication as follows:

(i) software ORBs running on both cameras (Figure 9);

(ii) hardware ORBs hosted on both cameras (Figure 10).

CORBA TAO client and server ran on both smart cameras having Linux with kernel 2.6.31 and TAO version 1.7.3 as

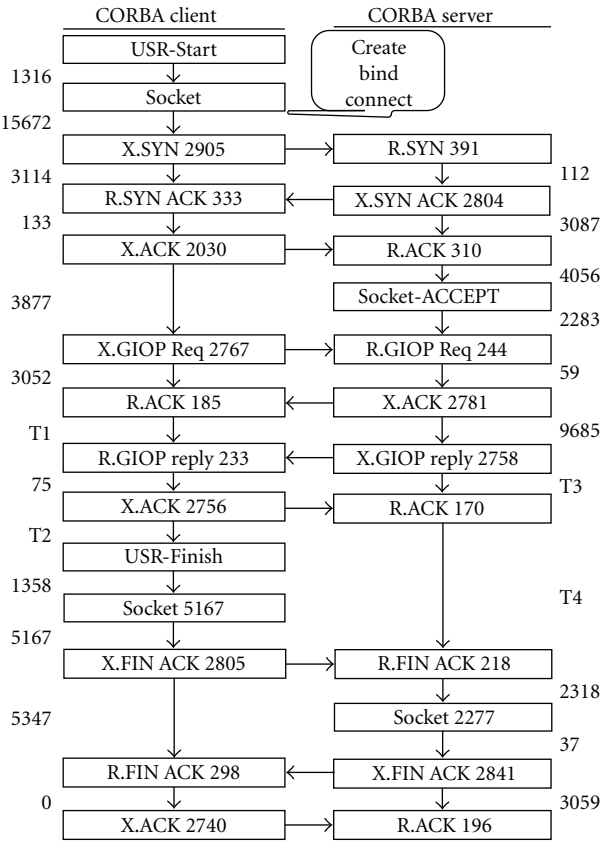

Figure 11: Measured latency values (in μs) from CORBA IIOP transaction state diagram for software ORB.

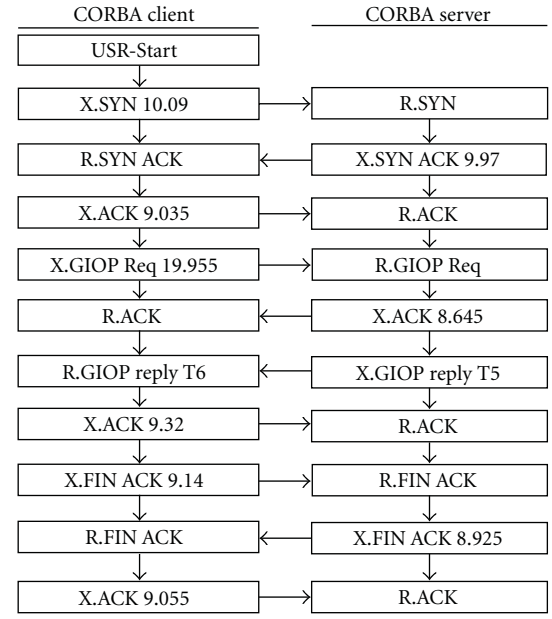

Figure 12: Measured latency values (in μs) from CORBA IIOP transaction state diagram for hardware ORB.

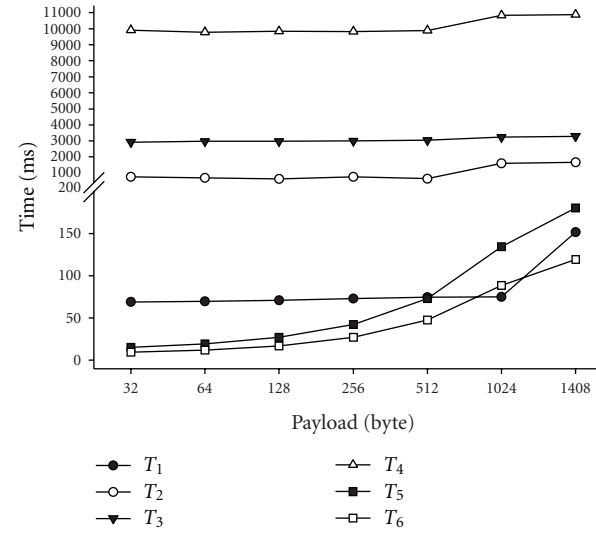

Figure 13: Timing parameters for Figures 11 and 12.

ORB. In both test scenarios, the measurements were achieved with a CORBA client invoking on a CORBA servant the operation *getshortSeq()*, returning a short sequence value as a result of operation. The network nodes were connected to each other directly without any other interfaces. In both benchmarks, the whole progress is repeated for different values of the *ShortSequence* parameter length, doubling the payload size starting from 32 bytes up to 1408 bytes. The tests were performed with the network interface speed of 100 Mbps.

Figures 11 and 12 show the transaction state diagrams in both client and server nodes based on the on-the-wire protocol of CORBA (internet inter-ORB protocol (IIOP)), corresponding to the software ORB and hardware ORB approaches.

Here, the major focus was on the demonstration of the packet processing time when traversing the protocol stack layers. In both Figures 11 and 12, *X.* stands for transmit and *R.* for receive, respectively. There is one number inside of each state, which represents measured processing time (μs) for that particular state. There is another number in Figure 11, related to each two adjacent states, which shows the waiting time for either receiving a special event from the other node or interprocessing time between two consequent states in the same node. Depending on the payload size, there are some variable timing values in both figures, called T_1, T_2, T_3, T_4, T_5, and T_6 which are shown in Figure 13, as well. Considering Figure 11, the whole transaction starts with invoking the operation *getshortSeq()* in the client node in the user space application. It corresponds to *USR-start* state. Afterwards, there are some interactions between the client and server across the TCP state diagram. Finally, *USR-finish* is the point where the application receives the request results. Figure 12 demonstrates the same story for the hardware ORB implementation. But there are some remarkable differences. First, in the software ORB approach, a very costly procedure is placed between *USR-start* state and *X.SYN* in the client side. It consists of the CORBA internal processing

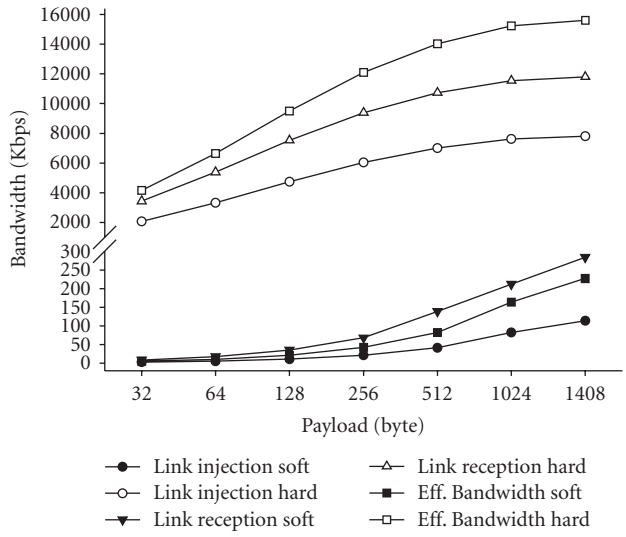

Figure 14: Comparison of latencies between software and hardware ORBs.

tasks in the user space and then coming through the socket operation in the kernel space, and thereafter, starting the TCP state diagram (*X.SYN*). According to our experiments, this segment is very expensive and nondeterministic. The second time-consuming zone is positioned between *X.ACK* and *X.GIOP reply* states in the server side. It constructs the major part of *link injection* time. Finally, the third critical point appears in the client side when delivering the received data to the user space application. On the contrary, the entire client and server transactions in the hardware ORB approach are summarized in the pure TCP state diagram, and it drastically shrinks the overheads. It is interesting to mention that the processing time for the whole transaction is 100 times faster than the software ORB. Furthermore, it should be noted that on the receiving side of each state in the hardware ORB, there are no timing values due to its pipelining nature. Collectively, considering the measurement of the *link injection* and *reception* in both benchmarks, Figure 14 shows the calculated bandwidth based on the different payloads. In terms of the software ORB, the measured time for T_1 on the client side, as a good representation of the server processing time, is used for the *link injection* parameter and summation of T_2 and T_3 for the *link reception*. In the same manner, T_5 and T_6 parameters are used for *link injection* and *link reception* in the hardware ORB approach. As it can be seen in Figure 14, the proposed hardware ORB will greatly increase the network throughput, much better than the software ORB. Additionally, a combination of the hardware ORBs client and server provides a realistic solution for interconnection between two smart cameras, resulting in huge improvement of the network latency, thus guaranteeing a real-time behavior. Judging from the data in this graph, the trend for the software ORB is continuing in the higher payload sizes. It can be easily concluded from Figure 14 that the software ORB is realizable when having higher payload size.

6. Conclusion and Future Research

Through this paper, we demonstrated the concept of a distributed person tracker system as an upper layer on top of the communication layer. The tracker leverages a geometry-based approach to establish correspondence among different views in an overlapped installed camera setup. The SoC design of an FPGA-based smart camera was presented. The design has emerged as a novel hardware/software architecture aimed at enforcing the real-time requirement of image processing applications. Experimental results reveal that the low level pixel processing hardware implemented module outperformed the PC one by the factor of 7. To provide the required communication facilities dictated by upper application layer, that is, distributed person tracker, we provided a real-time hardware/software intercommunication system among smart camera nodes, the hardware ORB which drastically shrinks the network latencies. Moreover, an exhaustive performance analysis was accomplished to accurately track packets traversing inside the Linux kernel. Very outstanding advantage of the hardware ORB in terms of speed was pointed out in favor of using gate-level implementation rather than embedded processor. The outlook of this work includes the model of the person tracker. Because of occurrence of segmentation noise and also instant arrival of group of people to scene, such simple solution is not a stable one. Considering the proposed method by Calderara et al. in [11], such kind of probabilistic models like Bayesian framework is a promising solution to increase the certainty and stability of the tracker system.

References

[1] M. Quaritsch, *A lightweight agent-oriented middleware for distributed smart cameras*, Ph.D. dissertation, Graz University of Technology, 2008.

[2] C. Bobda, A. A. Zarezadeh, F. Mühlbauer, R. Hartmann, and K. Cheng, "Reconfigurable architecture for distributed smart cameras," in *Proceedings of the Engineering of Reconfigurable Systems and Algorithms (ERSA '10)*, pp. 166–178, 2010.

[3] A. A. Zarezadeh and C. Bobda, "Hardware orb middleware for distributed smart camera systems," in *Proceedings of the Engineering of Reconfigurable Systems and Algorithms (ERSA '10)*, pp. 104–116, 2010.

[4] A. A. Zarezadeh and C. Bobda, "Performance analysis of hardware/software middleware in network of smart camera systems," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 196–201, 2010.

[5] The evolution from single to pervasive smart cameras, 2008.

[6] F. Humcke, "Making fpgas first class sca citizens," SDR Forum Technical Conference, 2006.

[7] F. Casalino, G. Middioni, and D. Paniscotti, "Experience report on the use of corba as the sole middleware solution in sca-based sdr environments," SDR Forum Technical Conference, 2008.

[8] "Data streams kernel interface," http://www.ittc.ku.edu/kusp/initial/initial_datastreams.html.

[9] J. Demter, C. Dickmann, H. Peters, N. Steinleitner, and X. Fu, "Performance analyzer of the tcp/ip stack of linux kernel 2.6.9," http://user.informatik.uni-goettingen.de/.

[10] S. Khan and M. Shah, "Consistent labeling of tracked objects in multiple cameras with overlapping fields of view," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 10, pp. 1355–1360, 2003.

[11] S. Calderara, A. Prati, and R. Cucchiara, "HECOL: homography and epipolar-based consistent labeling for outdoor park surveillance," *Computer Vision and Image Understanding*, vol. 111, no. 1, pp. 21–42, 2008.

[12] S. Velipasalar, *Peer-to-peer tracking for distributed smart cameras*, Ph.D. dissertation, Princeton University, 2007.

[13] T. Horprasert, D. Harwood, and L. S. Davis, *A Statistical Approach for Real-Time Robust Background Subtraction and Shadow Detection*, vol. 99, Citeseer, 1999.

[14] F. Mühlbauer, L. O. M. Rech, and C. Bobda, "Hardware Accelerated OpenCV on System on Chip," Reconfigurable Communication-Centric Systems-on-Chip Workshop, 2008.

[15] T. M. Pinkston and J. Duato, *Interconnection Networks—Appendix E of Computer Architecture: A Quantitative Appraoch*, Morgan Kaufmann, 4th edition, 2006.

[16] "The ace orb," http://www.cs.wustl.edu/~schmidt/TAO-status.html.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 219717, 10 pages
doi:10.1155/2012/219717

Research Article

Exploration of Uninitialized Configuration Memory Space for Intrinsic Identification of Xilinx Virtex-5 FPGA Devices

Oliver Sander, Benjamin Glas, Lars Braun, Klaus D. Müller-Glaser, and Jürgen Becker

Institute for Information Processing Technology (ITIV), Karlsruhe Institute of Technology (KIT), Engesserstr. 5, 76131 Karlsruhe, Germany

Correspondence should be addressed to Oliver Sander, sander@kit.edu

Received 8 June 2011; Revised 25 October 2011; Accepted 25 October 2011

Academic Editor: Claudia Feregrino

SRAM-based fingerprinting uses deviations in power-up behaviour caused by the CMOS fabrication process to identify distinct devices. This method is a promising technique for unique identification of physical devices. In the case of SRAM-based hardware reconfigurable devices such as FPGAs, the integrated SRAM cells are often initialized automatically at power-up, sweeping potential identification data. We demonstrate an approach to utilize unused parts of configuration memory space for device identification. Based on a total of over 200,000 measurements on nine Xilinx Virtex-5 FPGAs, we show that the retrieved values have promising properties with respect to consistency on one device, variety between different devices, and stability considering temperature variation and aging.

1. Introduction

Identification of devices is a primitive that plays a crucial role for a number of applications, including authentication of devices and protection against cloning of devices (cocalled *product piracy*) or intellectual property. IDs which are stored in nonvolatile memory can often be easily cloned or modified. Hence approaches have been published to overcome the aforementioned drawbacks. They are usually based on unique physical properties of the single chip. For example, such properties are caused by manufacturing process variations. The two main approaches in this context are physical fingerprinting and the use of physical uncloneable functions (PUFs). Former strives to identify a given circuit directly by physical characteristics latter use physical characteristics to perform a challenge-response authentication.

A promising technique used for both approaches is to observe the state of uninitialized SRAM cells. When voltage above a certain threshold is applied to an SRAM cell its initial unstable state will change to one of two possible stable states "0" or "1". The probability for each stable state is heavily dependant on small variations originated during the CMOS fabrication process causing slight deviation in threshold voltage inside the cells. The probability varies between different cells even inside a single chip thus representing a characteristic initial memory content on power-up for each device. Depending on the probability distribution the major part of the memory content is stable for most of the power-ups. Other bits having a probability around 50% show a power-up behaviour similar to random noise. Assuming a high rate of stable data the memory content can be used to provide high quality identification data that is very hard to reproduce deliberately.

Considering SRAM-based field-programmable gate arrays (FPGAs), configuration memory or BRAM cells can potentially be used to retrieve fingerprints from uninitialized SRAM cells. Nevertheless this technique depends on the availability of SRAM that is not automatically initialized to a designated fixed value at device power-up. As a random configuration might well lead to short circuits and therefore damage the device physically, many vendors enforce the clearing of configuration memory by an unavoidable initialization phase on power-up. This is the case, for example, for the Xilinx Virtex-5 series FPGAs considered in this contribution. Certainly not all parts of the configuration are so critical. Initializing SRAM cells on power-up to a fixed value might consume additional area on the chip. Due to area efficiency, there is some chance for certain regions

in configuration memory without this kind of reset-on-powerup procedure. The challenge is to find possibilities for secure hardware identification without having to implement and configure complex additional logic on the device.

In this work we present a method to retrieve identification data from the configuration memory space using readouts from presumably unused and therefore uninitialized hidden address ranges. Besides description of the method and the used tools we give statistical data from our measurements on a population of Virtex-5 devices indicating the potential to use the memory region for identification purposes. Moreover we demonstrate a straightforward methodology to generate reference keys that allow for robust identification of devices.

The remainder of this paper is structured as follows. Section 2 gives an overview of some related publications. The situation and identification approach and the data measurement basics are given in Section 3. The methods for data examination are presented in Section 4. Section 5 presents the analysis results from the measurements that are interpreted in Section 6. Also open points and potential applications are discussed in Section 6. In Section 7 we give some examples of possible security applications. Specific properties and benefits of the used memory region are considered in Section 8. The paper is concluded in Section 9.

2. Related Work

As previously mentioned there exist two main approaches for physical identification of CMOS devices which have received considerable attention in recent years. An introduction to PUFs can be found, for example, in [1–4], and a collection of related publications is given in [5]. For FPGAs also the use of flip flops is presented in [6, 7]. PUFs implementation as proposed in [6] is possible but it requires a specific configuration of the device.

In the following we will focus on physical fingerprinting based on SRAM cells. The use of fabrication process-related variations of CMOS gates for identification has been widely examined. Excellent identification properties show dedicated circuits as in [8]. The use of SRAM cells (see, e.g., [9], patent [10]) comes at the cost of more noisy data but with the benefit of not requiring extra space on the chip. To extract IDs and cryptographic keys from this noisy data, various mechanisms like fuzzy extraction [11] have been proposed. Finally a preliminary version of this work was previously published in [12].

3. FPGA Configuration Memory and Proposed Approach

On many current FPGA devices the used configuration memory is initialized to a defined value at startup, rendering the SRAM cells useless for device identification by physical fingerprinting. We look at the Xilinx Virtex-5 series [13] as representatives of this kind of devices. The challenge is to find areas in the configuration memory which are not initialized due to not being critical for the chip integrity.

Table 1: Frame address register description (Xilinx Virtex-5).

Address type	Bit index	description
Block type	[23:21]	Used in Virtex-5: 000 up to 011.
Top_B bit	20	Selects between top and bottom
Row address	[19:15]	Selects the current row
Column address	[14:7]	Selects a major column
Minor address	[6:0]	Selects a frame

In first experiments we found, that the SRAM configuration cells are reliably set to zero on device power up and can therefore not be used for identification of a single device. Even the BRAM blocks where random content poses no direct threat to the physical device are reliably zeroed out.

As a solution approach we looked at parts of the configuration memory address space that are not used for configuration and are therefore possibly not included in the initialization process. It turned out that readout of address ranges reserved for additional future block types other than configurable logic, BRAMs, special frames and non-configuration frames yielded device-specific data that can serve for identification. We therefore looked at the address regions officially not used, that is, the addresses starting with a "1" in bit 23 (see Table 1).

3.1. Structure of Configuration Memory. Configuration memory in Xilinx Virtex-5 devices is organized in frames as smallest configurable units. All frames have an identical, fixed length of 1312 bits, split into 41 words of 32 bits [14]. Each frame can be addressed individually using a 24-bit address written to the Frame Address Register (FAR) of the device. The FAR is divided into five subfields as described in Table 1. Over several interfaces like JTAG and SelectMAP, frames can be written for configuration and the content can also be read out for verification.

As can be seen, the three most significant bits [23:21] are designated for identifying the block type. The block types used in the Virtex-5 series [14] are Interconnect and block configuration (000), Block RAM content (001), Interconnect and Block Special Frames (010), and Block RAM Non-Configuration Frames (011). Not (officially) used is the complete address range starting with MSB 1 ($1xx$).

3.2. Tooling. For readouting the configuration memory we used the 1149.1 JTAG [15] configuration interface and the readout procedure given by Xilinx [14]. A PC was connected via Digilent USB programming cable to the FPGA board. Based on the Digilent Port Communications Utility (DPCUTIL) [16] Library and API we created a tool for directly reading and writing configuration registers and data frames based on C#. Besides simple JTAG access the tool allows multiple readbacks and generation of some additional data thus providing the basis of our statistical analysis. For writing complete bitstreams to program the devices we used

the standard Xilinx Suite v10.1, the associated programming cables, and the iMPACT tool.

4. Examination

In this section we give some figures about the collected data. Examination results based on this data are given in Section 5.

We looked at a total of nine Xilinx XC5VLX110T devices [17], integrated in Digilent XUPV5-LX110T Evaluation Platforms [18]. In the following we denote the devices with letters A, ..., I. Since the substructure of the address space outside the area used for configuration is not public, we used the autoincrementation of the FAR to determine valid addresses. It turned out that at least 96 kbit of nontrivial data could be read out from an unconfigured device within a certain memory region. A block of ten consecutive frames (13,120 bit) was chosen for closer investigation. In this paper, we refer to this data stream consisting of 10 frames of data. Each time we perform a readout of the device, we read these 10 frames.

For statistical examination for each device $X \in (\mathrm{A}, \ldots, \mathrm{H})$ two measurement series of 10,000 readouts $\Theta_X = (\vartheta_X^1, \ldots, \vartheta_X^{10,000})$ each were collected, one series from the unconfigured device (Θ_X^i) and one from the programmed device $(\overline{\Theta}_X^i)$. So a total of 160,000 data streams (each consists of 10 frames) could be used for statistics and creating master identification keys. In addition, test data $T_X = (\tau_X^1, \ldots, \tau_X^{100})$, resp., $\overline{T}_X = (\overline{\tau}_X^1, \ldots, \overline{\tau}_X^{100})$ of 100 readouts was collected from every board setup, that is, a total of 1,600 test streams. Eventually for the exploration of temperature stability we made another 40000 measurements for three selected boards D, H, and I which were exposed to a wide temperature range.

For ideal identification of devices it would be optimal to have a bijective mapping from an ID to a physical device and vice versa. The examination of the measurement data was therefore done looking in two directions. First the correlation and consistency of the different measurements on one board were investigated, to get a unique mapping from a device to an ID. From each measurement series a reference data stream as master identification key was created that serves as a candidate for device representation. In a second step the results from different boards were compared aiming for an injective mapping from one ID to one specific device. Validation of the identification process was performed using the test data sets and the reference IDs. The results are given in the following paragraphs.

5. Results

5.1. Similarity and Reference Keys. First we look at the conformity of different readouts from the very same device X to map each device to an ID. Therefore we used 10,000 readouts to create a frequency distribution of zeros and ones of every bit in the data stream for each FPGA. Figures 1 and 2 show the results for two single devices. The other devices showed similar figures.

The measurement reveals a distinct accumulation of bits showing constantly the same value. As can be seen in Table 2

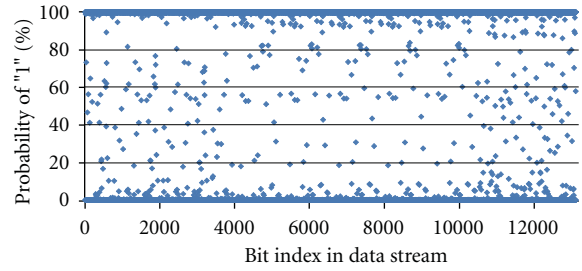

Figure 1: Probability of bit value "1" over 10 k measurements on device D.

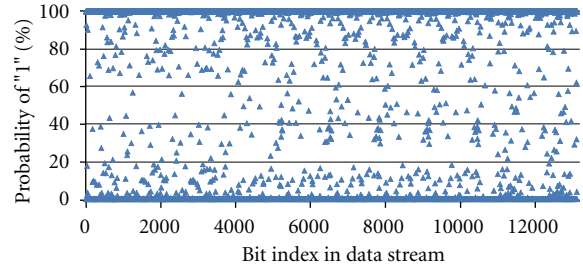

Figure 2: Probability of bit value "1" over 10 k measurements on device F.

the number of ones dominates the number of zeros by a factor of 1.7. The portion of flipping bits is below ten percent ranging from 7.2% to 9.7% with a mean value of 8.4% over the devices under consideration (see Figure 3). The distribution of zeros and ones is pretty similar for different devices (Figure 3). For the very same device—configured or unconfigured—the numbers are not identical but relatively close to each other. These differing results for one device are more probably caused by temperature variances than by the fact of configuration dependencies.

Figure 4 depicts the total amount of constant bits observed over a variable number of compared readouts for different devices. The measurements show that more than 90% of all bits are constant over all compared readouts. We therefore notice a distinctive coherence of measurements from one device.

To quantify this coherence we compared the single data streams with a reference stream ρ to get a measurable deviation value. To achieve this we used the probability distribution to generate a reference data stream ρ of the measurement, setting each bit to the value with the higher probablity of occurrence according to the measured data. ρ therefore represents the bitwise rounded mean of all measured streams.

We then determined the Hamming Distance (HD) of every readout to this reference key. Figure 5 shows the respective distribution for one device. The readouts from one board show a close cross-correlation. For device D the Hamming distances show an expectancy value of 99, equivalent to only 0.75% of the total data, and a standard deviation of 7.4. Device F gives an expectancy value of 137 (1.0%) with a standard deviation of 9.1. All HD values

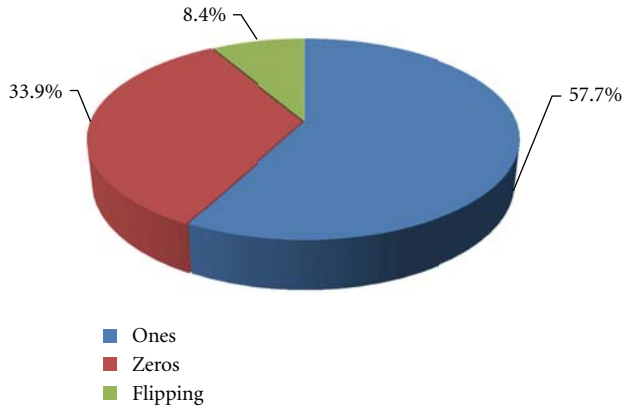

Figure 3: Mean occurrences of constant ones and zeros as well as flipping bits of Table 2.

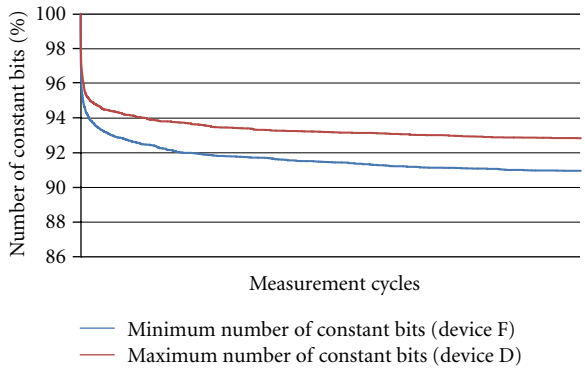

Figure 4: Percentage of constant bits over 10,000 measurements for two devices, that represent the maximum (device D) and minimum (device F) over all devices. All other values reside in the area between the two curves.

computed from all devices resided in an interval of [53–208]. The analyses of the measurement series from the other devices are in between the two extremal values given.

So far, the compared values were originated from unconfigured devices and directly measured after startup. For identification of the device, the key data should be independent from the content of the address ranges of configuration memory that are used for programming the device. Therefore, also the programmed device was examined. Figure 6 depicts the HD values of readouts of a programmed board in relation to the reference key determined on the same board in unconfigured state.

The values read out from the configured board show a slightly higher deviation from the reference but are still within a mean deviation of well below 1% of the total data stream. This is also the case for the other examined devices (see also Table 3). So all data streams from one board show a great mutual similarity. To verify the usability of the proposed approach, we compare in a next step data from different devices.

5.2. Distinction and Identification. For unique identification, the mapping of a given ID value to a device is necessary.

Table 2: Number of constant zeros, ones, and flipping bits.

Device	Ones		Zeros		Flipping	
A	7798	59,4%	4238	32,3%	1084	8,3%
$\overline{A}$	7744	59,0%	4190	31,9%	1186	9,0%
B	7698	58,7%	4435	33,8%	987	7,5%
$\overline{B}$	7661	58,4%	4414	33,6%	1045	8,0%
C	7388	56,3%	4676	35,6%	1056	8,0%
$\overline{C}$	7325	55,8%	4642	35,4%	1153	8,8%
D	7702	58,7%	4479	34,1%	939	7,2%
$\overline{D}$	7638	58,2%	4432	33,8%	1050	8,0%
E	7626	58,1%	4350	33,2%	1144	8,7%
$\overline{E}$	7566	57,7%	4357	33,2%	1197	9,1%
F	7517	57,3%	4418	33,7%	1185	9,0%
$\overline{F}$	7491	57,1%	4352	33,2%	1277	9,7%
G	7498	57,1%	4577	34,9%	1045	8,0%
$\overline{G}$	7418	56,5%	4543	34,6%	1159	8,8%
H	7537	57,4%	4543	34,6%	1040	7,9%
$\overline{H}$	7460	56,9%	4552	34,7%	1108	8,4%

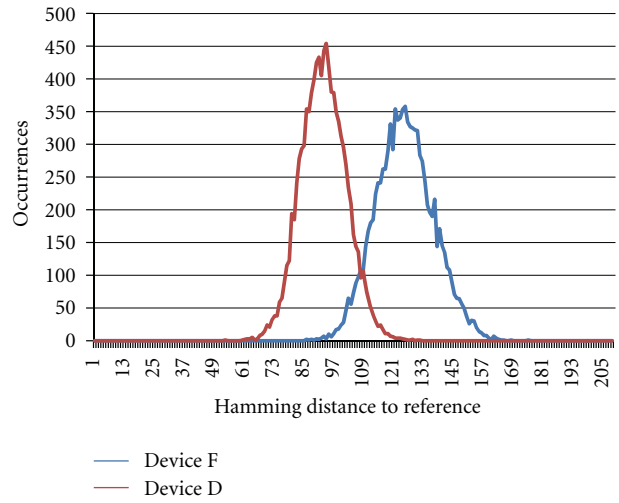

Figure 5: Distribution of Hamming distances to the respective reference streams for unconfigured devices D and F and 10,000 measurements each.

Here the difference between readouts of different boards is the crucial property. A first indicator is the difference between the computed reference keys. Table 3 shows the mutual Hamming distances of all reference keys.

The table shows a very close correlation between the measurements of the same device, meaning that only a slight dependency on the configuration state can be determined. In contrast, HD values between different devices are all near 6000, meaning a 45% deviation and therefore near to the expected value of 50% for pure random streams. A closer look at the streams revealed some bits that are constantly zero over all measurements of all devices, which in combination with the slight nonuniform distribution of ones and zeroes (see Table 2) may account for the lower absolute deviation.

Table 3: Mutual Hamming distances of reference streams from unconfigured devices A . . . G and configured devices $\overline{A}$ . . . $\overline{G}$.

Device	A	$\overline{A}$	B	$\overline{B}$	C	$\overline{C}$	D	$\overline{D}$	E	$\overline{E}$	F	$\overline{F}$	G	$\overline{G}$	H	$\overline{H}$
A	0	90	5903	5918	6161	6151	6020	6024	6089	6118	6201	6201	5981	5973	6073	6100
$\overline{A}$	90	0	5897	5912	6157	6147	6008	6012	6101	6130	6199	6199	5975	5967	6047	6072
B	5903	5897	0	87	6268	6276	5967	5963	5990	6011	6050	6048	6118	6128	5952	5969
$\overline{B}$	5918	5912	87	0	6283	6291	5972	5968	5965	5980	6077	6075	6103	6113	5949	5964
C	6161	6157	6268	6283	0	74	6339	6343	6054	6053	6170	6176	6266	6250	6012	6027
$\overline{C}$	6151	6147	6276	6291	74	0	6315	6319	6032	6031	6146	6152	6262	6246	6014	6029
D	6020	6008	5967	5972	6339	6315	0	78	5991	6010	6165	6159	5989	5991	5979	6000
$\overline{D}$	6024	6012	5963	5968	6343	6319	78	0	5997	6016	6135	6129	5977	5979	5993	6014
E	6089	6101	5990	5965	6054	6032	5991	5997	0	113	6088	6092	6032	6024	6042	6079
$\overline{E}$	6118	6130	6011	5980	6053	6031	6010	6016	113	0	6101	6105	6041	6033	6039	6080
F	6201	6199	6050	6077	6170	6146	6165	6135	6088	6101	0	30	6194	6198	6108	6135
$\overline{F}$	6201	6199	6048	6075	6176	6152	6159	6129	6092	6105	30	0	6188	6192	6098	6125
G	5981	5975	6118	6103	6266	6262	5989	5977	6032	6041	6194	6188	0	58	6080	6077
$\overline{G}$	5973	5967	6128	6113	6250	6246	5991	5979	6024	6033	6198	6192	58	0	6086	6085
H	6073	6047	5952	5949	6012	6014	5979	5993	6042	6039	6108	6098	6080	6086	0	119
$\overline{H}$	6100	6072	5969	5964	6027	6029	6000	6014	6079	6080	6135	6125	6077	6085	119	0

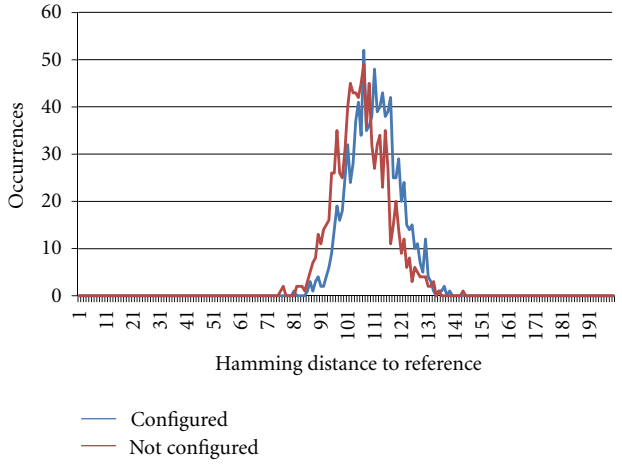

Figure 6: Distribution of Hamming distances to the device D reference stream ρ_D (generated from the unconfigured device data Θ_D) for 1000 test data streams from the unconfigured ($T_X = (\tau_X^1, \ldots, \tau_X^{1000})$) and configured ($\overline{T}_X = (\overline{\tau}_X^1, \ldots, \overline{\tau}_X^{1000})$) device each.

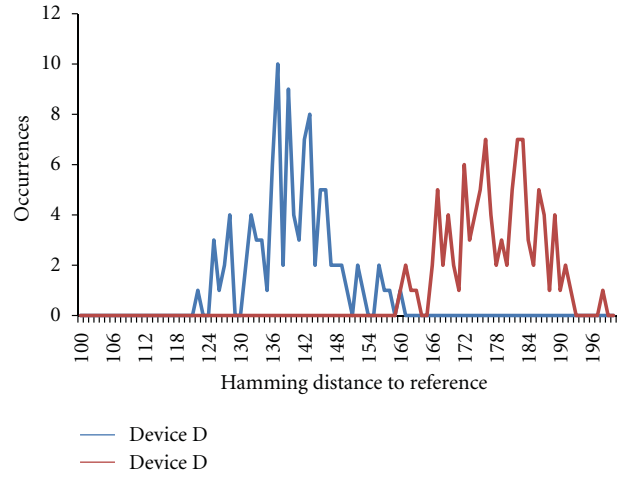

Figure 7: Enlarged presentation of the Hamming distance distribution of T_D relative to ρ_D and $\overline{\rho}_D$.

Figure 8 shows the comparison of the test value set from device D with all reference strings. Two distinctive peak clusters are visible in the chart. Two peaks with relatively small average Hamming weights of about 160 originate from the comparison with the two reference sets ρ_D (unconfigured) and $\overline{\rho}_D$ (configured) from the same board D (see Figure 7 for an enlarged view). The remaining 14 peaks belong to reference values of the other devices and are clustered in a narrow interval around an HD of 6000. Figures 7 and 9 detail the two interesting intervals rescaled for better visibility.

The charts show a clear separation of matching and non-matching devices with differences in the Hamming distance of more than an order of magnitude. The results for the other devices are very similar and also provide recognition rates of 100% using very simple threshold algorithms.

5.3. Temperature Stability. So far all measurements have been made at standard laboratory conditions including a regular temperature of app. 25°C. In diverse applications the temperature of the FPGA may vary in some range around this temperature. In order to ensure applicability of our approach some independability of temperatures must be proven. For this we selected three different boards D, F, and I and exposed these to different temperatures ranging from −30°C to +80°C in a climate chamber.

The first experiment conducted was to determine the variation of the fingerprint at different temperatures. This was done by reading back the device 1000 times and building a master bitstream out of this measurement. Then each

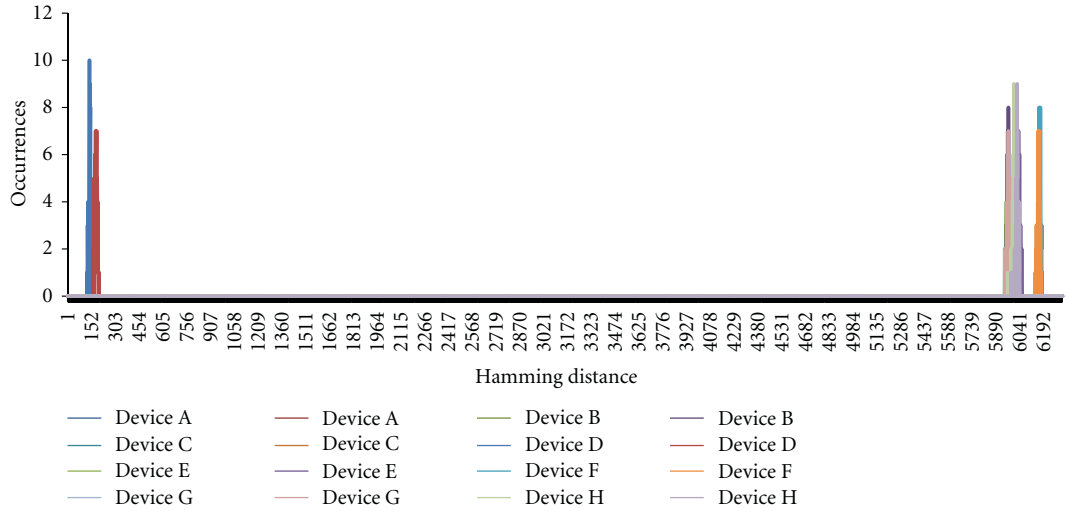

Figure 8: Distribution of Hamming distances of test set $T_D = (\tau_D^1, \ldots, \tau_D^{100})$ to all reference values ρ_X and $\overline{\rho}_X$ for $X \in \{A, \ldots, G\}$.

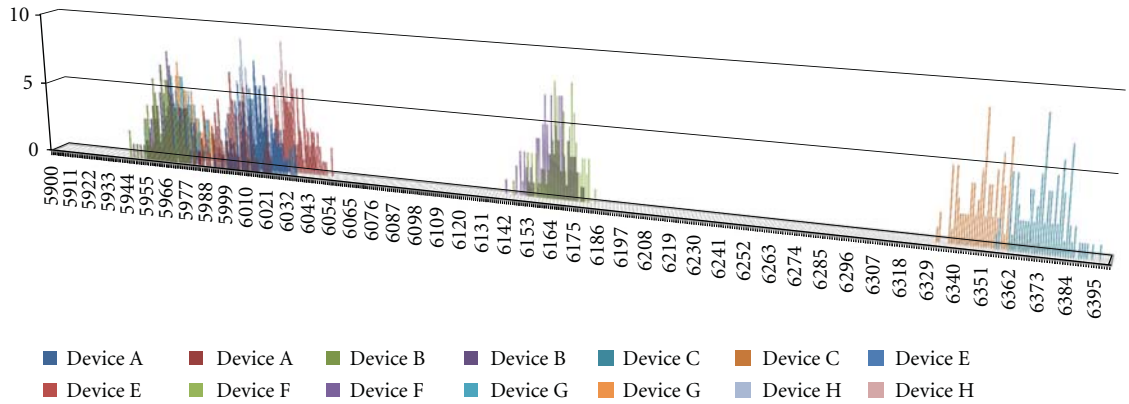

Figure 9: Enlarged presentation of the Hamming distance distribution of T_D relative to ρ_X and $\overline{\rho}_X$ for $X \neq D$.

bitstream is compared to the master. Representative results are depicted in Figures 10, 11, and 12 for the mean value of 20°C and both extrema −30°C and +80°C. The mean hamming distance varies for about 40 bits depending on the temperature, what is quite a moderate value, having in mind a nonmatching HD of about 6000. Moreover for different temperatures the curves look quite similar. Interestingly there is no common temperature-dependent behavior. While in Figure 11 the standard temperature has the smallest variation and the lowest temperature the highest one, the situation changes for the second board, where the lowest temperature has the lowest variation and the highest temperature the highest one. Actually this is the behavior we expected before our experiments. However the situation reverses for board D in Figure 12. So we can only conclude that there is some slight variation with different temperatures.

In our second experiment we compared the measurements taken at different temperatures to the reference bitstream generated from the readbacks at 20°C. The results for Board H are depicted in Figure 13. Results for other boards look similar. Obviously the comparison shows optimum results for 20°C. For all other measurements we get a rising hamming distance with greater temperature differences for both measurements. Again the form of the curves remains similar which again means the variance of the hamming distances are almost equal. The maximum hamming distance goes up to around 300 for the highest and lowest temperatures which is very moderate compared to non matching hamming distances.

The third experiment is a temperature sweep, starting at 85°C and cooling down to −30°C. This process took about 40 min, and 4800 measurements were taken with 500 ms between each measurement. Out of all measurements a master bitstream was built and the hamming distance for each readback calculated. The results are shown in Figure 14. At the beginning the hamming distance is around 350, goes down to 120 at approximately 0°C, and then goes up again with even more decreasing temperature. The blue line shows the chip temperature which is around 5 to 10° higher than the surrounding temperature for an unconfigured Virtex-5 device.

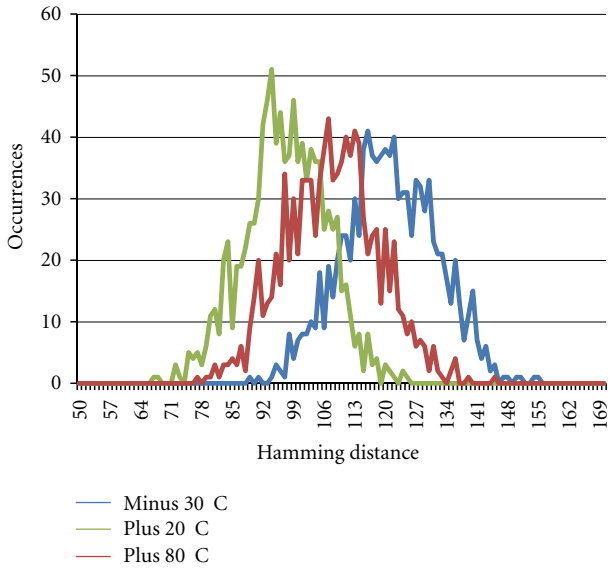

Figure 10: Hamming distance of single readback and master bitstream of board H at different temperatures.

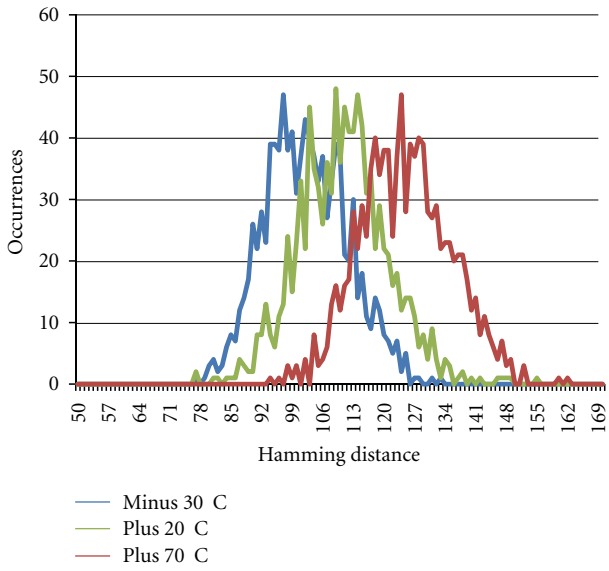

Figure 11: Hamming distance of single readback and masterbitstream of board I at different temperatures.

One can conclude that our device identification mechanism shows slight temperature dependency, which does not influence the identification process. In a final experiment we used the 20°C master bitstreams for non matching devices at different temperatures. For such non matching devices the temperature has almost no influence, resulting in a deviation of about 20 in the hamming distance. Related to an absolute hamming distance of around 6000 this is neglectable. So finally device recognition is feasible for a wide temperature range.

5.4. Aging. Another important aspect is device aging. Again, we have to point out we do not know what information exactly is read from the device by our methodology. However

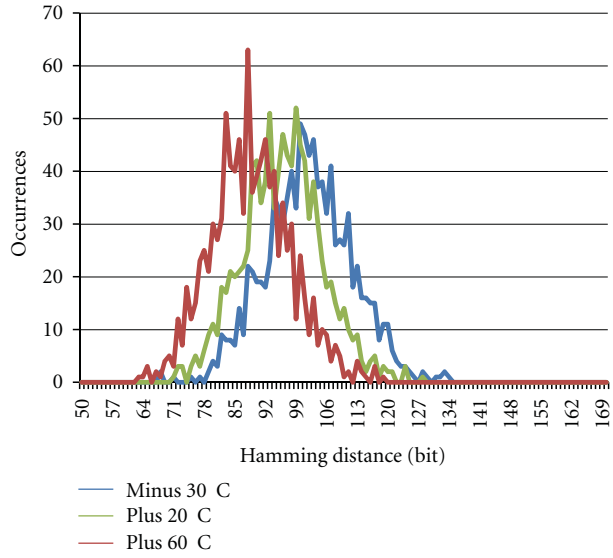

Figure 12: Hamming distance of single readback and masterbitstream of board D at different temperatures.

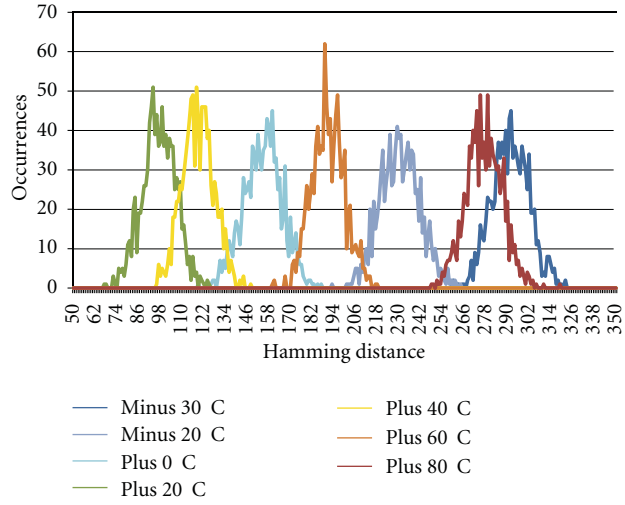

Figure 13: Matching device identification with master bitstream generated at 20°C and identification readback temperature range from −30°C to +80°C (1 k measurements at each temperature step).

we did a readback for different devices with a time difference of one year. The master bitstream generated one year ago was applied to the readback. Figure 15 shows the result for device H. One can see a slight deviation between both devices. The difference might be based on some aging of the device, but also slight deviations in the environment cannot be completely ruled out. However this effect is relatively small and does not affect the identification mechanism presented in this paper.

6. Discussion

The examination shows a very high correlation within data streams from a single device. Test data revealed deviations from the reference master around 1% of the total data

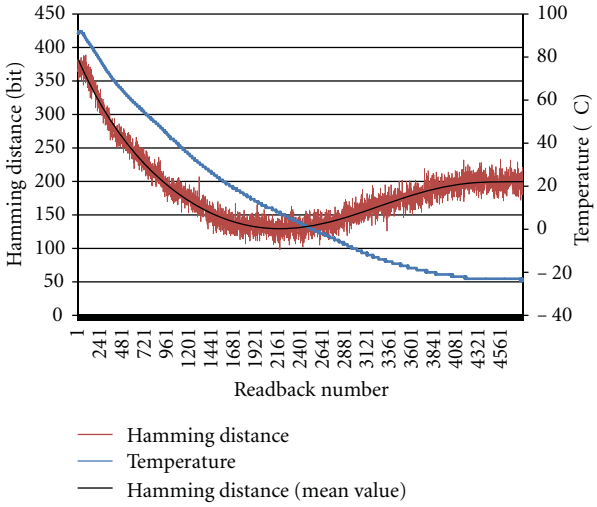

Figure 14: 4800 hamming distance measurements taken during temperature sweep from 85°C to −30°C environmental temperature. The blue curve shows the chip temperature.

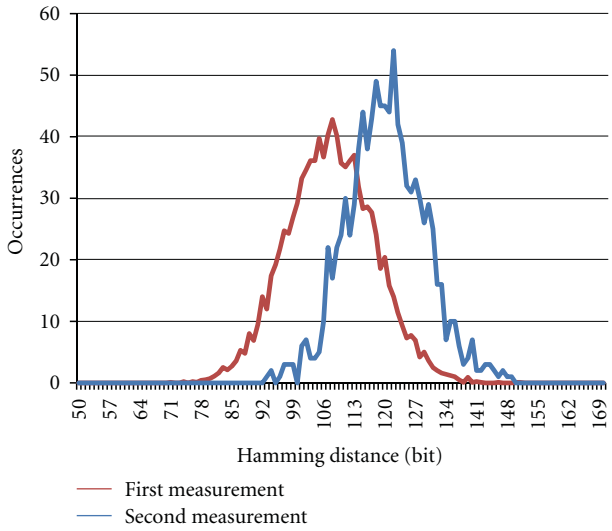

Figure 15: Two device identification measurements taken with one year between 1st and 2nd measurement.

stream. On the other hand comparison to reference keys from other devices resulted in HD values corresponding to a deviation in more than 40% of the bits enabling a reliable identification of the device being read out. In addition the identification technique needs no configuration on the device and therefore no area on the configurable fabric or other hardware resources. We therefore believe that the method can be used to securely and reliably identify physical devices.

An open issue is the question where the data originates from in the first place. Since the address space is not used officially on the devices it could as well be omitted. So it is possible that the readback retrieves data from memory locations that are in fact used for some purpose. On the other side there are the distinct differences between different instances of the same device model and the Hamming

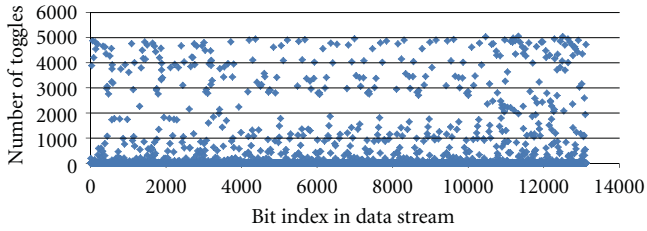

Figure 16: Number of single bit value changes for the different bit indices over 10,000 measurements on device D.

distances near the expectancy values for random strings. This seems to be unlikely for deliberately configured data. Nevertheless we are so far not able to determine the precise origin of the readout data. In addition the data shows slight repetitive patterns as can be seen in Figures 1 and 2 that have to be analysed to get valid statements about the number of usable identification bits.

Moreover the question arises whether the data could also be used to generate random numbers. Here not the constant but the variable bits are crucial for the quality. Figures 1 and 2 already show that the data have different distribution depending on the bit index. Figure 16 shows in addition the number of value changes over the measured set of data streams for the different bit indices.

The variety of behaviours of the different bits could be seen as an indication that an extraction of random numbers could be possible, but further analysis and postprocessing of the data is needed to get reliable results.

7. Security Applications

There are several possible applications for the described SRAM-based PUFs. Since the PUF consists of memory values, it is easy to evaluate also for an attacker. Hence it should be possible to create the memory response without the specific device but only a readout of the memory. A security application therefore has to make sure that the PUF is actually evaluated and not replayed values are used. One possibility for that is guaranteed direct physical access to the device. This is utilized in two applications examplarily looked at in the following: product piracy protection and IP protection.

For the first use case, the initial memory content is read out at production of the device and an adequate error correcting coding mechanism is applied to create a stable fingerprint. This fingerprint is then used to create a manufacturer certificate for the device, for example, by signing a hash value of the fingerprint with a manufacturer-specific secret key. This certificate is then delivered together with the device to the customer who is then able to verify whether he acquired a genuine device since a manufacturer of bogus devices is neither able to create valid certificates for his device nor to clone the PUF-fingerprints for the cloned units. Since the fingerprinting does not rely on memory regions used for configuration the verification can be performed even during operation without interruption and without need to restart the device.

Possible applications for the considered identification mechanism are not restricted to external evaluation of a device but could as well be used inside a configuration bitstream itself. This is considered in the second use case for IP protection. Since the configuration memory is accessible from the fabric through the common ICAP interface, a configured bitstream could easily use the available information to identify the device it is configured on and perhaps react accordingly. A straightforward application would be to bind a configuration to a specific device. To show feasibility we implemented the basic functionality.

Using an ICAP interface connected to a MicroBlaze System we were able to read out the identifying data from within the device, getting similar results as when using the external JTAG access. By comparing the data with a reference string stored in the configuration bitstream, the design can verify whether it is run on a predefined device. If a mismatch is detected, an internal enforcement mechanism can, for example, disable the design or reduce functionality, building a reliable copy protection mechanism for hardware configuration IP.

8. Specific Properties of the Memory Region

When the identification is used for policy enforcement and copy protection, circumventing the identification could be of interest for possible attackers. In the classical use case using memory cells for identification it has to be guaranteed that the cells are read out before any influence on the contents is possible, since the memory cells can easily be written with arbitrary values. In contrast to that we observed that writing on the considered addresses was not possible over the examined interfaces—the readout values were not changeable by write attempts. This is a major benefit in contrast to general memory cells.

In addition, since the used cells are outside the memory space used for configuration, we assume that they are not used for programming a design to the chip. The identification should therefore work also for configured devices during runtime and for partial bitstreams having no control and no information about the contents of the surrounding chip area or even about their own placing on the area as long as they have access to the ICAP interface. So no area has to be reserved for the identification and no constraints for placement and routing are imposed by the approach.

9. Conclusion and Further Work

For the challenge of identifying physical devices of Xilinx Virtex-5 FPGA family we examined configuration memory readouts from address ranges that are assumedly not used in the series. We chose an address range reserved for future block types to read out data and compared the readouts mutually from one device and between devices. Results showed a strong coherence of different streams from the same device and strong deviations between devices. This holds true also for a wide temperature range. Moreover we showed device identification is feasible after one year of aging. We therefore assume the data suitable for identification of physical devices. The method opens identification possibilities for FPGA series where other SRAM-based approaches fail because of enforced initialization to defined values at startup.

To proof reliable identification some future work is necessary. So far it is not clear how long the potential key sequences are and in what ratio they include real identification information and noise. It could also be investigated, whether it is also applicable to devices of other series.

Acknowledgments

The authors acknowledge support by Deutsche Forschungsgemeinschaft and Open Access Publishing Fund of Karlsruhe Institute of Technology.

References

[1] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*, pp. 148–160, ACM, New York, NY, USA, November 2002.

[2] B. Gassend, D. Lim, D. Clarke, M. van Dijk, and S. Devadas, "Identification and authentication of integrated circuits," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 11, pp. 1077–1098, 2004.

[3] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Fpga intrinsic pufs and their use for ip protection," in *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '07)*, pp. 63–80, Springer, Berlin, Germany, 2007.

[4] P. Tuyls and B. Skoric, "Strong authentication with physical unclonable functions," in *Security, Privacy, and Trust in Modern Data Management*, M. Petkovic and W. Jonker, Eds., Data-Centric Systems and Applications, pp. 133–148, Springer, Berlin, Germany, 2007.

[5] R. Maes, "PUF Bibliography," 2010, http://www.rmaes.ulyssis.be/pufbib.php/.

[6] R. Maes, P. Tuyls, and I. Verbauwhede, "Intrinsic pufs from flip-flops on reconfigurable devices," in *Proceedings of the 3rd Benelux Workshop on Information and System Security (WISSec '08)*, p. 17, Eindhoven, NL, USA, 2008.

[7] S. S. Kumar, J. Guajardo, R. Maes, G. J. Schrijen, and P. Tuyls, "The butterfly PUF protecting IP on every FPGA," in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST '08)*, pp. 67–70, June 2008.

[8] Y. Su, J. Holleman, and B. Otis, "A1.6pJ/blt 96% stable chip-ID generating circuit using process variations," in *Proceedings of the 54th IEEE International Solid-State Circuits Conference (ISSCC '07)*, pp. 406–611, San Francisco, Calif, USA, February 2007.

[9] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-up SRAM state as an identifying fingerprint and source of true random numbers," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1198–1210, 2009.

[10] S. Chaudhry, P. A. Layman, J. G. Norman, and J. R. Thomson, "Electronic fingerprinting of semiconductor integrated circuits," US patent 6,738,294, Agere Systems Inc., 2002.

[11] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy extractors: how to generate strong keys from biometrics and

other noisy data," *SIAM Journal on Computing*, vol. 38, no. 1, pp. 97–139, 2008.

[12] O. Sander, B. Glas, L. Braun, K. Müller-Glaser, and J. Becker, "Intrinsic identification of xilinx virtex-5 fpga devices using uninitialized parts of configuration memory space," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 13–18, December 2010.

[13] Xilinx Inc., *UG190: Virtex-5 FPGA User Guide*, 2009, v5.2, November 2009.

[14] Xilinx Inc., *UG191: Virtex-5 FPGA Configuration User Guide*, 2009, v3.8, 14.08.2009.

[15] IEEE, "1149.1: IEEE Standard Test Access Port and Boundary-Scan Architecture," IEEE-SA Standards Board, IEEE Standard 1149.1-2001 (R2008), 2006.

[16] Digilent Inc., *DPCUTIL Programmer's Reference Manual*, 2007, doc 576-000, August 2007, http://www.digilentinc.com/Data/Software/Adept/DPCUTIL_Programmers_RM.pdf.

[17] Xilinx Inc., *DS100: Virtex-5 Family Overview. Product Specification, 2009, v5.0*, February 2009.

[18] Xilinx Inc., *UG347: ML505/ML506/ML507 Evaluation Platform: User Guide, 2009, v3.1.1*, October 2009.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 608312, 12 pages
doi:10.1155/2012/608312

Research Article

Placing Multimode Streaming Applications on Dynamically Partially Reconfigurable Architectures

S. Wildermann, J. Angermeier, E. Sibirko, and J. Teich

Hardware/Software Co-Design, University of Erlangen-Nuremberg, 91058 Erlangen, Germany

Correspondence should be addressed to S. Wildermann, stefan.wildermann@cs.fau.de

Received 29 April 2011; Revised 21 October 2011; Accepted 25 October 2011

Academic Editor: Claudia Feregrino

By means of partial reconfiguration, parts of the hardware can be dynamically exchanged at runtime. This allows that streaming application running in different modes of the systems can share resources. In this paper, we discuss the architectural issues to design such reconfigurable systems. For being able to reduce reconfiguration time, this paper furthermore proposes a novel algorithm to aggregate several streaming applications into a single representation, called merge graph. The paper also proposes an algorithm to place streaming application at runtime which not only considers the placement and communication constraints, but also allows to place merge tasks. In a case study, we implement the proposed algorithm as runtime support on an FPGA-based system on chip. Furthermore, experiments show that reconfiguration time can be considerably reduced by applying our approach.

1. Introduction

Embedded systems used to have fixed functionality dedicated to a specific task. A trend of recent years is, however, to build embedded systems which are regarded as "smart." In this context, the functionality can be adapted, reorganized, and controlled by external requests from human operators or even internal self-managing, self-adapting mechanisms. This means that an embedded system's functionality is not fixed to a single configuration anymore, but several application are running. Still, not all applications are running all the time, but depend on the *operational mode* of the system. Here, we are speaking of *multimode systems*, which can, for example, be found in a typical "smart phone" scenario where the operational mode depends, for example, on the radio link quality or the user's input. Other examples are "smart cameras" which have internal adaptation mechanism that switch between appropriate signal processing algorithms. In this paper, we consider multimode streaming applications which can also be found in above-mentioned examples, in the form of signal and image processing flows, but also Data Stream Management Systems, and so forth.

In a multimode system, the system resources can be utilized better. This is due to the fact that, during system design, only those applications which run concurrently are considered. This makes it possible to perform resource sharing between processes which run mutually exclusive. When speaking of resource sharing of hardware resources, one precondition is the reconfigurability of the computing platform. In this context, field-programmable gate arrays (FPGAs) offer the capability of partial reconfiguration. This means that hardware modules can be dynamically loaded and partly exchanged during runtime. Applying this technology in the design of multimode streaming applications, however, requires (a) system architectures which support partial reconfiguration and (b) proper reconfiguration managers which provide algorithms for scheduling and placing applications at runtime.

This paper deals with modeling and dynamically placing streaming applications on FPGA-based platforms. The focus lies on systems where it is not predictable at design time which applications actually run and in which order they are executed. This is especially the case when implementing autonomic SoCs. For example, [1] introduces a methodology for self-organizing smart cameras which switch between configurations depending on runtime decisions. In this self-managing context, placement needs to be flexible since it is not known beforehand when and what kind of reconfiguration is requested.

One advantage of running streaming applications in hardware is to exploit parallelism by means of pipelining. Therefore, the complete data flow is placed and modules are connected via circuit switching techniques. Particularly in the domain of streaming applications, data flows may have strong similarities. For example, in image processing, convolution filters are commonly applied. Now, while one streaming application uses a Canny filter for edge detection, another application might use a Sobel filter. In this case, it is not necessary to replace the filters by means of reconfiguration when switching between both streaming applications, since the logic remains the same. A better strategy would be to use the same filter module for both applications, and to keep the filter parameters in registers. So, replacing the Canny filter module by the Sobel filter can be achieved by replacing the filter parameters in the registers. It is possible to reduce reconfiguration time when applying this kind of resource sharing.

In this work, we provide a novel approach to merge the data flow graphs of multiple applications with strong similarities into a single data flow graph. We review techniques to build proper reconfigurable systems to run these multimode streaming application. Furthermore, an algorithm is present for placing such streaming applications. The approach considers the heterogeneity of common FPGAs, such as Block-RAM columns, as well as the routing restrictions of on-chip streaming bars.

The paper is organized as follows. Section 2 presents related work. Section 3 provides the preliminaries of partial reconfiguration. Section 4 describes the model for streaming applications as well as the algorithm to generate the merge graph. The problem of placing streaming applications is then formally defined in Section 5. Section 6 provides the placing algorithm. An implementation of the algorithm on an FPGA platform is described in Section 7 which is used in the experiments provided in Section 8. Section 9 concludes this paper.

2. Related Work

Partial run-time reconfiguration of FPGAs has been investigated since several years. But still, reconfiguration harbors many problems. There are of course architectural issues concerning how to support partial reconfiguration and how to establish systemwide communication. Solutions can be classified into three basic principles: *on-chip buses* are the most common way of linking together communicating modules. For example, [2] and [3] present on-chip buses for flexible module placement. Hard macros implement the communication infrastructure within the FPGA fabric. *Circuit switching* is a technique where physically wired links are established between two or more modules as, for example, presented in [4] and [5]. The latter has a low overhead since it implements the wires directly within the routing fabric of the FPGA. We consider this kind of technique for streaming data in reconfigurable systems. *Packet switching* techniques, for example, networks on chip [6], are also possible but impose a certain overhead when implemented on FPGAs.

Hardware reuse is basically investigated on the logic and the system level. Reference [7] provides a mechanism for hardware reuse on the logic level. Frames in common with the previously placed configuration are determined which will then be reused. A similar approach is followed in [8] which aims at increasing the number of common frames by properly ordering the LUT inputs to keep their content the same over several configurations. Such approaches are highly dependent on the architecture and are computationally intensive.

System level approaches are often performed via graph matching techniques. For example, [9] presents an approach where common components between successive configurations are determined to minimize reconfiguration time. Reference [10] presents an approach which is based on the *reconfiguration state graph* (RSG) which models the configurations and the reconfiguration between them. These approaches are applied during synthesis and require knowledge of the switching order between configurations. The result is the synthesized system. Hence, these methods are not applicable when extending a running system without having to perform a complete redesign, also affecting flexibility as well as maintainability and limiting the use of self-managing mechanisms.

Reference [11] introduces an algorithm to place streaming applications with similarities. In order to express these similarities, the authors propose to model the applications by means of extended data flow graphs. Here, so-called OR nodes are used to describe differing parts of applications while keeping their similarities. A First-Fit heuristic is then introduced to place such applications. In the work at hand, we replace the model from [11] by merge graphs. In a merge graph, several data flow graphs are merged into a single representation. However, no OR nodes are required. Each node of the merge graph is annotated with the identifier of the applications it is part of. An algorithm to automatically generate merge graphs is proposed. The placing algorithm from [11] is then extended to work with this novel model.

3. Partially Reconfigurable Architectures

Reconfigurable architectures are typically divided into static and dynamic parts. Logic residing in the static part is needed throughout system operation, for example, the connection logic for the communication interfaces.

On the other side, the dynamic part is designed for loading and exchanging functionality at runtime. Functionality which is only required in some operational system modes may be implemented as *partially reconfigurable modules* (PR modules) and, thus, can be dynamically loaded and exchanged during runtime. In this way, logic and other resources can be shared by system components which only run mutually exclusive.

In the following, we concentrate on hardware reconfiguration on FPGAs. We present possibilities to organize the reconfigurable fabrics to support partial reconfiguration, and give an overview of sophisticated communication techniques, before presenting a concrete example.

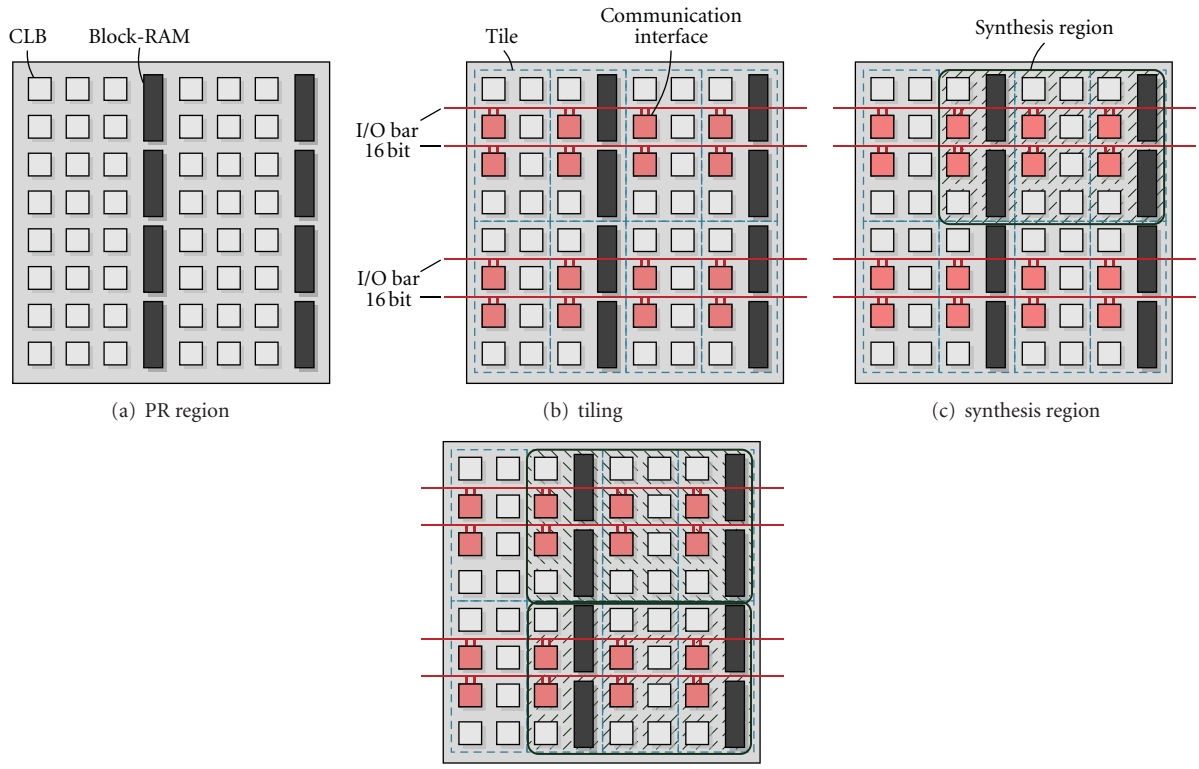

FIGURE 1: Example of a PR region and a possible tiling.

3.1. Enabling Partial Reconfiguration. PR modules can be loaded into dedicated areas on the reconfigurable systems. These areas are denoted as partially reconfigurable regions (PR regions). The possible shapes and positions for loading PR modules into these regions strongly depend on the PR region layout. Here, the problems are that, first, modern reconfigurable fabrics are an array consisting of cells of different types, such as logic blocks (CLBs), Block-RAMs, or multiply-accumulate units (DSPs), as illustrated in Figure 1(a). This increases the inhomogeneity of the underlying structure. Second, low-overhead infrastructures have to be provided that enable systemwide communication and support partial reconfiguration.

One common solution is to build so-called *tiled partially reconfigurable systems*, as proposed, for example, by [2]. Here, the PR regions are divided into *reconfigurable tiles* which are the smallest atomic units for partial reconfiguration. An example for a possible tiling is shown in Figure 1(b). The approach allows to place multiple PR modules with various sizes in the same PR region. At the same time, placement restrictions are decreased and usability is enhanced. Still, the tiles consist of several different cell types according to the underlying cell structure.

Sophisticated communication techniques for reconfigurable systems are commonly provided by *communication macros* which can be placed in the PR region to interconnect the reconfigurable tiles. Since each reconfigurable tile offers a *communication interface*, the tiling of the PR region depends on the chosen communication technique. Koch et al. give a rule in [12] how to calculate the size of tiles. It depends on the communication overhead in terms of required cells as well as the set of modules which are candidates for being placed as PR modules.

The synthesis flow for PR modules requires additional steps compared to the flow for static system designs (see, e.g., [2, 5]). First, an area on the reconfigurable system is selected which provides sufficient cell types as required for implementing the module. This area is denoted as *synthesis region*. Although this does not necessarily have to be the case, most synthesis flows propose the use of rectangular synthesis regions. Then, the communication macro is placed. Finally, the module can be synthesized in the selected region. This flow now allows to place the module at any location with the same arrangement of cell types as the synthesis region. An example of a synthesis region for a PR module requiring 10 CLBs and 4 Block-RAMs is given in Figure 1(c). The two *feasible positions* for placing this PR module are depicted in Figure 1(d).

As pointed out in this section, the chosen communication technique strongly influences the layout of the partially reconfigurable part of the system. In the following, the communication techniques targeted in this work are presented.

3.2. Communication Techniques. Section 2 describes approaches to provide communication interfaces for dynamic placement of hardware modules. *On-chip buses* are suitable for dynamically integrating hardware modules into an FPGA

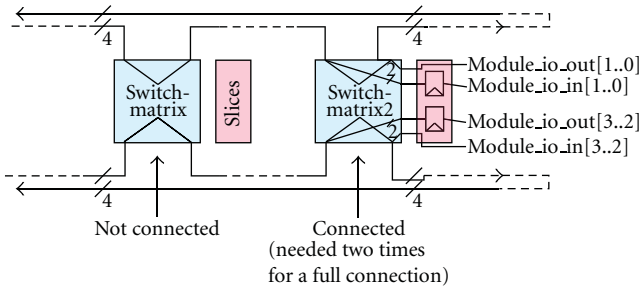

Figure 2: Streaming bar implemented through the FPGA routing fabric.

by partial reconfiguration. Sophisticated techniques [2, 3] are provided as FPGA macros which allow partial reconfiguration at runtime, even during bus transactions. Such buses permit connections between modules, including the communication to and from components of the static part of the system.

Circuit switching techniques are used for streaming data and establishing point-to-point connections between hardware modules. Sophisticated approaches, for example, *I/O bars* [5] allow to implement this behavior with a low overhead within the FGPA routing fabric. The basic principle is depicted in Figure 2. A module is able to read the incoming data, modify it if necessary, and pass it further to the next module. By properly configuring the switch matrix, modules can be connected to these signal by accessing the data via the flip-flops of the Configurable Logic Block (CLB) slices. Hence, modules occupy signals of the streaming bar to send data to successive modules.

An example for building a system that uses an I/O bar for streaming data through a PR region is shown in Figure 3. In this example, four communication macros are placed which are connected via multiplexers in the static part of the system. Furthermore, three modules are depicted which can subsequently access data streamed via the I/O bar.

4. Streaming Applications with Similarities

In many areas of application, it is necessary to process data streams. This is commonly the case in signal and image processing where a set of filters is applied on the data stream, or in Data Stream Management Systems for querying of data in data streams. We denote these kind of applications as *streaming applications*.

4.1. Application Model. A streaming application can be formally defined by a data flow graph (DFG), which is denoted as $G(V, E)$. Vertices $V = T \cup C$ represent both tasks T and communication nodes C. Edges E represent data dependencies between them. Examples of data flow graphs are given in Figure 4.

Each task $t \in T$ represents a functional IP core, which can be implemented and dynamically loaded as PR module. As described in Section 3.1, PR modules can only be placed at locations which have the same arrangements as their synthesis regions. Therefore, the set $M(t)$ contains all PR modules available for implementing a task $t \in T$. Each PR module $m_i \in M(t)$ is described by a rectangular bounding box denoted by the parameters height(m_i) and width(m_i). The arrangement of reconfigurable tiles within this bounding box is represented as a string, denoted by string(m_i). This is a common approach in reconfigurable computing since string matching techniques can be applied to determine the feasible positions for PR modules during online placement. In addition, the parameter type(m_i) $\in$ {Master, Slave} states whether the module serves as a slave module or a master module, where the latter is able to initialize bus requests. This specification is necessary since communication macros only provide a limited amount of signals for connecting master modules to the arbiter and, thus, impose constraints for the online placement.

Our graph merging algorithm, described in the next section, requires that each data flow graph has an initial *no-operation* (NOP) node NOP_0 which is connected to all processes with no predecessor. Furthermore, a final *no-operation* node NOP_n is required to which all processes with no successor are connected.

Each communication node $c \in C$ has exactly one predecessor module (the sender), but may have several successor modules (the receivers). This paper concentrates on communication of streaming applications established via circuit switching techniques, such as I/O bars. Nonetheless, if on-chip buses are also included in the design, this may influence the placement: since buses allow a mutually exclusive medium access, they have to be arbitrated before usage. Each macro provides a limited number of dedicated signals for connecting master modules with the arbiter. Thus, only a restricted number of master modules can be placed per macro.

As explained in Section 3.2, circuit switching techniques provide a limited number of signals which can be accessed by modules. Since the signals are streamed over the architecture, this communication affects the placement. The parameter com(c) denotes requirements of communication bandwidth c in terms of the number of signals that are required to establish the communication between the sending module pred(c) and the receiving modules succ(c). This number of signals is occupied until they are consumed by the last receiving module.

4.2. Merging Data Flows with Similarities. We propose a heuristic based on [13] for merging two DFGs represented by graphs G_i and G_j which contain some or several processes implemented by the same IP cores. The approach is presented in Algorithm 1. It works by extracting all paths of the data flow graphs. Each path represents a sequence of processes and communication nodes. Then, paths of the two graphs are subsequently merged. The basic functionality is to identify common nodes in the paths, and then, merge paths such that common nodes are only included once. Common nodes are defined in the following manner.

Definition 1. We denote processes of two paths as common processes if they may be implemented by the same IP core.

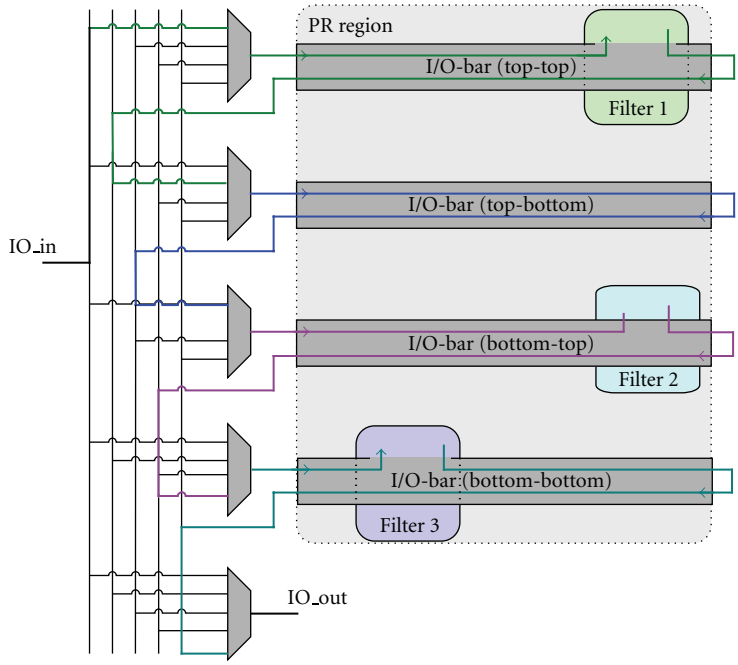

Figure 3: I/O bar used for streaming data through PR region.

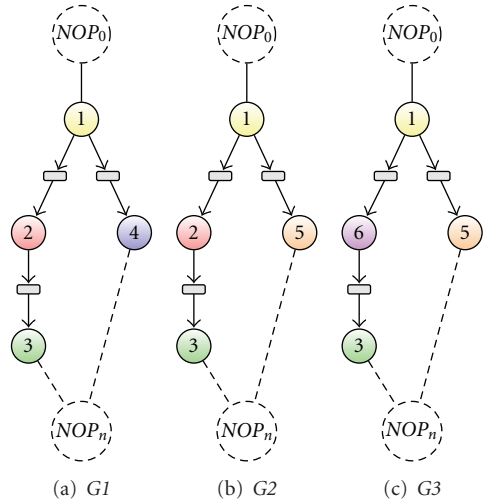

Figure 4: Example of data flow graphs with strong similarities that can be merged.

Moreover, communication nodes of two paths are considered as common communication nodes if they connect senders which are common processes and receivers which are common processes within the paths.

The overall objective is to reduce the reconfiguration time by exploiting similarities. The decision of which paths to merge is therefore based on the idea of [13]. Here, for maximizing the reconfiguration time reduction, the Maximum Area Common Subsequence (MACSeq) and Substring (MACStr) is applied. Each sequence of string consists of a number of components. In the case of [13], not the longest common sequence or string of components in two paths is determined by MACSeq and MACStr, respectively. Rather, the sequence and string determine which components have the highest accumulated area.

In our case, the area is replaced by the reconfiguration time of the PR modules represented by the processes so that sequences and strings with high reconfiguration times are preferred. (Reconfiguration time may depend on the area. In the case of framewise reconfiguration, however, reconfiguration time depends on the width of the PR module.) In the algorithm (lines 4–7), the MACSeq and MACStr $S_{i,j}$ of all pairs of paths are calculated. The pair containing the sequence/string $S_{i,j}^{\max}$ with the maximal accumulated reconfiguration time is selected and merged according to

(1): **initialize** merged graph G_M;
(2): **extract** all paths $P_{i/j}$ of $G_{i/j}$;
(3): **While** at least one pair of paths with non-empty MACSeq or MACStr exists **do**.
(4): **calculate** MACSeq/MACStr $S_{i,j}$ for each pair $(p_i, p_j) \in P_i \times P_j$;
(5): **find** pair (p_i, p_j) with $S_{i,j}^{\max}$ having maximum reconfiguration time;
(6): **merge** paths p_i and p_j according to $S_{i,j}^{\max}$ and add them to G_M;
// see Algorithm 2
(7): $P_{i/j} = P_{i/j} \setminus p_{i/j}$;
(8): **end while**
(9): **add** all remaining paths $p \in P_i$ and $p \in P_j$ as or-paths to G_M;
// see Algorithm 3

Algorithm 1: Merging two data flow graphs G_i and G_j.

(1): **ensure** G_M contains the NOP nodes NOP_0 and NOP_n;
(2): $v_0 = NOP_0$
(3): **for** $k = 0$ **to** $|S_{i,j}|$ **do**
(4): $(v_i^{(k)}, v_j^{(k)}) = S_{i,j}[k]$; // the k-th pair in the sequence
(5): **determine** or-paths $or_{i/j}$, that is, all vertices in $p_{i/j}$ between $v_{i/j}^{(k-1)}$ and $v_{i/j}^{(k)}$
(6): **add** $or_{i/j}$ to G_M; // see Algorithm 3
(7): **make** new node n_M;
(8): **link** $v_M[i]$ to $v_i^{(k)}$ and $n_M[j]$ to $v_j^{(k)}$;
(9): **add** v_M to G_M;
(10): **add** edge to G_M connecting the last vertices added from p_i and p_j to v_M;
(11): $v_0 = v_M$;
(12): **end for**

Algorithm 2: Merging two paths p_i and p_i according to common sequence $S_{i,j}$ and adding them to the merged graph G_M.

Algorithm 2. When no further paths with similarities exist, the remaining paths are added to the merge graph (line 9).

Path merging works as shown in Algorithm 2. Here, $(v_i^{(k)}, v_j^{(k)}) = S_{i,j}[k]$ denotes the kth pair of common nodes in the sequence (line 4). All common nodes are successively added to the merged graph G_M. The nodes of the paths which are not part of the sequence/string and lie between common nodes are denoted as *or-paths* (line 5) and added separately according to Algorithm 3 (line 6). When merging a pair of common nodes, a new node is added to the merged graph which represents the common node. It contains references to the common nodes and is connected with the last vertices from the paths added to the merged graphs (lines 7–10).

Algorithm 3 shows how to add an or-path to the merged graph. Here, the elements in the or-paths are iteratively added. If the current element $v_i^{(k)}$ is not already part of the merged graph G_M, a new node is added to G_M with a reference to $v_i^{(k)}$ (lines 3–6). Then, the last element of the or-path added to G_M is connected to this new node in G_M (line 10).

The proposed heuristic can now be used to subsequently merge several data flow graphs. In all cases, it is required that all graphs have a initial NOP node NOP_0 and a final NOP node NOP_n. As an example, Figure 5 shows the data flow graph resulting when merging the three graphs from Figure 4.

5. Problem Formulation

Placing tasks on a reconfigurable device is a complex problem, in which many resource constraints must be respected. In our scenario, one important limited resource consists in the communication infrastructure, for example, the signals available for circuit switching. Typically, the amount of used connection bandwidth at each position and instant in time must not exceed a certain maximum number. Therefore, it may be necessary to delay the placement of one module until some more communication bandwidth is released. Furthermore, the data dependencies must be respected, that is, all the temporary results on which one tasks depends must have been already computed before it can be executed. In addition, each module may require Block-RAM resources at specific relative positions; thus the Block-RAM resource limitations must also be respected for the task placement. In a more formal manner, the placement problem can be specified as follows.

Given a task graph $G_T(V_T, E_T)$ and a tiled reconfigurable architecture where A represents the set of reconfigurable tiles,

find for each process $t \in T$ with outgoing communication nodes $C_t^{\text{out}} = \text{succ}(t)$ and incoming communication nodes $C_t^{\text{in}} = \text{pred}(t)$ a feasible location $F \subseteq A$

```
(1): for k = 0 to |or_i| do
(2):     v_i^(k) = or_i[k];
(3):     if v_i^(k) is not already part of G_M then
(4):         make new node v_M;
(5):         link v_M[i] to v_i^(k);
(6):         add  v_M to G_M;
(7):     else
(8):         get node v_M in G_M where v_M[i] = v_i^(k);
(9):     end if
(10): add edge (v_0, v_M) to G_M;
(11):    v_0 = v_M
(12): end for
```

Algorithm 3: Adds an or-path or_i to the merged graph G_M. The or-path is part of the path p_i. The last vertex of the path p_i added to G_M is denoted by v_0. If no such vertex is given, $v_0 = NOP_0$ is used as default, that is, the NOP node of G_M.

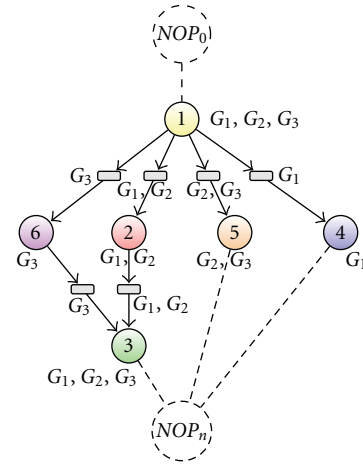

Figure 5: Example of the merge graph of the data flow graphs G_1, G_2, and G_3 from Figure 4. Each node of the merge graph holds a reference to the data flow graph it is a part of.

such that there exists an implementation of t as PR module $m_i \in M(t)$

subject to the following conditions:

(i) height(F) = height(m_i) and width(F) = width(m_i): the module and the selected region have the same bounding box;

(ii) string(F) = string(m_i): the module and the selected region have the same underlying arrangement;

(iii) for all $c \in C_t^{in}$: all predecessors pred(c) have already been placed, and there exists a valid route between their location and F;

(iv) Master(row_y) < MaxMaster Nbr(row_y), where row_y denotes the communication macro accessed via location F. Master(row_y) counts the number of masters already placed at macro row_y, and MaxMasterNbr(row_y) gives the number of links provided by the macro to connect to the arbiter. This constraints take into account the limited number of master modules allowed per macro.

(v) The number of required output lines $\text{com}_{\text{out},t}$ with

$$\text{com}_{\text{out},t} = \sum_{c \in C_t^{\text{out}}} \text{com}(c) \tag{1}$$

may not exceed the number of freely available communication bandwidth, such that there exists sufficient signals to route the communication.

6. Algorithm

A placement algorithm decides where and at what moment in time a module is loaded and executed. In the online case, information about all tasks to be placed is not known from the beginning, but tasks arrive over time. Our proposed online placement is given as Algorithm 4. In each iteration, a module candidate is determined out of the list of modules to be placed. Therefore, it is checked for each module if all its predecessors are already placed. In the case that this is fulfilled, the communication bandwidth requirements, for example, streaming bar signals, are checked. Either, there must be enough free bandwidth available, or bandwidth claimed by some predecessor can be freed, because it is no longer necessary as the data end points have already been placed. If there are multiple modules which all fulfill these requirements, then a priority-based approach is used to determine the selection of the module. Usually modules with more descendants in their data flow graph receive a higher priority. Furthermore, the priorities can also depend on user-defined objectives or the underlying architecture. The priorities are dynamic, meaning that they are determined new in each interation.

An important step of the algorithm is to determine the possible feasible positions. Hereby, the candidate rectangles are dependent on the underlying architecture graph. As described in Section 3, heterogeneities on the reconfigurable device partition the reconfigurable area. Based on that, an initial list of possible rectangles is created. Hereby, rectangles can have different heights, measured in resource slots, and are sorted in the order of their appearance in the architecture graph. A rectangle is determined in the lists by the *First-Fit* rule, that is, the first rectangle found in which the module can be placed. Thus, modules are tried to be placed near the origin at coordinate $(0, 0)$ to reduce fragmentation. Data dependencies are respected by first checking for each module if all predecessing modules are already placed. Special care must be taken for modules with Block-RAM requirements which are represented with a string and are checked by a string matching approach.

The complexity of the algorithm is an important issue, because it is not performed offline, but during the runtime. The overhead of performing the algorithm contributes to the overall execution time. Therefore, we kept an eye on

Initialize list of rectangles, communication bandwidth usage;
for $i = 1$ to $|M|$ **do**
  Determine set of modules for which data-dependencies are met, and still unplaced, named K_i;
  Remove modules from K_i, for which currently not enough communication bandwidth is available;
  Remove modules from K_i, for which no suitable feasible position can be found in G_A with a first-fit heuristic.
  If more than one module is still included in K_i, then select one according to criteria such as
  number of descendants, module size, Block-RAM usage, and so forth.
  Place K_i in the determined feasible position and update resource usage informations;
end for.

Algorithm 4: Placement Algorithm.

the performance of the algorithm and followed a list-based technique. The algorithm consists of a loop with various checks for suitability over all still available tasks. More precisely, it has a running time of $O(|M|^2)$. Thus, it can be executed efficiently also at runtime. Other offline approaches, like [14], might achieve better solutions, but are to complex too be employed in an online setting, where the tasks to be considered might also change over runtime, and downtime of the system must be minimized. Thus, new modules can be developed and be integrated in the running reconfigurable system, because the placement algorithm is fast and flexible enough to adapt at runtime to the new situation.

However, when placing a merge graph, special care must be taken. In this case, we start placing the modules from the left top-most position towards the right bottom-most position. Furthermore, we keep pointers for all streaming applications that are aggregated into the merge graph. Each pointer indicates the position from which the First-Fist heuristic starts searching. Each time a module is selected according to Algorimth 4, the heuristic starts searching from that pointer that is most advanced and belongs to an application the module is part of. After the module is placed, all affected pointers are advanced onto the first free position after the bounding box of this module. By successively performing this step all modules are placed, and amodule running mutually exclusive may use the same resources.

7. Case Study

Figure 6 illustrates the FPGA-based system on chip from [15] which is implemented on a Xilinx Virtex-II XUP Pro board and serves as a case study. The two dark-red areas on the right top and bottom compose the dynamic part of the system. Reconfiguration is only possible in the dynamic part which contains a reconfigurable on-chip bus and streaming bars as communication primitives. Both primitives are generated by the framework ReCoBus-Builder [5]. Note that the partial modules possess different sizes and that modules can be placed in a two-dimensional manner since four macros are placed in the partial part.

The proposed placement algorithm was implemented for the described SoC. The procedure of runtime reconfiguration is illustrated in Figure 7, highlighting the steps required to dynamically load PR modules. Reconfiguration is basically performed by keeping an image of the current FPGA configuration in the memory. This is used as a virtual representation of the FPGA. When a PR module is loaded, its bitfile is loaded from the memory and combined with the virtual representation (see steps 1 to 4 in Figure 7). The module is then physically loaded by writing the modified frames of this bitfile to the FPGA via the ICAP interface (see steps 5 to 8 in Figure 7). The use of the virtual representation is necessary since reconfiguration happens in a framewise manner, and thus, also logic from the static part has to be rewritten when placing a new PR module. Details of this procedure are described in [15].

In addition to the static bitfile and the bitfiles of the PR modules, the CF card also contains the application descriptions provided in XML format. This XML format allows to described the application by specifying the nodes and the edges of the data flow graphs or merge graphs, respectively. Each node is associated with the required parameters, such as the name of the bitfile of the PR module and the parameters specified in Section 4. When loading a new application, the XML description is first loaded to initialize the placement algorithm. Then, candidate modules are selected and positions determined as described in the last section.

8. Experiments

We have performed several experiments in order to evaluate the proposed approach. The architecture from the case study was chosen to perform the experiments. Furthermore, 6 data flow graphs were generated, consisting of 5 to 15 nodes. Processes were chosen from 5 different IP block types, where each type is associated with a PR module, specified by its bounding box and arrangement as well as a reconfiguration time which is proportional to the PR module width. These data flow graphs are chosen according to image processing applications already implemented on the architecture described in Section 7. Details of this kind of filters and data flows are presented in [15, 16]. As shown there, the input image can be sent pixel by pixel via the I/O bar. The pixels can then be accessed by image filters which are loaded as PR modules. The width of the I/O bar chosen in our experiments is 32 bit. The test case applications are assumed to work on grayscale input images. Therefore, the required output lines per communication node are set to 8 bit.

Now, the presented placement approach allows to place such applications by respecting the placement constraints

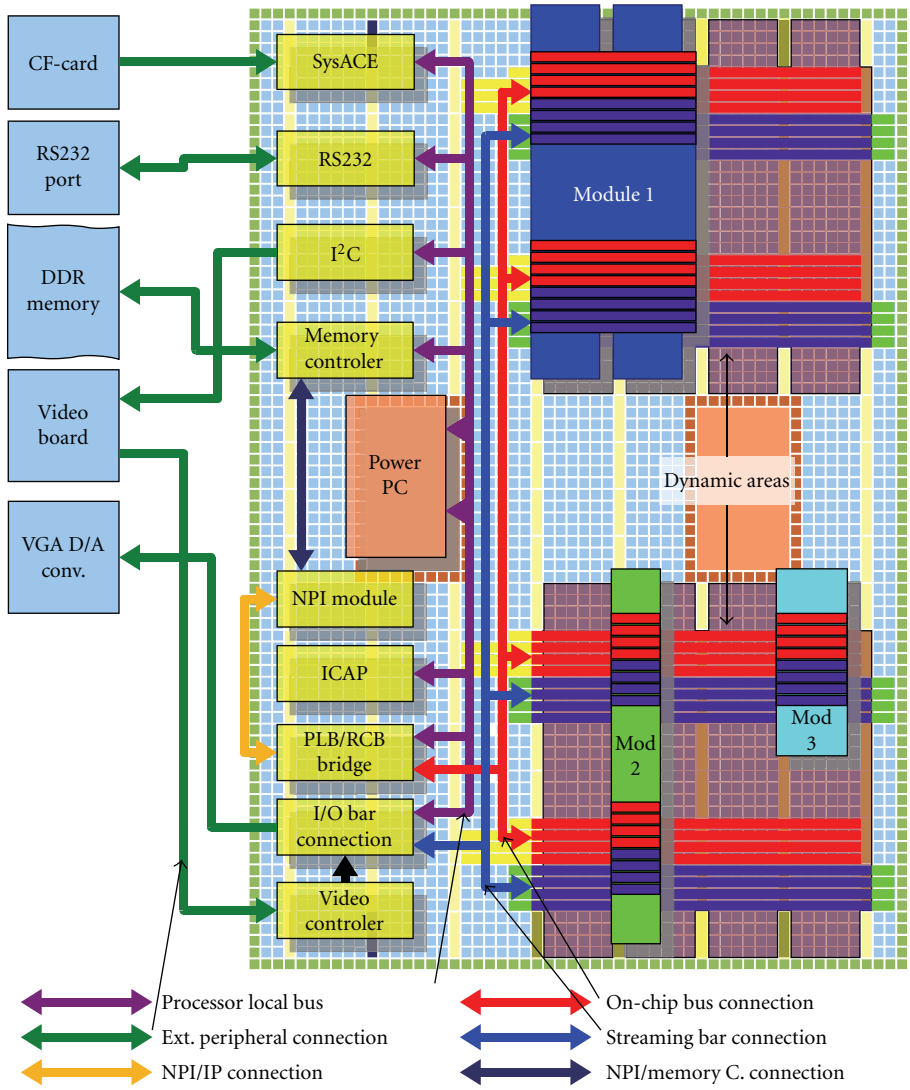

Figure 6: System overview of one possible partitioning of a heterogeneous FPGA-based SoC platform consisting of CPU subsystem and reconfigurable area from [15]. Reconfigurable modules can vary in size and be freely placed, allowing a very good exploitation of the FPGA space. The on-chip ReCoBus (red) and the streaming bar (blue) are routed through the dynamic part, allowing a systemwide communication between hardware and software modules.

as well as the data dependencies expressed by the data flow graph, so that pixels are accessed and processed in the correct order. Furthermore, when switching between image processing applications, our merging approach allows to reduce the required reconfiguration time. This is analyzed in the following experiments.

We have chosen the following metrics to compare our test runs. Let $\mathcal{G}$ denote the set of data flow graphs which are merged into graph $G_M(V_M, E_M)$ according to Algorithm 1. Then, the *similarity* s_0 is measured according to

$$s_1 = \frac{\sum_{G(V,E)\in\mathcal{G}} |V| - |V_M|}{\sum_{G(V,E)\in\mathcal{G}} |V|},$$
$$s_0 = \frac{s_1}{(|\mathcal{G}| - 1)/|\mathcal{G}|}, \tag{2}$$

where $0 \leq s\_1 \leq (|\mathcal{G}| - 1)/(|\mathcal{G}|)$ measures how much we can reduce the number of nodes by merging the graphs $G(V, E) \in \mathcal{G}$. In the best case, all graphs would be exactly the same and consist of vertex set V. Here, s_1 would have the value $(|V| \cdot |\mathcal{G}| - |V|)/(|V| \cdot |\mathcal{G}|)$. In the worst case, no similarities do exist and s_1 would be 0. So, the similarity $s_0 \in [0, 1]$ measures how far we are away from the best case.

The area rate indicates how much the area required for placing the merged graph differs from the area required when not merging the graphs. In the second case, resource sharing is enabled so that graphs are placed separately and may also occupy the resource used by the other graphs. Each graph $G \in \mathcal{G}$ is placed separately according to the proposed algorithm, and the area requirements $area_G$ stemming from this placement is calculated. Then, the maximal area $\text{area}_{|\mathcal{G}|}^{\max} = \max_{G\in\mathcal{G}}\{\text{area}_G\}$ is determined. We have

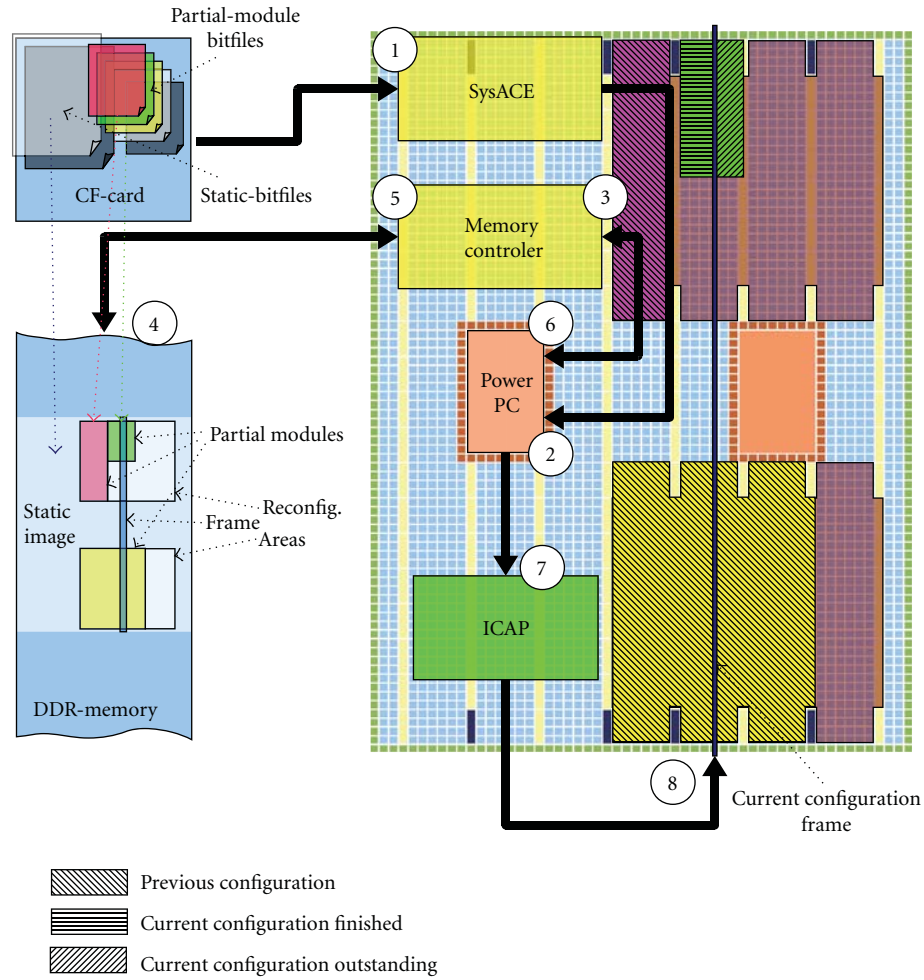

Figure 7: Reconfiguration procedure.

furthermore calculated the area required for placing the merged graph according to the described algorithm. This area is denoted as area_{G_M}. The *area rate* is given as

$$\frac{\text{area}_{G_M}}{\text{area}^{\max}_{|\mathcal{G}|}}. \tag{3}$$

The *reconfiguration time rate* states how much we can reduce the reconfiguration time when merging the graphs. For each pair $G_1, G_2 \in \mathcal{G}$ with $G_1 \neq G_2$, we calculate the reconfiguration time required for switching from G_1 to G_2 (a) by completely placing G_2 without exploiting similarities of the data flow graphs and (b) by using the merged graph and only loading the differing PR modules. By dividing the maximal values, we obtain the reconfiguration time rate.

In the first set of experiments, we have generated all $\binom{6}{2} = 15$ pairs for the data flow graphs. Each pair was merged according to the proposed algorithm and evaluated according to the metrics described above. The results are shown in Table 1 and indicate a correlation between the similarity of graphs and the potential to reduce the reconfiguration time. Test-case A2 has the biggest similarity and lowest reconfiguration time rate, thus reducing the reconfiguration time by 76% when applying the merged graph. However, the similarity metric does not consider the reconfiguration time of the merged processes. So, it is not possible to directly relate this metric with the reconfiguration time rate. This can, for example, be seen in test-cases A7 and A8. A7 has the minimal similarity but using the merged graph reduces the reconfiguration time by 17%. In contrast, A8 has a higher similarity than A7. Nonetheless, using merge graphs only reduces the reconfiguration time by 8%. Still, the experiments show the benefits of the proposed approach. Reconfiguration times can be reduced in all cases, ranging from 6% to 76% for the test cases.

The second observation is that, in most cases, there is no difference between separate placement and using the merged graph regarding the required area. In some cases, the area requirement can even be reduced. Then, in one case, it is larger than placing the graphs separately. This difference stems from the fact that the candidate list used in Algorithm 4 differs in the approaches. This affects the order in which PR modules are placed and may consequently result in different area requirements.

In the second set of experiments, we have generated all 20 combinations when choosing 3 out of the 6 data flow graphs. The graphs were merged and again evaluated

Table 1: Overview of the results of the first set of experiments with two graphs merged.

test case	No. of graphs	similarity	area rate	reconf. rate
A1	2	0.29	1.0	**0.94**
A2	2	**0.89**	1.0	**0.24**
A3	2	0.80	0.86	0.47
A4	2	0.67	1.00	0.75
A5	2	0.67	1.00	0.43
A6	2	0.25	**0.62**	0.76
A7	2	**0.22**	0.86	0.83
A8	2	0.40	1.00	0.92
A9	2	0.25	1.00	0.70
A10	2	0.73	0.86	0.40
A11	2	0.57	1.00	0.81
A12	2	0.40	1.00	0.48
A13	2	0.50	**1.16**	0.87
A14	2	0.55	1.00	0.57
A15	2	0.29	1.00	0.87

Table 2: Overview of the results of the second set of experiments with three graphs merged.

Test case	No. of graphs	similarity	Area rate	Reconf. rate
B1	3	0.63	**0.62**	0.95
B2	3	0.58	1.00	**0.97**
B3	3	0.50	1.00	0.94
B4	3	0.50	1.00	0.70
B5	3	0.80	0.86	0.57
B6	3	**0.82**	1.00	0.81
B7	3	0.75	1.00	**0.43**
B8	3	0.75	1.00	0.87
B9	3	0.70	0.73	0.57
B10	3	0.68	1.00	0.96
B11	3	0.64	**1.16**	0.83
B12	3	0.45	1.00	0.81
B13	3	0.46	1.00	0.70
B14	3	**0.41**	0.86	0.87
B15	3	0.43	1.00	0.83
B16	3	0.45	1.00	0.87
B17	3	0.69	1.00	0.87
B18	3	0.66	1.00	0.57
B19	3	0.63	1.00	0.96
B20	3	0.58	**1.16**	0.87

according to the chosen metrics. The results are shown in Table 2. We see again that there is no direct proportionality of reconfiguration time rate and similarity, since the latter only measures the similarity of the graph topology. In all test cases, we are able to reduce the reconfiguration time. In the best case, this are 57% for B7, and in the worst case, 3% for B2. Again, the area requirements stay the same for most test cases. However, in B11 and B20, the requirements grow by 16%. Then again, the requirements can be reduced by 38% in the case of B1.

All in all, the experiments show the benefits regarding the reconfiguration time when using our approach. In the best case, we could reduce the time by 76% for the first set of experiments, and by 57% for the second set. The experiments have also shown, that the area requirements are only affected in some cases. This is due to the fact, that the candidate lists used by the proposed First-Fit heuristic differ when placing the graphs separately, or as a merged graph. Here again, the replacement of the proposed First-Fit algorithm by a more sophisticated search would probably lead to a convergence towards 1.00.

9. Conclusion

Partial reconfiguration of FPGAs allows to dynamically change the configuration of a hardware system at runtime. By applying this technique, it is possible to perform resource sharing between streaming applications which are part of different operational mode. While this allows to build smaller systems, the time and power needed to perform hardware reconfiguration increases. In this case, it is wise to keep similar modules and only replace the differing parts.

This paper deals with merging streaming applications which into a single representation, denoted as merge graph. Besides an algorithm to perform this step, architectural issues of partial reconfiguration and on-chip communication are discussed, and an architecture is described which contains techniques to deal with this. In addition, the paper presents a placement algorithm for streaming applications and merge tasks. In order to be applied in real world, the placement approach takes into account the heterogeneities of the reconfigurable device and respects the bandwidth constraints of the communicating processes. The complexity of the proposed algorithm allows the execution to be performed efficiently at runtime. Of course, offline approaches [14] might find better solutions, but are to complex to be employed in an online scenario.

The experiments show that the proposed approach allows fast switching between streaming applications. In all cases, we were able to reduce the reconfiguration time. In the best-case, the reconfiguration time could even reduce by 76%. The experiments furthermore show that the area requirements do not degenerate by placing the merge graphs approach for a majority of the test cases. In some cases, it even gets better than placing the streaming applications separately. Here, however, the results demonstrate that, due to varying candidate list resulting from separate or merged placement, the proposed First-Fit heuristic is very sensitive on changes in the placement order of PR modules. In future work, the heuristic will therefore be replaced by a more stable placement technique.

References

[1] S. Wildermann, A. Oetken, J. Teich, and Z. Salcic, "Self-organizing computer vision for robust object tracking in smart

cameras," in *Proceedings of the 7th International Conference on Autonomic and Trusted Computing (ATC '10)*, vol. 6407 of *Lecture Notes in Computer Science*, pp. 1–16, Springer, 2010.

[2] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, "Design of homogeneous communication infrastructures for partially reconfigurable FPGAs," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '07)*, pp. 238–247, Las Vegas, Nev, USA, June 2007.

[3] D. Koch, C. Haubelt, and J. Teich, "Efficient reconfigurable on-chip buses for fpgas," in *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 287–290, April 2008.

[4] H. ElGindy, H. Schroder, A. Spray, A. K. Somani, and H. Schmeck, "RMB—a reconfigurable multiple bus network," in *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA '96)*, pp. 108–117, February 1996.

[5] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder—a novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 119–124, September 2008.

[6] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "DyNoC: a dynamic infrastructure for communication in dynamically reconfigurable devices," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 153–158, August 2005.

[7] U. Malik and O. Diessel, "On the placement and granularity of FPGA configurations," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 161–168, December 2004.

[8] K. P. Raghuraman, H. Wang, and S. Tragoudas, "A novel approach to minimizing reconfiguration cost for LUT-based FPGAs," in *Proceedings of the 18th International Conference on VLSI Design: Power Aware Design of VLSI Systems*, pp. 673–676, January 2005.

[9] N. Shirazi, W. Luk, and P. Y. K. Cheung, "Automating production of run-time reconfigurable designs," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pp. 147–156, IEEE Computer Society, Napa Valley, Calif, USA, 1998.

[10] M. Rullmann and R. Merker, "Design methods and tools for improved partial dynamic reconfiguration," in *Dynamically Reconfigurable Systems—Architectures, Design Methods and Applications*, pp. 147–163, Springer, Berlin, Germany, March 2010.

[11] J. Angermeier, S. Wildermann, E. Sibirko, and J. Teich, "Placing streaming applications with similarities on dynamically partially reconfigurable architectures," *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 91–96, 2010.

[12] D. Koch, C. Beckhoff, and J. Teich, "A communication architecture for complex runtime systems and its implementation on spartan-3 FPGAs," in *Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, pp. 253–256, February 2009.

[13] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *Proceedings of the 41st Design Automation Conference (DAC '04)*, pp. 395–400, June 2004.

[14] S. Wildermann, F. Reimann, D. Ziener, and J. Teich, "Symbolic design space exploration for multi-mode reconfigurable systems," in *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '11)*, pp. 129–138, 2011.

[15] A. Oetken, S. Wildermann, J. Teich, and D. Koch, "A bus-based SoC architecture for flexible module placement on reconfigurable FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 234–239, Milano, Italy, August 2010.

[16] D. Ziener, S. Wildermann, A. Oetken, A. Weichslgartner, and J. Teich, "A flexible smart camera system based on a partially reconfigurable dynamic FPGA-SoC," in *Proceedings of the Workshop on Computer Vision on Low-Power Reconfigurable Architectures at the FPL*, Chania, Greece, September 2011.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 872610, 11 pages
doi:10.1155/2012/872610

Research Article

Redsharc: A Programming Model and On-Chip Network for Multi-Core Systems on a Programmable Chip

William V. Kritikos,[1] Andrew G. Schmidt,[1] Ron Sass,[1] Erik K. Anderson,[2] and Matthew French[2]

[1] *Reconfigurable Computing Systems Laboartory, ECE Deptartment, UNC Charlotte, 9201 University City Boulevad, Charlotte, NC 28223, USA*
[2] *Information Sciences Institute, University of Southern California, 3811 North Fairfax Drive, Suite 200, Arlington, VA 22203, USA*

Correspondence should be addressed to William V. Kritikos, will.kritikos@gmail.com

Received 6 May 2011; Revised 21 July 2011; Accepted 28 September 2011

Academic Editor: Claudia Feregrino

The reconfigurable data-stream hardware software architecture (Redsharc) is a programming model and network-on-a-chip solution designed to scale to meet the performance needs of multi-core Systems on a programmable chip (MCSoPC). Redsharc uses an abstract API that allows programmers to develop systems of simultaneously executing kernels, in software and/or hardware, that communicate over a seamless interface. Redsharc incorporates two on-chip networks that directly implement the API to support high-performance systems with numerous hardware kernels. This paper documents the API, describes the common infrastructure, and quantifies the performance of a complete implementation. Furthermore, the overhead, in terms of resource utilization, is reported along with the ability to integrate hard and soft processor cores with purely hardware kernels being demonstrated.

1. Introduction

Since the resources found on FPGA devices continue to track Moore's Law, modern, high-end chips provide hundreds of millions of equivalent transistors in the form of reconfigurable logic, memory, multipliers, processors, and a litany of increasingly sophisticated hard IP cores. As a result, engineers are turning to multi-core systems on a programmable chip (MCSoPC) solutions to leverage these FPGA resources. MCSoPC allow system designers to mix hard processors, soft processors, third party IP, or custom hardware cores all within a single FPGA. In this work, we are only considering multi-core systems with a single processor core, multiple third party IP cores, and multiple custom hardware cores.

A major challenge of MCSoPC is how to achieve intercore communication without sacrificing performance. This problem is compounded by the realization that cores may use different computational and communication models; threads running on a processor communicate much differently than cores running within the FPGA fabric. Furthermore, standard on-chip interconnects for FPGAs do not scale well and cannot be optimized for specific programming models; contention on a bus can quickly limit performance.

To address these issues, this paper investigates Redsharc—an API and common infrastructure for realizing MCSoPC designs. Redsharc's contribution has two parts.

First, introduction of an abstract programming model and API that specifically targets MCSoPC is presented. An abstract API, as described by Jerraya and Wolf in [1], allows cores to exchange data without knowing how the opposite core is implemented. In a Redsharc system, computational units, known as kernels, are implemented as either software threads running on a processor, or hardware cores running in the FPGA fabric. Regardless of location, kernels communicate and synchronize using Redsharc's abstract API. Redsharc's API is based both on a streaming model to pass data using unidirectional queues and a block model that allows kernels to exchange index-based bidirectional data. Section 3 explains the Redsharc API in detail.

Redsharc's second contribution is the development of fast and scalable on-chip networks to implement the Redsharc

API, with special consideration to the hardware/software nature of MCSoPC. The stream switch network (SSN), discussed in Section 4.1, is a run-time reconfigurable crossbar on-chip network designed to carry streams of data between heterogeneous cores. The block switch network (BSN), discussed in Section 4.2, is a routable crossbar on-chip network designed to exchange index data elements between cores and blocks of memory.

To evaluate the proposed approach, the API and infrastructure have been implemented as a software library and a collection of VHDL components with generics. A series of benchmarks were run on a Xilinx ML510 development board to demonstrate Redsharc's performance. A BLAST bioinformatics kernel was also ported to a Redsharc core to demonstrate the full use of the Redsharc API and to show scalability. Results are listed in Section 6.

2. Related Work

The idea of creating an API to abstract communication between heterogeneous computational units has its roots in both the DARPA adaptive computing systems (ACSs) project [2] and PipeRench project [3]. These projects focused on processor to off-chip FPGA accelerator communication and control, they existed prior to the realization of MCSoPC.

More recently, ReconOS [4] and hthreads [5] implemented a programming model based on Pthreads to abstract the hardware/software boundary. They successfully demonstrate the feasibility of abstract programming models in heterogeneous systems. However, their middleware layer requires an impractical amount of FPGA resources. Furthermore, their communication bandwidth is limited due to a reliance on proprietary buses. Redsharc addresses these problems by developing an abstract API better suited for MCSoPC and developing custom on-chip networks, respectively, supporting the API.

Using a message passing interface (MPI) for parallel communication has also been explored on FPGAs. MPI is commonly used in scientific applications running on large clusters with traditional microprocessor-based nodes. Several works, including TMD-MPI [6] and SoC-MPI [7], have implemented some of the standard MPI function calls in an FPGA library. While supporting MPI may be useful when trying to port an existing scientific application to an FPGA cluster, it adds several layers of abstraction that are not necessary for streaming models. Some logic must be present to decode received message, strip out the real data from the message, and deliver that data to the appropriate input of the computational unit. The streaming model allows for direct connection of a stream to computational units, as the stream contains only data. We depend on the streams being properly set up before data is delivered. The goal of Redsharc is to implement as light-weight as possible, so a streaming system was chosen over an MPI or Pthreads-based system.

The stream model is common within FPGA programming. Projects such as [8, 9] use high level stream APIs or stream languages with compilation tools to automatically generate MCSoPC. Unlike Redsharc, these efforts focus on a purely streaming programming model and do not include support for random access memory. Furthermore, these models do not permit the mixing of heterogeneous hard processors, soft processors, or custom FPGA cores as may be needed to meet stringent application demands.

Researchers have also customized streaming networks to communicate between heterogeneous on-chip cores using abstract interfaces. For example, aSOC [10] is notable for communication between heterogeneous computational units, but it targets ASICs instead of FPGAs. Finally, SIMPPL [11] developed an FPGA-based streaming network but do not consider heterogeneous computational units. Moreover, existing on-chip networks only support pure streaming applications and do not include support for random access memory.

3. Redsharc API

Developing an abstract API for MCSoPC is not a trivial task. The API must support parallel operations, be realizable in all computational domains, be resource friendly, and be flexible enough to incorporate a large number of application domains. However, these goals can be in conflict at times.

To find the middle ground in these conflicting requirements, Redsharc is based on the stream virtual machine (SVM) API [12]. SVM was originally designed as an intermediate language between high level stream languages and low level instruction sets of various architectures being developed by the DARPA polymorphous computing systems (PCSs) program. The main idea was that multiple high-level languages would map to a common stable architectural abstraction layer that included SVM. That layer would then map to specific polymorphous computing systems such as TRIPS, MONARC, RAW, and others as illustrated in Figure 1 [13–15]. SVM has no preference to the computational model for individual kernels and only specifies how kernels communicate with each other. SVM is primarily based on the relatively simple stream model, but it includes the concept of indexed block data to support random access. These features make it an ideal candidate for porting to MCSoPC. The REDSHARC system implements the same SVM API that would be used on a TRIPS, MONARCH, or RAW architectures while targeting FGPAs instead of ASIC processors. Future work could allow the high level languages shown in Figure 1 to target FPGA-, TRIPS-, MONARCH-, and RAW-based processors.

The Redsharc API recognizes two types of kernels, worker and control. There is only one control kernel in the system and it is responsible for creating and managing streams, blocks, and worker kernels. There may be multiple worker kernels that perform functions on data elements presented in streams or blocks. A stream is a unidirectional flow of data elements between two kernels. A block is an indexed set of data elements shared between two or more kernels. In the current Redsharc implementation, a data element can be of any size 2^n bits where n is an integer and greater than 5.

Applications are divided into kernels. Because both hardware and software kernels utilize the same block and stream API, the initial stages of application development do not need to be concerned with a particular kernel

Table 1: Redsharc control kernel's API calls.

Control's API call	Description
`kernelInitNull (kernel *k)`	Initializes `k` to a default state
`kernelInit (kernel *k, streams *s, blocks *b)`	Configures streams/blocks in array `s/b` for communication with `k`
`kernel Add Dependence (kernel *k1, kernel *k2)`	Makes `k2` dependent on the completion of `k1`
`kernelRun(kernel *k)`	Adds `k` to the list of schedulable kernels
`kernelPause(kernel *k)`	Removes `k` from the list of schedulable kernels
`streamInitFifo(stream *s)`	Initializes `s`
`blockInit(block *b)`	Initializes `b`

Table 2: Redsharc worker kernel's API calls.

Worker's API Call	Description
`void kernelEnd()`	Indicates kernel has completed its work
`void streamPush(element *e, stream *s)`	Pushes `e` onto a stream `s`
`void streamPushMulticast(element *e, stream *s)`	Pushes `e` to multiple streams
`void streamPop(element *e, stream *s)`	Pops top element from `s` and stores value in `e`
`void streamPeek(element *e, stream *s)`	Reads top element from `s` storing value in `e`
`void blockWrite(element *e, int index, block *b)`	Writes `e` to `b` at index
`void blockRead(element *e, int index, block *b)`	Reads `index b` and stores in `e`

being implemented in software or hardware. The system can be viewed as kernels with logical connections as shown in Figure 2. After each kernel is defined, the kernels are implemented using either hardware or software, depending on the suitability of the kernel's task to a software or hardware environment. At that point in time, the system will resemble Figure 3.

3.1. Software Kernel Interface. There are two types of software kernels in Redsharc, worker kernels which perform work for the application, and a single control kernel which sets up the blocks, streams, and which manages the other kernels. Control kernels are always implemented in software. The software kernel interface (SWKI) is implemented as a traditional software library. User kernels link against the library when generating executable files. Different processors (hard or soft) may implement the SWKI in different ways, however, a common approach has been to use a threading model to manage multiple software kernels executing on a single processor.

The control kernel API, in C syntax, is presented in Table 1. The symbols, `kernel`, `stream`, `block`, and `element`, are variable types in Redsharc. The control kernel creates variables of these types and initializes them for use during run-time. In Redsharc, the streams and blocks a kernel communicates with are set at run-time with the `kernelInit` command. A dependency is used in conjunction with a block to allow one kernel to write elements to a block and prohibit any reading kernel from starting until the write is complete. Only the control kernel is aware of each worker kernel's location, either hardware or software. Due to the added complexity control, kernels may only be implemented in software.

The worker kernel API is listed in Table 2. While `kernelEnd` is used to communicate with the control kernel when work is completed, the remainder of the API is dedicated to stream or block communication. In much the same way that a function in an object-oriented language may be overloaded for different variable types, the Redsharc API is independent of data element width, block location, or the transmitting or receiving kernel's implementation.

3.2. Hardware Kernel Interface. The hardware API layer, known as the hardware kernel interface (HWKI), is implemented as a VHDL entity that is included during synthesis. There is one HWKI for each hardware kernel. The HWKI is a thin wrapper that connects hardware kernels to the SSN and BSN. Described in more detail in Section 4, the HWKI implements the Redsharc stream API as a series of interfaces similar to FIFO ports, and the Redsharc block API as a series of interfaces similar to BRAM ports. The use of SVM has simplified the development of hardware kernels. Figure 4 illustrates the HWKI as implemented in VHDL. A kernel developer only needs to know how to pop and push data from an FIFO to use the Redsharc stream switch network. Similarly, all accesses to memory, on-chip, or off-chip, are only a simple BRAM access with additional latency.

4. Redsharc's Networks on a Chip

The stream switch network (SSN) and block switch network (BSN) were necessitated by performance and scalability. Programming models that are built on top of existing general-purpose on-chip networks such as IBM's CoreConnect [16] must translate higher level API calls to lower level network procedures often with a large overhead. For example, even a simple `streamPeek()` operation may require two bus read transactions, the first to check if the stream is not empty, the second to retrieve the value of the top element. In previous work, Liang et al., [10] showed custom on-chip networks

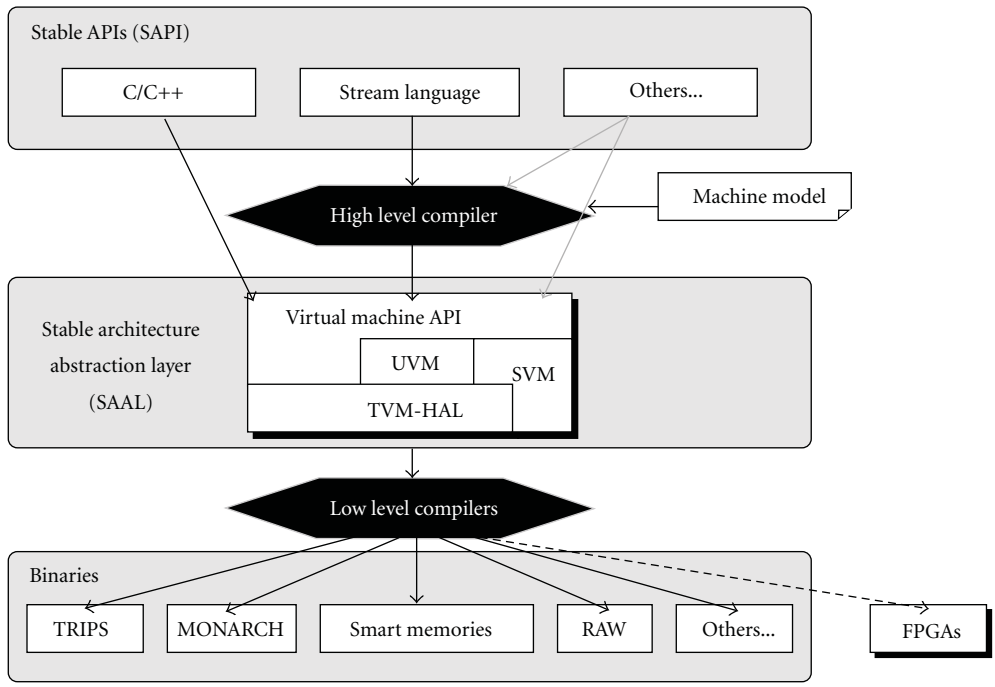

Figure 1: Layers in PCS program.

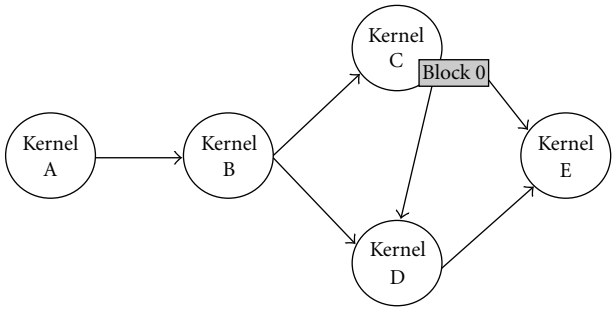

Figure 2: Logical view.

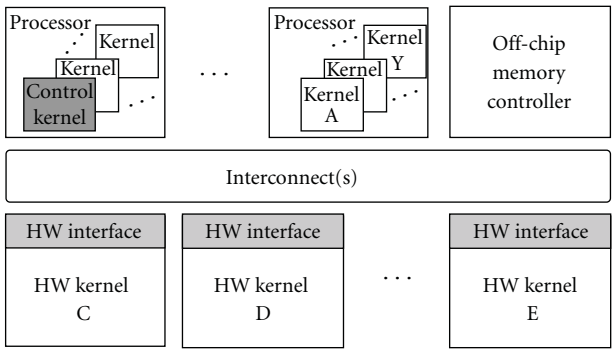

Figure 3: Physcial view.

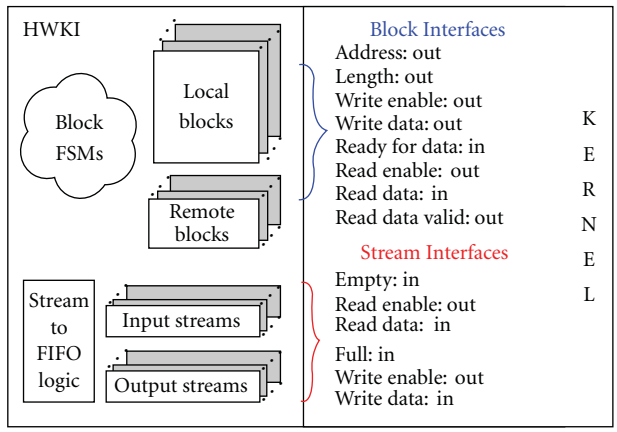

Figure 4: HWKI.

can outperform general-purpose networks. Therefore, the SSN and BSN were designed and implemented specifically to support the Redsharc API and thereby improving scalability and performance.

4.1. Stream Switch Network. The Redsharc stream switch network is a dedicated on-chip network designed to transport streams of data between communicating kernels. The SSN connects to all hardware kernels via the HWKI and the processor using DMA controllers. Software kernels communicate with the SSN through the SWKI which uses DMA descriptors to send and receive data. The SSN is composed of a full crossbar switch, configuration registers, FIFO buffers, and flow control logic.

The SSN's crossbar switch is implemented using a parametrized VHDL model which sets the number of inputs and outputs to the switch at synthesis time. Each output port of the crossbar is implemented as large multiplexers in the FPGA fabric. The inputs and outputs of the switch are buffered with FIFOs allowing each hardware kernel

to run within its own clock domain. Data is transferred from a kernel's output FIFO to another kernel's input FIFO whenever data is available in the output FIFO and room exists in the input FIFO. The flow control signals used in the SSN are based on the LocalLink specification [17]. The SSN's internal data width is set at synthesis time and defaults to 32 bits. To fulfill the Redsharc API requirement that each stream has a variable width, the FIFOs on the edges of the SSN translate the stream's data from 32 bits to the kernel's data width requirement. The SSN runs by default at 200 MHz, however, this system clock can be reduced to improve the timing score for very dense designs.

Figure 5 shows the streams and kernels in an SSN system. The input streams are shown on the left side of the figure flowing into hardware kernels, with the output streams on the right. Hardware kernels may have any number of input and output streams. Also shown are the streams connecting the processor's DMA controllers ports which connect directly to the switch, along with the switch configuration registers which are accessible from the system bus.

Hardware kernels are presented with standard FIFO interfaces as their Redsharc stream API implementation. While abstracted from hardware developers the FIFOs are directly part of the SSN.

Software kernels communicate with the SWKI library. The SWKI in turn communicates with hardware kernels by interacting with the DMA controllers. The DMA controllers can read or write data from off-chip memory into a LocalLink port. The SWKI's stream API is written to send a pointer and length to the DMA engine for the amount of data to send or receive and the location of that data. An interrupt occurs when the DMA transaction has finished. Using the DMA controllers, the processor is more efficient when there is a large amount of data to push from software to hardware or vice versa, which is often the case with streaming applications. The SWKI is also responsible for configuring the SSN's switch to connect the processor with the receiving or transmitting hardware kernel.

An advantage of the crossbar switch is that multicast stream pushes are possible by simply having several output ports reading data from the same input port. The control kernel can change the switch configuration at run-time to modify application functionality or for load balancing optimizations.

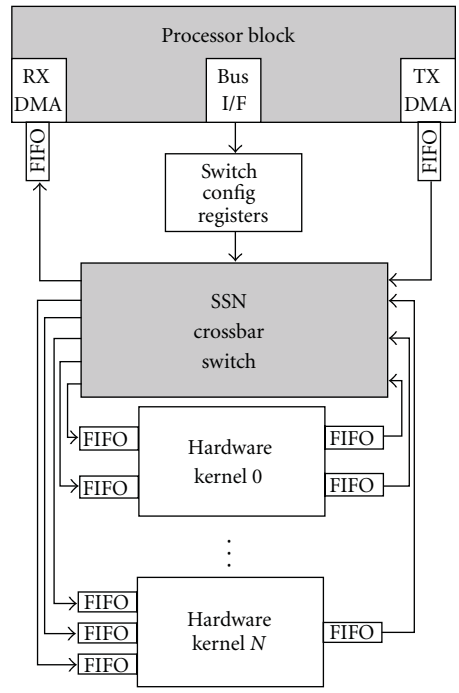

Figure 5: Stream switch network.

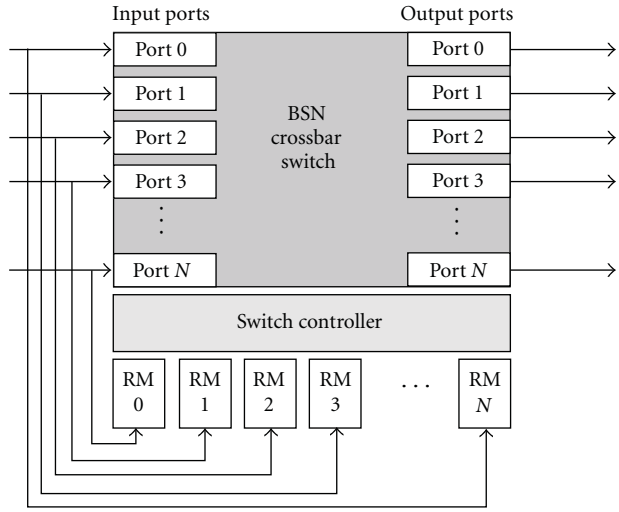

Figure 6: Block switch network's router consisting of a full crossbar switch, routing modules per each input port, and a single switch controller used to configure the inputs to the requested outputs.

4.2. Block Switch Network. The purpose of the block switch network is to implement the low level communication needed by the Redsharc block API. Redsharc blocks are created at synthesis time but allocated at run time by the control kernel. Blocks may be specified to be on-chip as part of an HWKI or a section of off-chip volatile memory. The BSN is implemented as a routable crossbar switch and permits access from any kernel to any block.

Figure 6 shows the BSN's router. The router consists of a full crossbar switch, switch controller, and routing modules. When a kernel requests data, the routing module decodes the address request and notifies the switch controller. The switch controller checks the availability of the output port and, if not in use, configures the switch. This configuration can take as few as 2 clock cycles. The configuration will stall if the desired output port is currently in use by another input until the first port finishes the transmission. The input ports are shown on the left side of the figure which are connected to the output of the hardware kernels. The output ports on the right connect to the input port of the hardware kernels. Hardware kernels can consist of any number of local, remote and off-chip memory ports, giving the kernel a higher degree of parallelism with memory access.

"Local" blocks, commonly implemented as BRAM, are accessible to a hardware kernels with low latency (≈2

clock cycles). They are instantiated as part of a hardware kernel's HWKI. A "remote" block is on-chip memory that is located in a different hardware kernel's HWKI, but accessible through the BSN. This is made possible through dual-ported BRAMs. One port is dedicated to the local hardware kernel, the second port is dedicated to the BSN. "Off-chip" blocks are allocated in volatile off-chip RAM (e.g., DDR2 SDRAM). The BSN communicates directly with the memory controller. While hardware kernels still share this connection for block communication, requests are serialized at the last possible point helping to improve overall performance.

The BSN abstracts away the different block types to provide hardware developers with a common interface via the HWKI. The HWKI block interface is an extension of the common BRAM interface with the addition of "ready for data," "valid," "request size," and "done." The added signals offer the ability to burst larger amounts of data in and out of memory. The use of the ready for data and valid signals is necessary since requests to remote and off-chip memory may take an undetermined amount of time.

The BSN also abstracts away data (element) sizes from the kernel developer. In conventional bus-based systems, the developer may need to access 32-bit data from one location and 128-bit data from a second location. As a result, the developer must be aware of the amount of data needed for each request type. In the Redsharc system, the BSN, gives the developer the exact data size needed and handles the internal transfers accordingly. With the BSN a developer can still transfer 128-bit data, but instead of actively transmitting the four 32-bit words, only a single transaction is required. The BSN still transfers all 128-bits; however, it does so internally as a burst of four 32-bit words.

Another advantage is the block location can be moved (e.g., from a local block to a off-chip block) at synthesis time based on available resources without requiring the kernel developer to redesign their code. For example, if the design requires a significant amount of BRAMs to be used elsewhere in the system, the data can be placed in off-chip memory. To the kernel developer the interface remains fixed and only the connection within the HWKI changes.

Software kernels access blocks through the SWKI. If the block is located off-chip, the SWKI routes the request through the PPC's memory port (avoiding bus arbitration). If the block is located within the BSN, the request is translated to memory mapped commands to the BSN on the PLB.

While blocks are conceptualized as either being stored in on-chip block RAMs or in off-chip RAM, many designs also need access to a set of small control or "one off" registers to interface with peripheral or kernels. We group these together as special purpose registers. For example, a design may need access to one or more of these registers in order to set some initial parameters, such as sequence length or number of transfers. Rather than dedicating an entire block, a designer may want to have the more familiar register access within the kernel. The BSN enables access to these registers by both the control kernel and other hard and soft kernels in the system. A designer can specify at synthesis the number of registers which are to be allocated for the kernel. To the hardware kernel interface, the registers are additional elements with a specific address range. However, to the kernel, the registers are directly accessible without performing block transfers.

5. Implementation

The Redsharc infrastructure spans several IP cores and incorporates multiple interfaces. Nonetheless, by utilizing VHDL generics and by careful organization, Redsharc can be synthesized for multiple FPGA families. Specifically, this is possible because of four key principles. First, it simplifies hardware and software kernel development through an abstract programming model that has been implemented in a conventional software and hardware kernel interface (SWKI and HWKI, resp.). Second, Redsharc can use both hard and soft processing cores to control kernel execution. Third, by design, Redsharc enables low-latency and high-bandwidth access to streams of data between hardware and software kernels through the stream switch network (SSN). Last, having a separate network for data transfers to random access block memory (through the block switch network, BSN) reduces resource contention.

5.1. Kernel Integration. Among its many contributions, Redsharc enables rapid kernel integration into new and existing systems. Through the use of the hardware kernel interface (HWKI), a kernel developer can exclusively focus on the design of the kernel rather than how to access streams and blocks of data. Complexities associated with bus transactions and memory controllers need not be considered by the kernel developer. Instead, the Redsharc infrastructure simplifies these accesses to mere FIFO and BRAM interfaces. The HWKI is then responsible for translating these requests into the more complex stream and block transfers. This is especially important as systems migrate from on FPGA to the next because modifying low-level kernels to access memory should be the last concern for a system designer.

5.2. System Integration. Separate from kernel development is system integration. Redsharc emphasizes the difference in order to enable designers to more efficiently construct large systems comprising of several kernels that are executing in both hardware and software. The goal is to allow the rapid assembly of such systems without significant involvement from individual kernel developers. In fact, so long as the kernel developer adheres to FIFO- and BRAM-like interfaces, the system designer is free to assemble and modify the system.

Ultimately, a Redsharc system is comprised of one or more processors, memory controllers, and kernels. The processors can be hard or soft IP, such as the PowerPC, MicroBlaze, or Nios processor. While performance and/or available resources may dictate which processor to use, the Redsharc system has been designed to be completely processor and FPGA vendor agnostic. The LocalLink interface standard used in the SSN and BSN can be replaced with a similar point to point streaming protocol with flow control. The internal LocalLink channels in the SSN and BSN would not need to be modified to accept a new streaming protocol,

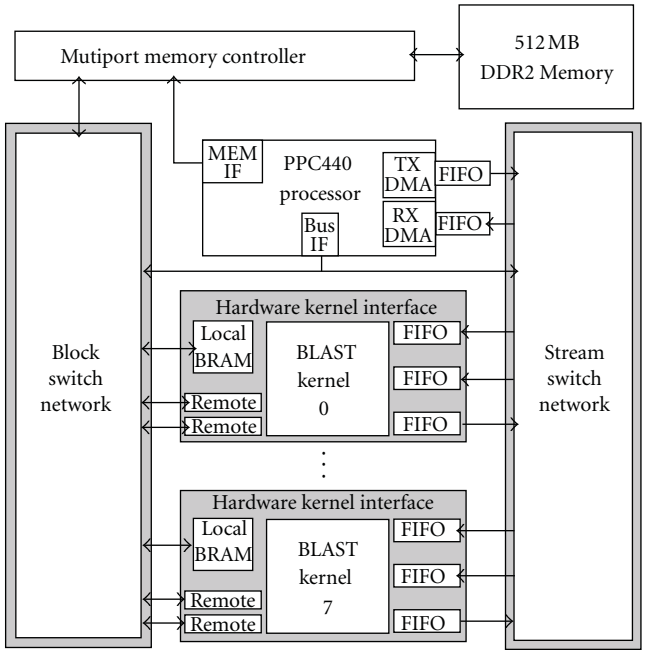

Figure 7: PowerPC BLAST system.

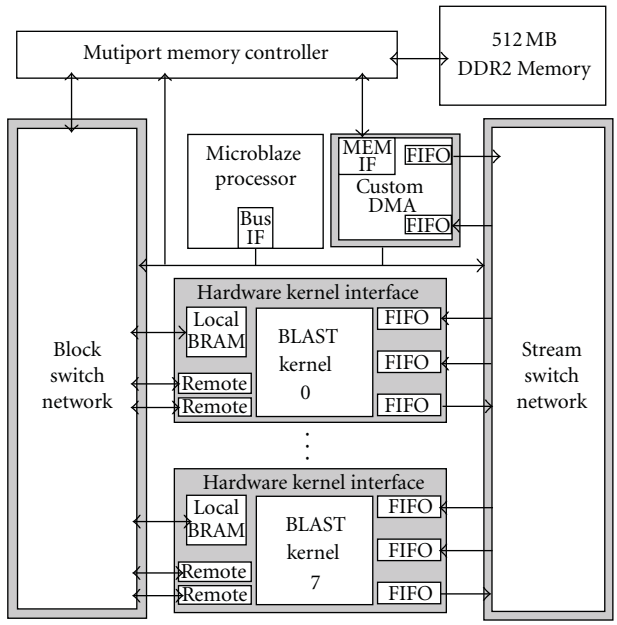

Figure 8: MicroBlaze BLAST system.

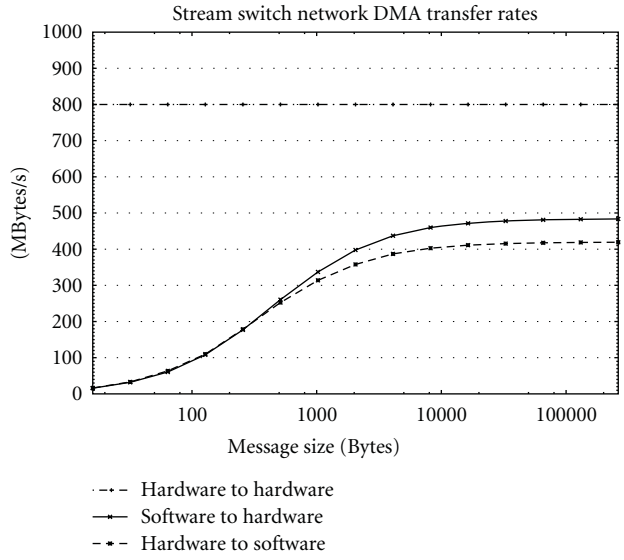

Figure 9: Stream bandwidth across hardware and software with 200 MHz SSN clock.

only the specific interfaces that connect between the SSN and BSN switches with the processors or system bus of the system.

The only requirement of the processor is that it needs to have software-addressable registers to configure the SSN and BSN networks. This is easily accomplished over a system bus interface. The BSN needs direct access to the external memory controller for high efficiency. Both Xilinx and Altera FPGAs provide such an interface for user logic. In this paper, we have implemented Redsharc using two processor systems, a PowerPC 440 hard processor and MicroBlaze soft processor. Both systems were implemented on a Xilinx ML510 development board which is based on a Virtex 5 FX 130T FPGA. Figures 7 and 8 show these two systems.

5.3. Memory Interface. While a variety of interfaces to memory exist, Redsharc is best suited to be directly connected to on-chip and off-chip memory controllers. Specifically, both the SSN and BSN have separate interfaces to the memory controller to enable independent access, reducing contention, and simplifying arbitration for the individual resource.

The SSN is connected to the processor core via two FIFOs for stream communication and a set of registers to configure the SSN. In this implementation, we have used DMA to transfer a buffer from software, in DDR2 main memory, to or from the SSN FIFOs. The PowerPC 440 system uses the DMA controllers embedded in the processor IP block. The MicroBlaze system uses a custom DMA controller since no embedded DMA controller exists in the current MicroBlaze implementation.

The BSN requires a connection to external memory to access blocks stored off-chip. In this Redsharc implementation, we use the Xilinx Multi-Port Memory Controller (MPMC) to connect the BSN directly to external memory. While the current HDL will only interface with the MPMC, adapting the BSN to use a different memory controller would only require changes to one component of the BSN.

6. Results

A Redsharc system with between-one and eight BLAST kernels has been implemented to measure the performance,

scalability, and resource utilization of the Redsharc abstract API and infrastructure. The characteristics, measured on a Virtex 5 FX 130T FPGA, are presented in this section. Xilinx ISE Design Suite version 11.4 was used for all experiments for FPGA implementation.

6.1. Network Performance. The SSN's performance is measured by the bandwidth between two kernels. While the HWKI abstracts these issues from the hardware developer, these metrics are dependent on (1) width of the stream, (2) width of the SSN, (3) clock frequency of the SSN and kernels. Between two hardware kernels, running at 200 MHz and a 32 bit data width, the SSN's bandwidth is 800 MB/s. When the receiving or transmitting kernel is software, the bandwidth is limited by the throughput of the DMA controller and the message size. Figure 9 shows the measured bandwidth for different stream lengths between hardware kernels and software kernels. The SSN performs best with large message sizes.

The BSN's performance is measured by the latency and bandwidth of a read operation. Similar to the SSN, the Redsharc API abstracts synthesis-time configurations that may affect the bandwidth of a specific system. The settings that affect BSN's bandwidth are (1) location and width of the data, (2) operating frequency of the hardware kernel and BSN (clock domain crossing adds overhead), and (3) possible contention at the remote block. Table 3 provides an overview of the BSN performance for the three types of data locality, given both the BSN and hardware kernels on the same 100 MHz clock. The BSN performs very favorable compared to the PLB which, in previous work, has a measured peek performance of 25 MB/s [18] for a 32 bit read transaction and a 200 MB/s for 1024 bit transaction.

6.2. Kernel Performance. In order to demonstrate Redsharc's scalability, a BLAST bioinformatics kernel was implemented. BLAST is a bioinformatics application that performs DNA comparisons. BLAST compares one short DNA sequence, called the query, against a long database of DNA sequences, and produces a stream of indexes where the two sequences match. The database of DNA sequences is 35 KBytes. The BLAST query is encoded into the 8 KByte local block. Researchers have implemented BLAST on FPGAs to demonstrate impressive speedups [19]; however, previous work [20, 21] has shown that bandwidth requirements for BLAST limit the scalability such that a common bus interconnect is insufficient for transferring databases to each BLAST core as well as being used for off-chip memory lookups. Specifically, each BLAST core can sustain database input rates of 3.2 GB/s. Common bus implementation, such as the processor local bus (PLB), offers sufficient bandwidth (12.8 GB/s) although there are limitations on the burst size and the number of concurrent transfers. For these reasons and due to the aforementioned previous research investigating conventional bus-based implementations, this work focuses on analyzing the scalability of BLAST cores within the Redsharc system.

The BLAST hardware kernel used in these experiments is a reimplementation of the BLAST algorithm with two input streams: one for the input database and one for the length of each sequence in the database. One output stream is used for the matching results. Three blocks (one local and two off-chip) store query information. Each BLAST kernel added to the system can run one query in parallel. Furthermore, it is possible to use the multicast capabilities of the SSN to broadcast the same database to all of the running BLAST kernels.

Table 3: BSN latency and bandwidth for each block type with 100 Mhz BSN clock.

Block type	32 b data width	1024 b data width
Local	2 cc (200 MB/s)	2 cc (6400 MB/s)
Remote	16 cc (25 MB/s)	47 cc (272 MB/s)
Off-chip	25 cc (16 MB/s)	56 cc (228 MB/s)

For comparison, a constant work size of eight queries and one database is used to test the system. The system with a single BLAST kernel must sequentially evaluate the queries, while the system with eight kernels can evaluate them in parallel. This test will show if the Redsharc system can scale with increasing hardware kernels. Table 4 shows the execution time results of the system when executing with a PowerPC processor. The performances of the BLAST application when implemented on a microblaze based system are shown in Table 5.

Both systems show no speedup in the time to load the queries into the blocks or read the result data back from the kernels. These operations are entirely sequential and offer no possible speedup. The 100 MHz Microblaze processor takes significantly longer than the 400 MHz PowerPC processor in these sequential steps. A nearly linear speedup is observed in both systems in the time spent comparing the database to the query. Note that with eight queries running in parallel, the BSN must handle the increasing load for the query information stored in off-chip memory blocks while the SSN is also reading the database from off-chip memory. This contention for the single off-chip memory resource prevents totally linear scaling.

6.3. Resource Utilization. This subsection presents the sizes of the BSN, SSN, and HWKI, the three critical hardware components that comprise a Redsharc system. Table 6 shows the resource utilization of a single BLAST kernel and its associated HWKI. The most used resources are the lookup tables. The remote block interface components of the HWKI appears to use an extraordinary amount of resources. This is due to one of the remote blocks having a data width of 448 bits, the natural data width of a single lookup of the query information stored in the off-chip memory block. The HWKI handles expanding the data from the 32 bit memory interface into the 448 bit data width, a job that would normally have to be done by the application internally.

Figure 11 and Table 8 show the number of lookup tables used in the one, two, four, and eight kernel systems used in these tests. The figure shows that even at eight kernels, the Redsharc infrastructure, the BSN and SSN, uses fewer resources than the kernels. It also shows that the SSN is

Table 4: Performance of one to eight BLAST cores running in a PowerPC 440 Redsharc system. 100 MHz SSN, BSN, and BLAST Kernel Clocks.

Cores	Load queries	Speedup	BLAST exec.	Speedup	Read results	Speedup	Total time	Total speedup
1	1643.94 μs	1x	9013.1 μs	1x	49.49 μs	1x	10706.53 μs	1x
2	1641.16 μs	1x	4512.16 μs	2x	41.74 μs	1.19x	6195.06 μs	1.73x
4	1639.78 μs	1x	2265.38 μs	3.98x	38.73 μs	1.28x	3943.90 μs	2.71x
8	1638.65 μs	1x	1134.11 μs	7.95x	36.4 μs	1.36x	2809.16 μs	3.81x

Table 5: Performance of one to eight BLAST cores running in a Microblaze Redsharc system. 100 MHz SSN, BSN, and BLAST Kernel Clocks.

Cores	Load queries	Speedup	BLAST Exec.	Speedup	Read results	Speedup	Total time	Total speedup
1	4309.3 μs	1x	6767.1 μs	1x	83.2 μs	1x	11159.6 μs	1x
2	3959.2 μs	1.1x	3512.1 μs	1.9x	77.4 μs	1.1x	7548.7 μs	1.5x
4	3783.3 μs	1.1x	1820.5 μs	3.7x	75.1 μs	1.1x	5678.9 μs	2.0x
8	3695.3 μs	1.2x	947.6 μs	7.1x	74.4 μs	1.1x	4717.3 μs	2.4x

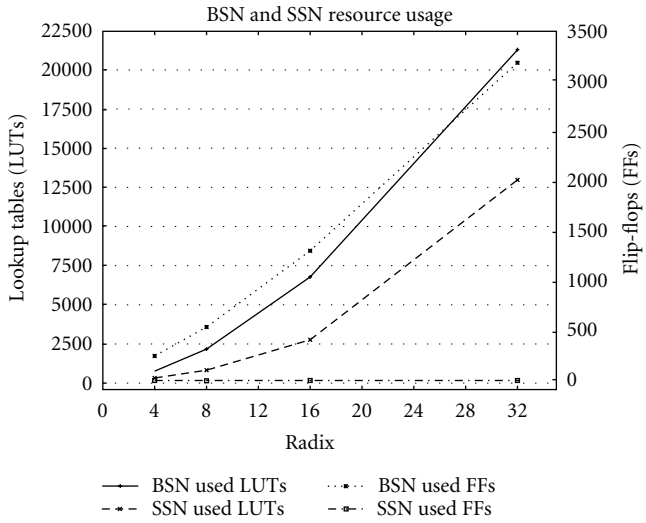

Figure 10: Block switch and stream switch network resource utilization in terms of lookup tables (LUTs) and flip-flops (FFs).

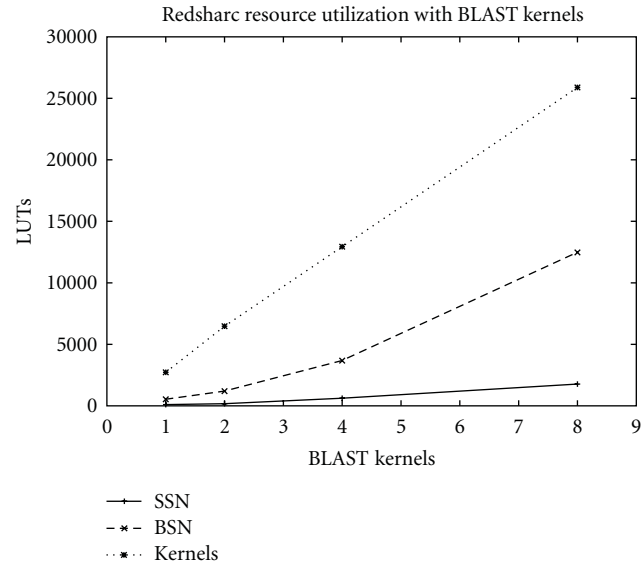

Figure 11: Redsharc system LUT utilization.

Table 6: Resource utilization of a single BLAST Kernel on a Virtex 5 FPGA.

Kernel component	LUT	FF	Carry Chain	BRAM
Local block IF	262	239	118	2
Remote block IF	1708	1077	198	0
Stream IF	1	1	0	3
BLAST application	756	471	105	0
Total	2727	1788	421	5

Table 7: BSN and SSN LUT and FF resource utilization.

Radix	BSN LUTs	BSN FFs	SSN LUTs	SSN FFs
4	456	240	140	0
8	1432	522	576	0
16	5169	1267	2272	0
32	20536	3104	11198	0

scaling lineally with the number of kernels, while the BSN has some exponential growth. Figure 10 and Table 7 illustrate the BSN and SSN's usage of lookup tables and flip-flops for any system. The term radix in this figure refers to the number of ports on each switch. For example, each BLAST kernel has three port connections to the BSN and two to the SSN. Note that the SSN is purely combinatorial and as a result has no flip-flops. The BSN number includes the routing module logic and switch controller, which increases the resource count. Overall, the resources used consume a small portion of available resources for medium-to-large-scale FPGA devices. While a bus presents a smaller resource footprint, as a trade-off the dual switches provide significant bandwidth that is necessary to satisfy the type of high-performance applications targeted by this research.

Table 8: Redsharc system LUT utilization.

Kernels	SSN	BSN	Kernel logic
1	100	534	2727
2	172	1193	6476
4	615	3672	12942
8	1770	12470	25886

The HWKI supports access to variable number of streams and blocks with variable data element sizes. As such, we present the resources required for each additional stream or block and assume 32-bit data widths for all ports. For the SSN, only an LUT is required for each input and output port to drive the Xilinx LocalLink signals and the input and output stream FIFOs. The FIFO depth is configurable by the hardware developer so the number of BRAMs is variable. For the BSN, more logic is needed to support local and remote block requests. Each local block requires 176 flip-flops and 300 LUTs whereas each remote block only requires 161 flip-flops and 163 LUTs. These represent a minimal amount of resources needed to support the high-bandwidth memory transactions while maintaining a common memory interface to the hardware kernel.

7. Conclusion

Programming MCSoPC that span hardware and software is not a trivial task. While abstract programming models have been shown to ease the programmer burden of crossing the hardware/software boundary, their abstraction layer incurs a heavy burden on performance. Redsharc solves this problem by merging an abstract programming model with on-chip networks that directly implement the programming model.

The Redsharc API is based on a streaming programming model but it incorporates random access blocks of memory. Two on-chip networks were implemented to facilitate the stream and block API calls. Our results showed that the SSN and BSN have comparable bandwidth to state-of-the-art technology and scales nearly linearly with parallel hardware kernels. Redsharc can be implemented across multiple platforms, with no dependence on a particular FPGA family or processor interface. Ergo, programmers, and system architects may develop heterogeneous systems that span the hardware/software domain, using a seamless abstract API, without giving up performance of custom interfaces.

Disclaimer

The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] A. Jerraya and W. Wolf, "Hardware/software interface codesign for embedded systems," *Computer*, vol. 38, no. 2, pp. 63–69, 2005.

[2] M. Jones, L. Scharf, J. Scott et al., "Implementing an API for distributed adaptive computing systems," in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCMM '99)*, pp. 222–230, April 1999.

[3] R. Laufer, R. R. Taylor, and H. Schmit, "PCI-pipeRench and the swordAPI: a system for stream-based reconfigurable computing," in *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCMM '99)*, pp. 200–208, April 1999.

[4] E. Lubbers 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.

[5] D. Andrews, R. Sass, E. Anderson et al., "Achieving programming model abstractions for reconfigurable computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, 2008.

[6] A. Patel, C. A. Madill, M. Saldaña, C. Comis, R. Pomès, and P. Chow, "A scalable FPGA-based multiprocessor," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 111–120, April 2006.

[7] P. Mahr, C. Lörchner, H. Ishebabi, and C. Bobda, "SoC-MPI: a flexible message passing library for multiprocessor systems-on-chips," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 187–192, 2008.

[8] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*, pp. 49–56, April 2000.

[9] D. Unnikrishnan, J. Zhao, and R. Tessier, "Application-specific customization and scalability of soft multiprocessors," in *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 123–130, April 2009.

[10] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier, "An architecture and compiler for scalable on-chip communication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 7, pp. 711–726, 2004.

[11] L. Shannon and P. Chow, "Simplifying the integration of processing elements in computing systems using a programmable controller," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, vol. 2005, pp. 63–72, 2005.

[12] P. Mattison and W. Thies, "Streaming virtual machine specification, version 1.2," Tech. Rep., January 2007.

[13] M. B. Taylor, J. Kim, J. Miller et al., "The raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[14] K. Sankaralingam, R. Nagarajan, H. Liu et al., "Exploiting ILP, TLP, and DLP with the polymorphous trips architecture," *IEEE Micro*, vol. 23, no. 6, pp. 46–10, 2003.

[15] R. Rettberg, W. Crowther, P. Carvey, and R. Tomlinson, "The monarch parallel processor hardware design," *Computer*, vol. 23, no. 4, pp. 18–28, 1990.

[16] *128-Bit Processor Local Bus Architecture Specifications*, IBM, Version 4.7 edition.

[17] Xilinx, http://www.xilinx.com/products/design resources/conn central/locallink member/sp006.pdf.

[18] A. G. Schmidt and R. Sass, "Characterizing effective memory bandwidth of designs with concurrent high-performance computing cores," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 601–604, August 2007.

[19] S. Datta, P. Beeraka, and R. Sass, "RC-BLASTn: implementation and evaluation of the BLASTn Scan function," in *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 88–95, April 2009.

[20] S. Datta and R. Sass, "Scalability studies of the BLASTn scan and ungapped extension functions," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 131–136, December 2009.

[21] A. G. Schmidt, S. Datta, A. A. Mendon, and R. Sass, "Investigation into scaling i/o bound streaming applications productively with an all-FPGA cluster," *International Journal on Parallel Computing*. In press.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 418315, 21 pages
doi:10.1155/2012/418315

Research Article

On the Feasibility and Limitations of Just-in-Time Instruction Set Extension for FPGA-Based Reconfigurable Processors

Mariusz Grad and Christian Plessl

Paderborn Center for Parallel Computing, University of Paderborn, 33098 Paderborn, Germany

Correspondence should be addressed to Mariusz Grad, mariusz.grad@uni-paderborn.de

Received 13 May 2011; Revised 19 August 2011; Accepted 16 September 2011

Academic Editor: Viktor K. Prasanna

Reconfigurable instruction set processors provide the possibility of tailor the instruction set of a CPU to a particular application. While this customization process could be performed during runtime in order to adapt the CPU to the currently executed workload, this use case has been hardly investigated. In this paper, we study the feasibility of moving the customization process to runtime and evaluate the relation of the expected speedups and the associated overheads. To this end, we present a tool flow that is tailored to the requirements of this just-in-time ASIP specialization scenario. We evaluate our methods by targeting our previously introduced Woolcano reconfigurable ASIP architecture for a set of applications from the SPEC2006, SPEC2000, MiBench, and SciMark2 benchmark suites. Our results show that just-in-time ASIP specialization is promising for embedded computing applications, where average speedups of 5x can be achieved by spending 50 minutes for custom instruction identification and hardware generation. These overheads will be compensated if the applications execute for more than 2 hours. For the scientific computing benchmarks, the achievable speedup is only 1.2x, which requires significant execution times in the order of days to amortize the overheads.

1. Introduction

Instruction set extension (ISE) is a frequently used approach for tailoring a CPU architecture to a particular application or domain [1]. The result of this customization process is an application-specific instruction set processor (ASIP) that augments a base CPU with custom instructions to increase the performance and energy efficiency.

Once designed, the ASIP's instruction set is typically fixed and turned into a hardwired silicon implementation. Alternatively, a reconfigurable ASIP architecture can implement the custom instructions in reconfigurable logic. Such reconfigurable ASIPs have been proposed in academic research [2–6], and there exist a few commercially available CPU architectures that allow for customizing the instruction set, for example, the Xilinx Virtex 4/5FX FPGAs or the Stretch S5 processor [7]. But although the adaptation of the instruction set during runtime is technically feasible and provides a promising technology to build adaptive computer systems which optimize themselves according to the needs of the actually executed workload [8], the idea of adapting the instruction set during runtime has been hardly explored.

A number of obstacles make the exploitation of just-in-time (JIT) ISE challenging: (1) there are only very few commercially available silicon implementations of reconfigurable ASIP architectures, (2) methods for automatically identifying custom instructions are algorithmically expensive and require profiling data that may not be available until runtime, and (3) synthesis and place-and-route tool flows for reconfigurable logic are known to be notoriously slow. While it is evident that even long runtimes of design tools will be amortized over time provided that an application-level speedup is achieved, it is so far an open question whether the total required execution time until a net speedup is achieved stays within practical bounds. The goal of this work is to gain insights into the question whether just-in-time processor customization is feasible and worthwhile under the assumption that we rely on commercially available FPGA devices and tools.

In our previous work we have presented initial results in each of these three areas. We have introduced the Woolcano reconfigurable instruction set architecture in Grad and Plessl [6] (obstacle 1). Woolcano is based on a Xilinx Virtex 4FX FPGA and augments the PowerPC core in the device with user-defined custom instructions (UDCI) that can be changed at runtime using partial reconfiguration. In Grad and Plessl [9] we presented a circuit library and data path generator that can generate custom instructions for this architecture. In recent work [10] we have presented new heuristics for reducing the runtime of methods for identifying and selecting custom instructions for JIT ISE (obstacle 2). Further, we have presented a first evaluation [11] of how the long runtimes of FPGA implementation tools, mentioned as (obstacle 3) above, limit the applicability of the approach.

This paper makes the following specific contributions over our previous work.

(i) In contrast to our previous work which treated the individual subproblems in JIT ISE in isolation, this paper presents them in a comprehensive way and covers the architecture, the design methods, and the corresponding tool flow along with a more detailed evaluation.

(ii) We provide an extended discussion and formal description of our candidate identification, estimation, selection, and pruning methods for a just-in-time context. Further we describe the algorithms which were developed for candidate estimation and selection in detail.

(iii) Finally, we present an extended experimental evaluation of the candidate identification and estimation methods. In particular, we focus on the candidate identification process and evaluate the suitability of three state-of-the-art ISE algorithms for our purposes by comparing their runtime, number of identified instruction candidates, and the impact of constraining the search space.

2. Related Work

This work is built on research in three areas: reconfigurable ASIP architectures, ISE algorithms, and just-in-time compilation, which have mostly been studied in separation in related works. Just-in-time ISE inherently needs a close integration of these topics; hence a main contribution of this work is the integration of these approaches into a consistent methodology and tool flow.

From the hardware perspective, this work does not target the static but reconfigurable ASIP architectures such as our Woolcano architecture [6] or comparable architectures like CHIMAERA [4], PRISC [3], or PRISM [12]. These architectures provide programmable functional units that can be dynamically reconfigured during the runtime in order to implement arbitrary custom instructions.

Research in the areas of ISE algorithms for ASIP architectures is extensive; a recent survey can be found in Galuzzi and Bertels [13]. However, the leading state-of-the-art algorithms for this purpose have an exponential algorithmic complexity which is prohibitive when targeting large applications and when the runtime of the customization process is a concern as it is in the case for JIT ISE. This work leverages our preliminary work [10] in which new heuristics were studied for effective ISEs search space pruning. It was shown that these methods can reduce the runtime of ISE algorithms by two orders of magnitude.

The goal of this work is to translate software binaries on the fly into optimized binaries that use application-specific custom instructions. Binary translation is used, for example, to translate between different instruction sets in an efficient way and has been used, for example, in Digital's FX!32 product for translating X86 code to the Alpha ISA [14]. Binary translation has also been used for cases where the source and target ISAs are identical with the objective to create a binary with a higher degree of optimization [15, 16].

This work is conceptually similar to these approaches as it also does not translate between different instruction sets, but optimizes binaries to use specific user-defined instructions in a reconfigurable ASIP. This kind of binary translation has hardly been studied so far. One comparable research effort is the WARP project [17]. The WARP processor is a custom system on chip comprising a simple reconfigurable array, an ARM7 processor core, and additional cores for application profiling and place and route. This work differs from WARP in several ways. The main difference is that we target a reconfigurable ASIP with programmable processing units in the CPU's datapath, while WARP uses a bus-attached FPGA coprocessor that is more loosely coupled with the CPU. Hence, this work allows to offload operations at the instruction level where WARP needs to offload whole loops to the accelerators in order to cope with longer communication delays. Further, WARP operates at the machine-code level and reconstructs the program's higher-level structure with decompilation, while this work relies on higher-level information that is present in the virtual machine. Finally, WARP assumes a custom system on chip, while this work targets commercially available standard FPGAs.

Beck and Carro [18] present work on binary translation of Java programs for a custom reconfigurable ASIP architecture with coarse-grained reconfigurable datapath units. They show that for a set of small benchmarks an average speedup of 4.6x and power reduction of 10.9x can be achieved. The identification and synthesis of new instructions occur at runtime; however, the paper does not specify what methods are used for instruction identification and what overheads arise from instruction synthesis.

3. General ASIP Specialization Tool Flow and Just-in-Time Runtime System

Figure 1 illustrates the difference between a conventional static ASIP specialization process (ASIP-SP) and a runtime system with a just-in-time ASIP-SP support. The ASIP-SP is responsible for (a) generating *bistreams* for configuring the underlying reconfigurable ASIP hardware architecture with instruction extensions and (b) for modifying the source code

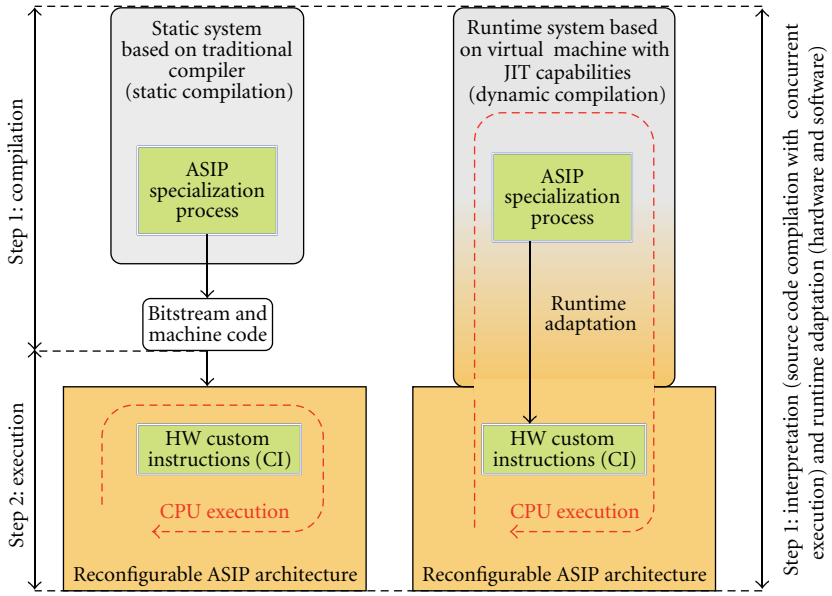

Figure 1: Overview of ASIP specialization process for conventional static and runtime systems.

to actually utilize the newly created instructions. So far, ASIP specialization has been applied almost exclusively in static systems, where steps (a) and (b) occur off line before the application is executed.

This work studies the feasibility of moving the ASIP specialization to runtime by integrating it into a virtual machine with just-in-time compilation capabilities. In such a system the ASIP specialization is performed concurrently with the execution of the application. As soon as (a) and (b) are available, the runtime adaptation phase occurs where ASIP architecture is reconfigured and the application binary is modified such that the newly available custom instructions are utilized. The main advantages of executing ASIP specialization as part of the runtime system are the following.

(i) The system can optimize its operation by reconfiguring the instruction set and by changing the code at runtime, which is fundamentally more powerful than static ASIP specialization.

(ii) The system can collect execution time, profiling, and machine level information in order to identify the code sections that are actually performance limiting at runtime; these sections are ideal candidates to be accelerated with custom instructions.

(iii) The virtual machine has the capability to execute various dynamic optimizations like hotspot detection, alias analysis, or branch prediction to further optimize the performance.

4. Our Tool Flow Implementation

For the purpose of evaluating the potential of just-in-time ASIP specialization, we have developed a prototypical tool flow that is presented in Figure 2. Our tool flow executes ASIP specialization as part of a runtime system as introduced in the previous section. However, since Xilinx's proprietary FPGA design tools can be executed only on X86 CPUs, our current version of the tool flow runs the ASIP-SP on a host computer and not on the Woolcano ASIP architecture itself; see Section 10 for details considering the experimental setup.

The details of our implemented tool flow and the Woolcano hardware architecture are presented in Figure 2. The process comprises three main phases: *Candidate Search*, *Netlist Generation*, and *Instruction Implementation*.

During the first phase, Candidate Search, suitable candidates for custom instructions are identified in the application's bitcode with the help of ISE algorithms which search the data flow graphs for suitable instruction patterns.

The ISE algorithms are computationally intensive with runtimes ranging from seconds to days, which is a major concern for the just-in-time ASIP specialization. To avoid such scenarios, the candidate identification process is preceded by *basic block pruning* heuristics which prune the search space for candidate identification algorithms to the basic blocks from which the best performance improvements can be expected. It was shown by Grad and Plessl [10] that the runtime of the ISE algorithms can be reduced by *two orders of magnitude* by sacrificing 1/4 of the speedup. The ISE algorithm identifies a set of custom instruction candidates. Afterwards the *selection* process using the performance estimation data singles out only the best one.

The *estimation* data are computed by the PivPav tool [9], and they represent the performance difference for every candidate when executed either in software or in hardware. This is possible since PivPav has a database with a wide collection of the presynthesized hardware IP cores together with more than 90 different metrics; see Grad and Plessl [9] for details. The next two phases in the process cover the generation of hardware from a software candidate and are also implemented with the help of the PivPav tool.

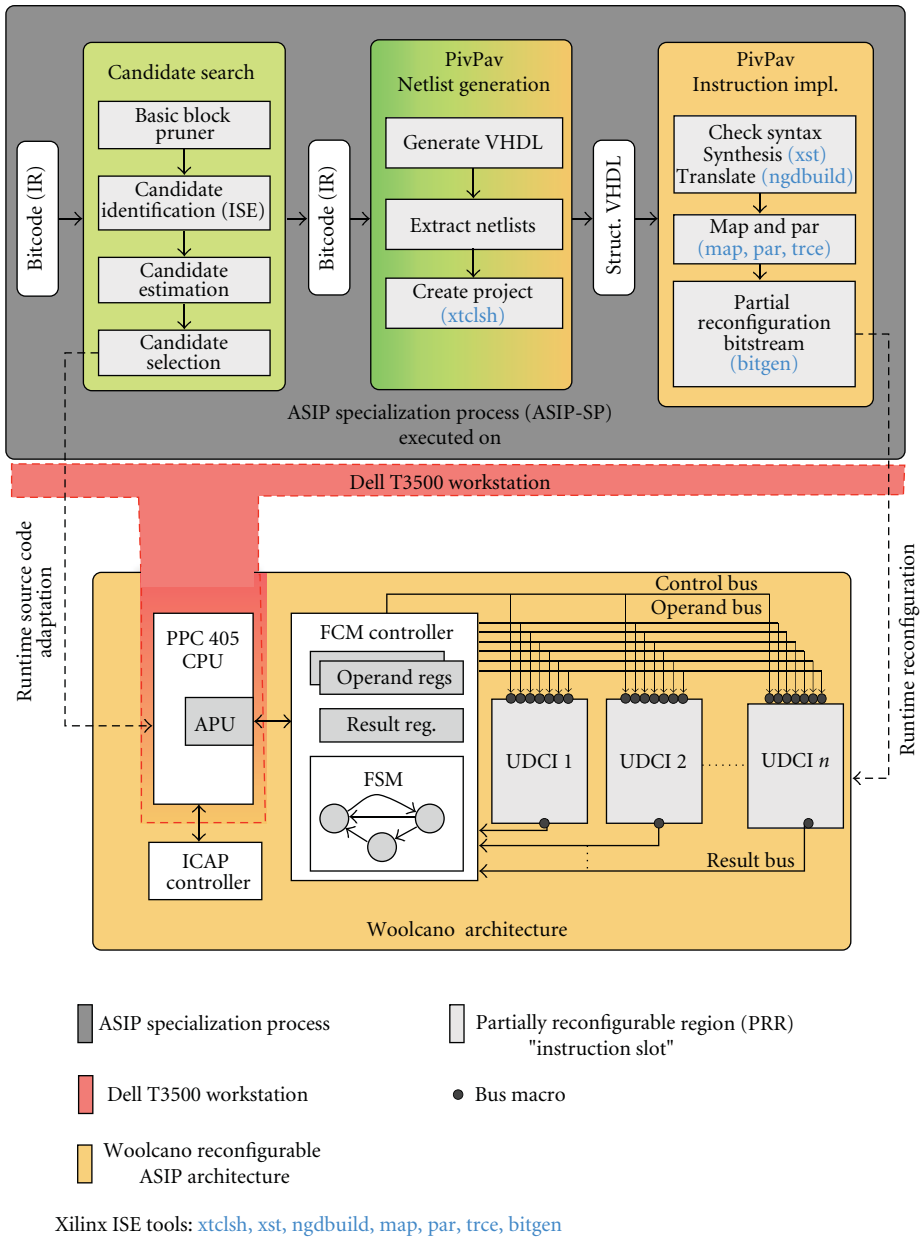

Figure 2: Overview of the developed tool flow and the targeted Woolcano hardware architecture. During experimental evaluation, instead of a PPC405 CPU, the ASIP specialization process was executed on a Dell T3500 workstation; see Section 10 for details.

The second phase, *Netlist Generation*, generates a VHDL code from the candidate's bitcode and prepares an FPGA CAD project for synthesizing the candidate. The *Generate VHDL* task is performed with PivPav's datapath generator. This generator iterates over the candidate's datapath and translates every instruction into a matching hardware IP core, wires these cores, and generates structural VHDL code for the custom instruction. Next, PivPav *extracts the netlist* for the IP cores from its circuit database. This is performed for every IP core instantiated during the VHDL generation and is used to speed up the *synthesis* and the *translation* processes during the FPGA CAD tool flow; that is, PivPav is used as a netlist cache. Finally, an FPGA CAD project for Xilinx ISE is created, the parameters of the FPGA are set up, and the VHDL and the netlist files are added.

In the third phase, *Instruction Implementation*, the previously prepared project is processed with the Xilinx FPGA CAD tool flow. This results in an FPGA configuration bitstream for the given custom instruction candidate. This bitstream can be loaded to the Woolcano architecture using partial reconfiguration. These steps are also handled by PivPav.

5. Woolcano Hardware Architecture

The bottom part of Figure 2 shows the Woolcano dynamically reconfigurable ASIP architecture. The main components

of the architecture are the PowerPC core, the internal configuration access port (ICAP) controller, the fabric co-processor module (FCM) controller, and the partial reconfiguration regions for implementing UDCI which we denote also as *instruction slots*. The FCM controller implements the interface between the CPU core and the UDCI. It forwards the inputs to the instruction slots via the operand bus and, after the custom instruction has finished computing, it transfers the output back to the CPU via the result bus. The control bus is used for sending control information, for example, activation or abort signals, to the UDCI.

Bus macros are placed at the interface between the instruction slot and the (a) operand, (b) control, and (c) result busses for enabling *dynamic partial reconfiguration* of the instruction slots. The instruction slots can be reconfigured via ICAP or the external configuration port of the FPGA.

The FCM controller was implemented as a *finite-state machine* (FSM) and is responsible for connection to the *auxiliary processor unit* (APU) interface of the *PowerPC 405* core. Its main function is to implement the APU protocol for transferring data and for dispatching instructions. During the UDCI execution the CPU is blocked and it is waiting for the UDCI results. The architectural constraints of the APU allow only for two input and one output operands to the UDCI. This restriction limits the amount of data a UDCI instruction can operate on, which in turn limits the achievable speedup. To circumvent this limitation, the FCM core implements internal operand registers for supplying the UDCI with additional operands.

The number of instruction slots as well as their input and output operands are compile-time configurable architecture parameters denoted as $C_{\max}$, $\text{in}_{\max}$, $\text{out}_{\max}$, respectively. Since all inputs and outputs to the instruction slots must be fed through Xilinx bus macros, the size and geometric placement options of the bus macros limit the number of input operands and results.

6. Candidate Identification

The candidate identification process identifies subgraphs in the *intermediate representation* (IR) code, which are suitable for fusing into a new UDCI which can be implemented for the Woolcano architecture. Suitable candidates are rich in *instruction-level parallelism* (ILP) while satisfying the architectural constraints of the target architecture.

6.1. Formal Problem Definition. Formally, we can define the candidate identification process as follows. Given a data flow graph (DFG) $G = (V, E)$, the architectural constraints $\text{in}_{\max}$, $\text{out}_{\max}$ and a set of infeasible instructions F, find all candidates (subgraphs) $C = (V', E') \subseteq G$ which satisfy the following conditions:

$$C_{\text{in}} \leq \text{in}_{\max}, \tag{1}$$

$$C_{\text{out}} \leq \text{out}_{\max}, \tag{2}$$

$$V' \cap F = \emptyset, \tag{3}$$

$$\forall t \in C : \text{convex}(t). \tag{4}$$

Here,

(i) the DFG is a *direct acyclic graph* (DAG) $G(V, E)$, with a set of *nodes* or *vertices* V that represent *IR operations* (instructions), *constants* and *values*, and an edge set E represented as binary relation on V which represents the data dependencies. The edge set E consists of ordered pairs of vertices where an edge $e_{ij} = (v_i, v_j)$ exists in E if the result of operation or the value defined by v_i is read by v_j.

(ii) $C = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

(iii) C_{in} is the set of *input nodes* of C, where a node $v_i \notin C$ with $(v_i, v_j) \in E$ for some node $v_j \in C$ is called an input node.

(iv) C_{out} is the set of *output nodes* of C, where a node $v_0 \in C$ with $(v_0, v_k) \in E$ for some node $v_k \notin C$ is called an output node.

(v) $\text{in}_{\max}$, $\text{out}_{\max}$ are constants that specify the input/output operand constraints for UDCIs which apply to the instruction slot implementation of the Woolcano architecture.

(vi) $F \subseteq V$ is a subset of illegal graph nodes (IR instructions) which are not allowed to be included in an UDCI. This set includes for instance all memory access instructions like *loads* and *stores* (since a UDCI cannot access memory) and other instructions which are not suitable for a hardware implementation, for example, for performance reasons.

(vii) Convex means that there does not exist a path between two vertices in C which contains a node that does not belong to C. In other words candidate C is convex if $v_i \in C$, $i = 0, \ldots, k$ for all paths $\langle v_0, \ldots, v_k \rangle$ with $v_0 \in C$ and $v_k \in C$ where path is defined as follows. A *path* from a vertex u to a vertex v in a graph C is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $v_0 = u$, $v_k = v$ and $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, k$. Convexity ensures that C can be executed atomically in the hardware. This property is required to make sure that an instruction can be executed as an atomic operation without exchanging intermediate results with the CPU.

Translating these formal definitions to practice means that the identification of UDCI candidates occurs at the level of *basic blocks* (BBs). BBs are suitable units for this purpose since they have a DAG structure as required by the ISE algorithms. In addition, it is feasible to enforce the convexity condition (cf. condition (4)) on them. When selecting IR code for a UDCI implementation, illegal instructions, such as control flow or memory operations, must be excluded (cf. condition (3)). Finally, the number of inputs and outputs of the candidate has to respect the architectural constraints (cf. conditions (1) and (2)). These architectural constraints are variable and are defined by the FCM controller and by the interface to the partially reconfigurable slots into which the UDCIs are loaded.

Table 1: Candidate identification ISE algorithms comparison.

Algorithm	# of inputs	# of outputs	Worst-case complexity	Overlapping candidates
MaxMiso	Invariant (∞)	Invariant (1)	$O(n)$	No
SingleCut	Variant	Variant	$O(\text{exp})$	Yes
Union	Variant	Variant	$O(\text{exp})$	No

6.2. Supported Instruction Set Identification Algorithms. For the identification process, we implemented three state-of-the-art ISE algorithms that are considered as the most suitable for runtime processor specialization: the *SingleCut* (SC) [19], the *Union* (UN) [20], and the *MaxMiso* (MM) [21] algorithm. The most relevant properties of these algorithms are presented in Table 1.

All identification algorithms do not try to find candidates in the entire application at once. Instead, they focus on the individual basic blocks of the application. Hence, in order to prune the search space, the ISE algorithm is executed selectively only for the most promising BBs (cf. Section 9). Each ISE algorithm analyzes the basic block to identify all feasible candidates. Further, some algorithms allow to constrain the number of inputs and outputs of candidates to match the architectural constraints of the targeted ASIP architecture. Finally, all algorithms fulfill the condition (3).

The advantage of the MM algorithm is its linear complexity $O(n)$. However, it finds only candidates which have a single output and does not allow to constrain number of inputs. Therefore, some of the generated candidates need to be discarded at additional costs later in the selection phase because they validate conditions and thus they cannot be implemented. In contrast, the SC and UN algorithms already allow for restricting the desired number of inputs and outputs for candidates during the identification phase. Finally, the SC algorithm may produce overlapping candidates. Hence, when using this algorithm an additional phase is needed in the selection process to eliminate the overlapping candidates.

7. Candidate Estimation

Since the number of UDCIs that can be implemented concurrently in the Woolcano architecture is limited, only the best candidates are selected for an implementation. The corresponding selection process which is described in the following section is based on the estimated performance of every candidate when executing in software on the CPU or in hardware as a UDCI. Based on these performance estimates the subsequent selection process can decide whether it is affordable and beneficial to implement a candidate as a hardware UDCI instruction.

7.1. Software Estimation. The Woolcano architecture consists of a PowerPC 405 CPU hard core which is used for software execution. In contrast to modern general-purpose CPUs, the PowerPC CPU has a relatively simple design. It has a scalar, in-order, five-stage pipeline, and most instructions have a latency of a single cycle. This simple design makes the task of software estimation relatively easy since the execution of the instructions in the candidate is sequential. Hence, the presented bellow estimation method is not a novel idea, and it is based on research published in Gong et al. [22].

We estimate the performance of the execution with the expression shown in (5) which corresponds to the sum of latencies of all instructions found in the candidate multiplied by the CPU clock period. This estimation technique has an algorithmic complexity of $O(n)$ where n is the number of PowerPC instructions found in a candidate

$$T_{\text{sw}} = T_{\text{cpu}} \cdot L_{\text{sum}}\ [\text{ns}],$$
$$L_{\text{sum}} = \sum_{i=0}^{n} L_i\ [\text{ns}]. \tag{5}$$

Here,

(i) T_{cpu} is the clock period of used PowerPC CPU.

(ii) L_{sum} is the sum of latencies of all instructions found in a candidate, where n is the number of instructions and L_i is the latency of the ith instruction found in a candidate (the instruction latencies have been determined from the PowerPC manual [23]).

The use of the T_{cpu} in (5) ensures that the differences in clock periods between the PowerPC CPU and the UDCI hardware implementation are taken into account.

The presented method yields correct results only when the candidate's IR code is translated one to one into the matching PowerPC instructions. In the case of a mismatch between these two, the estimation results are inaccurate. The mismatch can happen due to a few reasons, that is, folding a few IR instructions into a single target instruction, or because of differences in the register sets. The PowerPC architecture has a fixed amount of registers (32 general-purpose registers) whereas the IR code uses an unlimited number of *virtual registers*. For larger code, which requires more registers than available, the backend of the compiler produces additional instructions which will move data from registers into temporary variables kept on the stack. These additional instructions are not covered in the software estimation process. For such cases it has been shown that the estimation inaccuracy can be as high as 29% [24].

7.2. Hardware Estimation. Since each instruction candidate can be translated to a wide variety of functionally equivalent datapaths, the task of hardware estimation is much more complicated than the task of software estimation. In the following, we choose to illustrate approach used in this work by means of an actual example. Listing 1 shows an excerpt from a raytracing algorithm. The corresponding IR code of one of the identified candidates is presented in Listing 2. This candidate is constructed from adders and multipliers and corresponds to the scalar product which is shown in Listing 1 on the 5th line.

When translated to hardware, the DFG **structure** of the candidate is preserved; that is, instead of complex high-level synthesis [25], a more restricted and thus simpler datapath

```
(1)  bool hitSphere (const ray &r, const sphere& s..)
(2)  {
(3)  ...
(4)  vecteur dist = s.pos - r.start;
(5)  float B = rdx * dx + rdy * dy + rdz * dz;
(6)  float D = B * B - dist * dist + s.size * s.size;
(7)  if (D < 0.0f) return false;
(8)  ...
(9)  }
```

Listing 1: Part of the raytracing source code.

```
(1) ...
(2) %0 = mul float  %rdx, %dx
(3) %1 = mul float  %rdy, %dy
(4) %2 = add float  %0,   %1
(5) %3 = mul float  %rdz, %dz
(6) %4 = add float  %2,   %3
(7) ...
```

Listing 2: IR of candidate found in 5th line of previous listing.

synthesis (DPS) process [1] is required which does not generate complex finite state machines in order to schedule and synchronize computational tasks. The first step in the hardware estimation for DPS involves translating each IR instruction (node) into a corresponding *hardware operator*, where operators may exist as purely *combinational* operators or as *sequential* operators. Sequential operators produce a valid result after one or—in the case of pipelined operators—several clock cycles. Also, functionally equivalent operators can have a large variety of different properties such as hardware area, speed, latency, and power consumption. Therefore, the hardware estimation tasks have to deal with three different types of datapath structures: combinational, sequential, and hybrid datapaths, where a mix of sequential and combinational operators exists. Examples for such datapaths for the discussed candidate are shown in Figure 3. The hardware estimation process used in this work for estimating the delay of a UDCI supports all of these scenarios and is formally defined as

$$C_{\text{hw}} = T_{\text{udci}} \cdot R_D \cdot P_{\max}\,[\text{ns}], \tag{6}$$

$$T_{\text{udci}} = \max\{L\}\,[\text{ns}], \tag{7}$$

$$P_{\max} = \max\{P\}\,[\#]. \tag{8}$$

Here,

(i) T_{udci} corresponds to the minimal allowable *clock period*, which is visualized as the tallest green box in Figure 3. For combinational datapaths, scenario (a), it is equivalent to the latency of critical path, whereas for the sequential datapaths, scenario (b), to the maximal latency of all operators (`add[1]` in this example). For hybrid datapaths, scenario (c), it corresponds to the highest latency of all sequential operators and combinational paths; in this case, to the sum of combinational `mult[0]` and sequential `add[1]` operator latencies.

(ii) R_D is an experimentally determined routing delay parameter which is used to decrease the T_{udci}. The routing delays are equivalent to the communication latencies between connected operators caused by the routing switch boxes and wires. The precise value of R_D is unknown until the physical placement of the operators is performed in the FPGA; however, experiments showed that R_D often corresponds to about half of all circuit latencies.

(iii) $P_{\max}$ is the maximum number of pipeline stages. It can be interpreted as the maximum number of all green rectangles covering a given DFG. For scenarios (a), (b), (c), and (d), $P_{\max}$ equals to 1, 3, 2, and 24, respectively.

(iv) L is a set of latencies generated in Algorithm 1, and its maximum for all operators defines the minimal allowable clock period; see (7). The graphical interpretation of L, as presented in Figure 3, corresponds to a list of the height of all green boxes. Thus, combinational datapaths, scenario (a), have the highest latencies, whereas the smallest latency can be found in highly pipelined sequential datapaths, scenario (d). The latency of each operator is obtained from the PivPav circuit library with the `Latency()` function found in the algorithm in the 4th line.

(v) P is a set of all pipeline stages generated by Algorithm 1. P is used in (8) to select the maximum number of stages in a given datapath. The number of pipeline stages for operator is retrieved with `Pipeline()` function and it is presented in square brackets in the operator name; thus, the `mult [10]`- reflects the 10 stages multiplier.

Algorithm 1 is used to compute the values of the L and P sets, which are associated with the height and the number of green boxes, respectively. In the first line of the algorithm, initialization statements are found. In the second line,

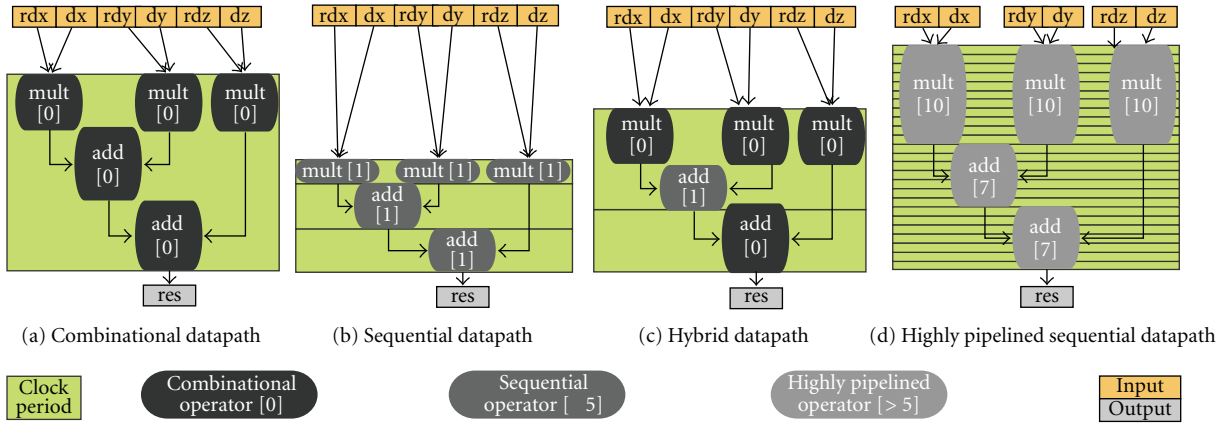

Figure 3: Different types of a DFG presented for a UDCI candidate shown in Listing 2.

(1) $L \leftarrow P \leftarrow \phi$
(2) **for** p in *critical_paths* **do**
(3) **for** $n \in$ nodes(p) **do**
(4) $L_p \leftarrow L_p +$ Latency(n)
(5) $P_p \leftarrow P_p +$ Pipeline(n)
(6) **if** Pipeline(n) $\neq$ 0 **then**
(7) $L \leftarrow L \cup L_p$ and $L_p \leftarrow 0$
(8) **end if**
(9) **end for**
(10) **if** Pipeline(n) = 0**then**
(11) $P_p \leftarrow P_p + 1$
(12) **end if**
(13) $L \leftarrow L \cup L_p$ and $L_p \leftarrow 0$
(14) $P \leftarrow P \cup P_p$ and $P_p \leftarrow 0$
(15) **end for**

Algorithm 1: Hardware estimation.

the algorithm iterates and generates results for every *critical path*, which indicates that a path leads from the *input* to the *output* node. In next three lines, for each node in a given path, the latency and the number of pipeline stages are accumulated in L_p and P_p temporal variables, respectively. If the given node is sequential, the new green box is created and the current latency value L_p is moved to the L set (lines (6)–(8)). Lines (10)–(15) are executed when the algorithm reaches the last node in the given critical path. Thus, lines (10)–(12) add an additional pipeline stage if the last node is a combinational one, whereas lines (13)-(14) move the values of temporal variables to the resulting sets. Since the candidate's template does not overlap, each node of the candidate is visited only once, and therefore this estimation technique is of $O(n)$ complexity.

In order to illustrate the estimation process in detail, we show how C_{hw} is estimated for (b) sequential datapath and (d) highly pipelined sequential datapath representation of the candidate. The metrics of the used operators obtained from the PivPav circuit library are described below and are presented in Table 2. The upper and lower parts of the table show the metrics of the operators used in scenarios (b) and (d), respectively. The estimation results found in Table 3 indicate that the sequential datapath is able to produce the first result almost twice as fast as the highly pipelined datapath (1.82x). However, processing many data (d) is able to fill the pipeline and work with 24 data at once, generating the results, accordingly to the formula $T_{\text{udci}} * R_D$, every 9.5 ns, whereas (b) generates a result only every 41.78 ns.

The T_{udci} and $P_{\max}$ factors in the equations depend on the characteristics of every used hardware operator; see *Latency()* and *Pipeline()* functions in Algorithm 1. Thus, the key to accurate hardware estimation is the quality of the characterization database that provides performance, latency, and area metrics for each operator. For some selected operators, for example, for floating point operators obtained from *IP Core Libraries*, data sheets that characterize each operator with the required performance metrics are available, for example, [26]. For most other operators—in particular those created on demand by HDL synthesis tools, such as integer, logic, or bit manipulation operators—no data sheets exist. Moreover, the characterization data in data sheets are not exhaustive and do not cover all FPGA families, models, and speed grades, which is problematic, since even within one device family the performance of operators can vary significantly. For example, the data sheet for Xilinx Coregen quotes the maximum speed of a floating-point multiplier as 192 MHz or 423 MHz, respectively, for two FPGA models from the Xilinx Virtex-6 family (Table 23 versus Table 25) [26]. This huge range makes it impractical to estimate accurate performance metrics for devices that are not even tabulated in the data sheets.

To meet the characterization data requirements for the hardware operators, we have developed the open-source PivPav tool [9], which we leverage also in this work to obtain accurate characterization metrics. PivPav generates a library of hardware operators, where each operator is specifically optimized for the targeted FPGA family. For each operator, the performance and many other metrics are automatically extracted from the implementation. These metrics are made available to the estimation process and other processes in the ASIP specialization tool flow via the PivPav API, which

Table 2: Excerpt from metrics requested by the candidate estimation process from PivPav circuit library for the XC4VFX100-FF1152-10 FPGA device.

HW Oper.	P_p #	L_p ns	Max FRQ after PAR [MHz]	FF #	LUT #	Slice #	BUF #	DSP #
mul	1	24.81	40.3	66	76	46	103	0
add	1	32.15	31.1	66	377	250	103	5
mul	10	7.31	136.8	66	134	150	103	4
add	7	7.19	139.0	66	556	326	103	4

Table 3: Results of hardware estimation process for DFGs presented in Figures 3(b) and 3(d) implemented with two different sets of operators found in Table 2.

		Sequential datapath (Figure 3(b))	Highly pipelined sequential datapath (Figure 3(d))
L	[ns]	24.81 and 32.15	7.31 and 7.19
T_{udci}	[ns]	32.15	7.31
P	#	3 and 2	24 and 17
$P_{\max}$	#	3	24
R_D	#	1.30	1.30
C_{hw}	[ns]	125.39	228.07

is illustrated in Figure 2. Since all the data are generated beforehand with the benchmarking facilities of the PivPav, there is no need to run the FPGA CAD tool flow in the estimation process.

It is worth noticing that the presented estimation method is not in itself a novel idea. There are many timing analysis approaches which perform equivalent steps to the one presented above [27]. Therefore, this subsection does not contribute to the hardware timing analysis field. The novelty in this estimation approach relies on the precise characterization data that were generated with PivPav tool. These data together with presented methods allow to precisely estimate the hardware performances for UDCI instructions.

8. Candidate Selection

Once the set of candidates has been determined and estimation data are available, the selection process makes the final decision about which candidates should be implemented in the hardware.

First, all the candidates that violate at least one of the constraints presented below are rejected:

$$C_{|\text{in}|} \leq \text{in}_{\max}, \tag{9}$$

$$C_{|\text{out}|} \leq \text{out}_{\max}, \tag{10}$$

$$\frac{C_{\text{sw}}}{C_{\text{hw}}} \leq \text{threshold}. \tag{11}$$

Here,

(i) $C_{|\text{in}|}$ and $C_{|\text{out}|}$ are equivalent to the constraints described in conditions (1) and (2), respectively. They correspond to architectural constraints for the number of input and output operands to the UDCI. They are applied to the ISE algorithms that are not able to perform this step themselves, such as the MM algorithm.

(ii) C_{sw} and C_{hw} correspond to the software and hardware estimations, respectively. If threshold = 1.0, then there are no performance gains; when threshold >1.0 there is a performance gain since it takes more cycles to execute the candidate in software than in hardware. Finally, if threshold <1.0, the hardware implementation has a lower performance than the software.

After applying these conditions, the search space of the selection process is significantly reduced, since candidates that are either infeasible or would provide only low speedups are discarded. As a result, the runtime of the subsequent steps in the tool flow is considerably lower.

8.1. Candidate Selection. In general, selecting the optimal set of UDCI instructions under different architectural constraints is a demanding task. Related work has studied different selection approaches, such as greedy heuristics, simulated annealing, ILP, or evolutionary algorithms; see, for example, Meeuws et al. [28] or Pozzi et al. [19]. For the purpose of the ASIP specialization, we use the greedy candidate selection algorithm that is presented in Algorithm 2 and which has a computational complexity of $O(|C_{\text{input}}|)$. When using the SC algorithm which may produce overlapping candidates for ISE identification our algorithm rejects any candidates that overlap with any candidate that has been selected so far. The aim of this process is to select up to $C_{\max}$ candidates from the set of C_{input} candidates generated by the identification process that offers the greatest advantage in terms of some metric M; in this case the *application performance*:

$$C_{\text{res}} = \max\left(\forall C_i \in C_{\text{input}} : \sum M(C_i)\right). \tag{12}$$

Here,

(i) C_{res} is the resulting set of best candidates, $|C_{\text{res}}|$ is a size of this set, and $C_{\max}$ is the architectural constraint representing the number of supported UDCI instructions.

(ii) M is a metric function defined as

$$M(C_i) = \frac{C_{\text{sw}}(C_i)}{C_{\text{hw}}(C_i)}. \tag{13}$$

while $|\mathcal{C}_{\text{res}}| \leq \mathcal{C}_{\max}$ **do**
  $c_i \leftarrow \max\{M(\mathcal{C}_i) \mid \mathcal{C}_i \in \mathcal{C}_{\text{input}}\}$
  if c_i does not overlap with $\mathcal{C}_{\text{res}}$ **then**
    $\mathcal{C}_{\text{res}} \leftarrow \mathcal{C}_{\text{res}} \cup c_i$
  end if
  $\mathcal{C}_{\text{input}} \leftarrow \mathcal{C}_{\text{input}} \cap c_i$.
end while

Algorithm 2: Best candidate selection.

8.2. Selection Metrics. The metric function is used as a policy in the greedy selection process and is responsible for selecting only the best candidates. While in (13), the *application performance* policy is used, and nothing prevents basing the decision preference on a different metric. It is worth mentioning that the PivPav tool could be used to provide a wealth of other metrics since the circuit library stores more than 90 different properties about every hardware operator. These properties could be used for instance to develop resource usage or power consumption policies. Consequently, they can be used to estimate the size of the final *bitstream* and partial reconfiguration time, the runtimes of netlist generation and instruction implementation, or many other metrics. Finally, all these policies could be merged together into a sophisticated ASIP specialization framework which would

(i) maximize the performances,

(ii) minimize the power consumption, and

(iii) constrain the resource area to the sizes of UDCI slot.

Such a combined metric can be defined as an *integer linear programming* model. While this method would allow a more precise selection of candidates based on more parameters, its algorithmic complexity is higher than $O(|C_{|\text{input}|}|)$, resulting in runtimes that are much longer, often by orders of magnitude. Since it is important to keep the runtimes of the ASIP-SP as low as possible, the tradeoff between the gains and the costs of the metric function is an interesting research topic in itself.

9. Pruning the Search Space

Pruning is the first process executed in the ASIP-SP outlined in Figure 2. Pruning uses a set of algorithms which act as *filters* to shrink the search space for the subsequent processes by rejecting or by passing certain BBs.

This decision is based on the data obtained from *program analysis* which provides information about loops, the sizes of BBs, and the contained instruction types. In addition, the ASIP-SP makes it possible to discard dead code by running the filters only for the code which was executed at least once.

The objective of the pruning process is described by the following term:

$$\max\left(\frac{\text{metric function}}{\text{runtime of candidate identification}}\right). \quad (14)$$

The pruning aims to maximize the ratio between a *metric function* to the time spent in the candidate identification process. The *metric function* is defined in (13) and is equivalent to the *application performance* gain. The denominator of the equation takes into the account only the *runtime of the candidate identification* since in comparison to this runtime the runtime of the *candidate estimation* and *selection* processes is insignificant.

For this work, we are using this *@50pS3L* filter heuristic, which has been shown in our previous work [10] to provide the best results for just-in-time systems. This filter has three sequential pruning stages which can be decoded from its name. The first filter stage (indicated by @) filters out *dead code*; that is, it discards all BBs that have not been executed at least once. This information is available during the ASIP-SP runtime without access to profiling data. The second filter stage (indicated by *50p*) selects BBs based on their size. Here, only BBs that have a size of at least 50% of the size of the largest BB in the application are selected. Preferring large over small BBs simplifies the task of identifying large candidates which are likely to provide more speedups. Finally, the last filter stage (indicated by *3L*) selects only BBs which are part of a loop and selects the 3 largest of these BBs. The rationale of this filter is that the BBs contained in a loop have a higher chance of being executed, and promoting candidates which are more frequently executed is one of the methods to increase the overall application speedup.

10. Experimental Setup

While this work targets the reconfigurable ASIPs, like our Woolcano architecture presented in Figure 2, due to practical limitations it is currently not feasible to execute the complete ASIP-SP on an embedded reconfigurable ASIP architecture. The specialization process heavily uses the LLVM compiler framework and the Xilinx ISE tools which require high-performance CPUs and desktop operating systems. These resources are not available in currently existing ASIP architectures. Hence, we used Linux and a Dell T3500 workstation (dual core Intel Xeon W3503 2.40 GHz, 4 M L3, 4.8 GT/s, 12 GB DDR3/1333 MHz RAM) as a host computer in place of the PowerPC 405 CPU of the Woolcano architecture to execute the ASIP-SP; see Figure 2.

The lack of the possibility to run the complete tool flow on the ASIP has a number of consequences for the experimental evaluation. Instead of running the ASIP-SP as a single process, we are forced to spilt this process into two steps. In the first step, the host computer is used to generate the partial bitstreams by executing the tasks corresponding to the upper half of Figure 2. In the second step, we switch to the Woolcano architecture where we use the generated bitstreams to reconfigure the UDCI slots and to measure the performance improvements.

It is also worth noticing that this two-step process has an impact on several reported measurements. First, all performance measurements reported in Table 5 and 6, in columns *Max ASIP-SP speedups* and *ASIP ratio*, are performed for Woolcano's PowerPC405 CPU and not for the host CPU.

Table 4: Characteristics of scientific and embedded applications. AVG-S represents the averages for scientific applications and AVG-E for the embedded applications. Ratio = AVG-S/AVG-E.

App	Sources		Compilation to IR				IR in BBs			Code coverage			Kernel size		
	files	loc	real	fun	blk	ins	max	avg	udci	live	dead	const	size	ins	freq
	#	#	[s]	#	#	#	#	#	[%]	[%]	[%]	[%]	[%]	#	[%]
164.gzip	20	8605	3.89	33	1006	6925	59	6.88	29.68	38.86	44.66	16.48	5.78	400	90.34
179.art	1	1270	1.06	21	376	2164	43	5.76	21.53	42.05	28.47	29.48	9.84	213	92.45
183.equake	1	1513	1.71	15	257	2670	132	10.39	23.0	75.39	8.91	15.69	15.32	409	92.9
188.ammp	31	13483	10.10	98	4244	26647	382	6.28	25.74	19.22	70.89	9.89	3.38	901	95.81
429.mcf	25	2685	0.97	18	284	1917	77	6.75	13.09	75.9	13.09	11.01	25.77	494	98.46
433.milc	89	15042	10.88	87	1538	14260	363	9.27	32.59	61.67	34.72	3.61	10.83	1545	93.99
444.namd	32	5315	22.77	84	5147	47534	291	9.24	37.82	31.71	62.81	5.48	7.33	3486	93.64
458.sjeng	23	13847	8.49	86	3373	20531	69	6.09	21.1	48.49	49.44	2.07	44.6	9157	100.0
470.lbm	6	1155	1.36	16	104	1988	405	19.12	57.55	55.23	24.9	19.87	32.75	651	97.57
473.astar	19	5829	3.68	45	757	6010	70	7.94	27.45	78.79	5.31	15.91	6.39	384	91.3
AVG_S	24.70	6874	6.49	50	1709	13065	189.1	8.77	28.95	52.73	34.32	12.95	16.20	1764	94.65
adpcm	6	448	0.29	6	43	233	39	7.21	33.48	60.66	29.18	10.16	41.97	128	91.79
fft	3	187	0.26	10	47	297	41	6.53	42.09	58.88	30.26	10.86	44.08	134	95.98
sor	3	74	0.13	4	19	99	22	7.06	34.34	46.51	50.39	3.1	24.81	32	97.52
whetstone	1	442	0.25	12	44	285	32	6.58	34.04	32.75	36.27	30.99	10.21	29	93.27
AVG_E	3.25	288	0.23	8	38.3	228.5	33.5	6.85	35.99	49.70	36.52	13.78	30.27	80.75	94.64
RATIO	7.60	23.89	28.22	6.29	44.67	57.18	5.64	1.28	0.80	1.06	0.94	0.94	0.54	21.85	1.00

Table 5: Specialization process executed for whole applications when targeting the Woolcano architecture without capacity constraints. The performance of the custom instructions has been determined with the PivPav tool. ISE algorithms: MM: MaxMiso, SC: SingleCut, UN: Union. SC and UN search is constrained to 4 inputs and 1 input.

App	Executionruntimes			ISE algorithm runtime			Candidates found			Max ASIP-SP Speedup		
	VM	Nat	Ratio	MM	SC	UN	MM	SC	UN	MM	SC	UN
	[s]	[s]	x	[ms]	[ms]	[ms]	#	#	#	x	x	x
164.gzip	23.71	18.47	1.28	40.6	549.0	11170.0	1621	44177	43682	1.172	1.213	1.213
179.art	69.92	74.70	0.94	12.3	55.1	3350.0	371	3534	3513	1.526	21.414	21.414
183.equake	7.97	6.79	1.17	13.5	457.9	4351.0	672	9690	9690	2.147	25.972	25.972
188.ammp	23.18	17.24	1.34	145.7	15840	—	7547	122441	—	3.449	20.826	—
429.mcf	23.94	24.06	1.00	11.1	68.7	200.5	571	3571	3562	1.112	1.112	1.112
433.milc	20.95	16.43	1.28	78.1	5065	—	3573	59450	—	1.301	21.546	—
444.namd	39.94	34.31	1.16	227.5	35854	—	11490	125970	—	1.609	24.846	—
458.sjeng	180.41	155.66	1.16	123.7	6244.1	235195.7	5540	83173	83035	1.118	1.137	1.137
470.lbm	5.68	5.36	1.06	8.6	2777.1	—	490	18216	—	2.554	**44.622**	—
473.astar	66.00	67.68	0.98	33.4	914.8	303796653	1408	37025	32368	1.159	1.19	1.19
AVG_S	46.17	42.07	1.14	69.45	6783	30405092	3328	50724	17585	1.71	16.39	5.20
adpcm	29.22	28.35	1.03	1.7	15.0	3869.4	83	819	819	1.243	1.309	1.293
fft	18.47	18.49	1.00	1.6	9.7	33.1	87	553	552	3.1	14.413	14.413
sor	15.83	15.85	1.00	0.7	4.3	14.6	35	384	375	14.418	14.422	14.418
whetstone	28.66	28.50	1.01	1.6	9.5	64.0	69	435	435	**18.012**	18.012	**18.012**
AVG_E	23.04	22.80	1.01	1.40	9.62	995.27	68.50	547.75	545.25	9.19	12.04	12.03
RATIO	2.00	1.85	1.13	49.61	705.05	30549.59	48.59	92.61	32.25	0.19	1.36	0.43

Table 6: The runtime overheads for the ASIP-SP.

App	Candidate Search: *@50pS3L*					ASIP	Runtime overheads				Break-even
	real	pruner	blk	ins	can	ratio	const	map	par	sum	break-even time
	[ms]	effic	#	#	#	x	[m:s]	[m:s]	[m:s]	[m:s]	[d:h:m:s]
164.gzip	1.44	71.79	2	100	19	1.00	56:22	13:02	18:28	87:52	206:22:15:50
179.art	1.05	23.37	3	79	9	1.01	26:42	8:58	13:20	49:00	1:12:18:13
183.equake	2.25	8.33	2	244	11	1.00	32:38	7:56	16:12	56:46	259:02:28:33
188.ammp	3.27	52.29	1	382	92	1.41	272:58	102:12	142:49	517:59	0:14:56:39
429.mcf	1.05	28.2	1	77	5	1.00	14: 50	4:06	7:48	26:44	213:20:05:55
433.milc	6.6	26.71	2	673	9	1.00	26:42	6:44	15:08	48:34	568:06:08:05
444.namd	7.68	57.43	3	776	129	1.03	382:45	117:24	178:04	678:13	6:16:00:48
458.sjeng	1.8	184.11	3	121	8	1.00	23:44	6:56	12:58	43:38	2403:01:35:57
470.lbm	10.62	2.43	3	961	179	2.53	531:07	181:51	308:24	1021:22	1:03:29:48
473.astar	2.25	38.2	3	184	33	1.00	97:54	29:46	46:59	174:39	5149:02:19:14
AVG_S	3.80	49.29	2.30	358	49	**1.20**	146:34	47:53	76:01	270:28	881:00:33:54
adpcm	0.84	5.59	2	61	8	1.08	23:44	6:00	10:34	40:18	0:04:34:10
fft	0.78	3.78	2	75	14	2.40	41:32	11:44	20:56	74:12	0:01:53:07
sor	0.24	2.21	1	22	2	1.00	5:56	4:48	10:12	20:56	0:00:24:19
whetstone	0.54	7.7	2	49	9	15.43	26:42	11:34	25:52	64:08	0:01:08:04
AVG_E	0.60	4.82	1.75	52	8	**4.98**	24:28	8:31	16:53	**49:53**	**0:01: 59:55**
RATIO	6.33	10.23	1.31	6.95	5.99	0.24	5.99	5.62	4.50	5.42	10580

Further, in order to compute the *break-even* time reported in Table 6 we used the *runtime overheads* values from the same table which were measured on the host computer. Therefore, this value is computed as if Woolcano's PowerPC CPU had the processing power of the host machine. Finally, while the ASIP specialization tool flow is capable of performing UDCI reconfiguration during runtime, in practice, we had to switch from the first to the second step manually.

The hardware limitations of Woolcano, in particular the number of UDCI slots, in practice do not allow us to measure the performance improvements on a real system for all applications. To this end, for these applications we estimate the speedups with the help of the techniques presented in Section 7.

11. Applications for Experimental Evaluation

Table 4 shows the characteristics of used applications divided into two groups. The upper part of the table shows data for applications obtained from the SPEC2006 and SPEC2000 benchmark suites which represent scientific computing domain whereas the lower part represents applications from the embedded computing domain obtained from the SciMark2 and MiBench.

While the used benchmark suites count 98 different applications altogether, we could not run our evaluation on all of them due to cross-compilation errors. Hence, from the set of available applications, we selected the ones which are the most representative and allow us to get comprehensive insights into the JIT ASIP specialization methodology. While we have used slightly different application sets in our previous publications which evaluated specific parts of the tool flow, we have chosen an adapted common set of applications for this work in order to get a consistent end-to-end evaluation of the whole tool flow.

11.1. Source Code Characterization and Compilation. The second and third columns of Table 4 contain the number of *source files* and *lines of code* and tell that scientific applications have on average 23.89x more code than the embedded applications. This difference influences the *compilation time* shown in the fourth column which for scientific applications is 28.22x longer on average, but still the average compilation time is only 6.5 s. The next three columns express the characteristics of the *bitcode* reflecting the total number of *functions*, *basic blocks*, and *intermediate instructions*, respectively. For the scientific applications the ratio between *ins* (13065) and the *LOC* (6874) is 1.9. This means that an average single high-level code line is expressed with almost two IR instructions and less than one (0.8) for embedded applications. Since scientific applications have 24 times more LOCs than embedded applications, this results in a 57x difference in the IR instructions.

The ISE algorithms operate on the BBs, and thus the *IR in BBs* column indicates the characteristics of these BBs in more detail. The *max* column indicates the BB with the highest number of IR instructions and *avg* is the average number of IR instructions in all BBs. These values in combination with the data presented in Figures 4, 5, and 6 allow to understand the *runtime* of the ISE and the *number of candidates*.

For embedded applications the largest BBs cover on average more than 14.7% of the application whereas the largest basic block for scientific applications covers only 1.4% of the total application. The difference between average-size BBs for embedded and scientific applications is 1.28x and results in a small average number of IR instructions of less than

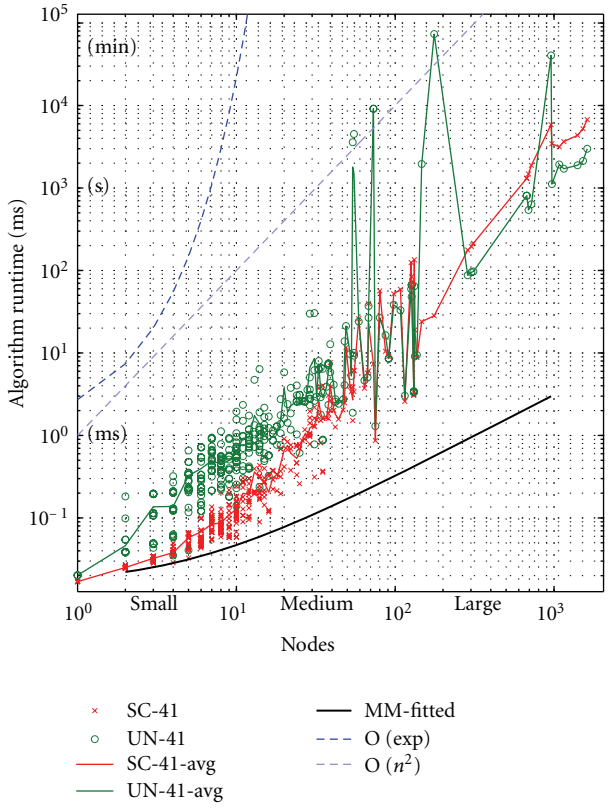

Figure 4: Runtimes of the ISE identification algorithm for different basic block sizes (nodes). The SC: SingleCut, UN: Union, MM: MaxMiso algorithms. The label "41" means that the SC and UN search has been constrained to 4 inputs and 1 output.

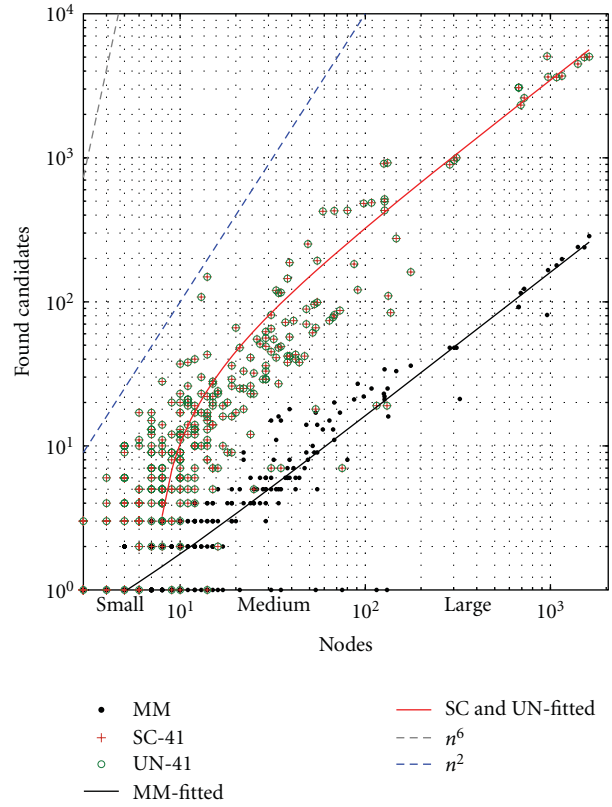

Figure 5: Number of found candidates by ISE algorithms for a large spectrum of BB nodes, where SC and UN are constrained to 4 inputs and 1 output.

10 for both cases. The small size of BBs in our applications needs to be attributed to the actual benchmark code, the compiler, and the properties of the intermediate representation. Our experiments have shown that the size of the BBs does not change significantly for different compiler optimizations, transformations, or with the size of application (LOC).

11.2. Feasible UDCI Instructions. The *udci* column lists the percentage of all IR instructions which are feasible for a hardware implementation. Feasible instructions include the arithmetic, logic, and cast instructions for all data types and make up to 1/3 of all instructions of the application. Considering the small average size of BBs this means that the size of an average-found candidate is only between 2 and 3 IR instructions. This fact emphasizes the need for a proper BB and candidate selection and stresses even more the importance of the proper pruning algorithms in order to avoid spending time with analyzing candidates that will likely not result in any speedup.

11.3. Code Coverage. The *Code Coverage* columns show the percentages of the size of *live*, *dead*, and *constant* code. These values were determined by executing each application for different input data sets and by recording the execution frequency of each BB. For the SPEC benchmark suite applications, the standard *test*, *train*, and *ref* data sets were used, whereas for the embedded ones, due to the unavailability of standard data sets, each application was tested with at least three different custom-prepared input data sets. After execution, the change in execution frequency per block between the different runs was compared. If the frequency was equal to 0, the code was marked as *dead*. If the frequency was different from 0 but did not change for different inputs, the code was marked as *constant*, and if the frequency has changed, the block was marked as *live*. This frequency information was used to compute the *break-even points* in the following section. In addition, the *live* frequency information indicates that roughly only 50% of the application has a dynamic behavior in which the ISE algorithms are interested in searching for candidates.

11.4. Kernel Size. The last three columns contain data on the size of the kernel of the application. These data are derived from the frequency data. The kernel of an application is defined as the code that is responsible for more than 90% of the execution time. The size of the kernel is measured as the total number of IR instructions contained in the basic blocks which represent the kernel. For scientific applications, 16.20% of the code affects 94.65% of the total application execution time, and it corresponds to more than 1.7 k IR

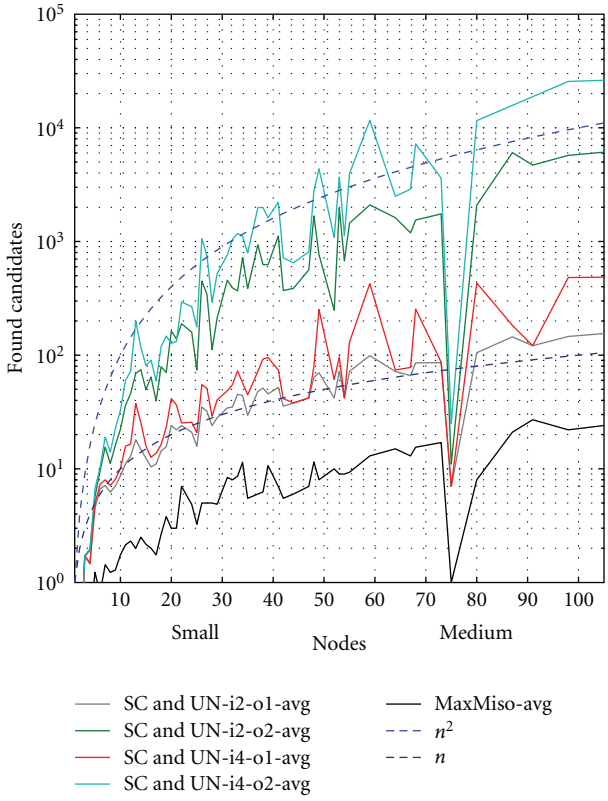

Figure 6: Number of found candidates by ISE algorithms for a medium spectrum of BB nodes, where SC and UN have variable constraints.

instructions. For embedded applications, the average relative kernel size is 30.27% and is expressed only with 80 IR instructions. These numbers indicate that it is relatively easier to increase the performances of the embedded applications than the scientific ones, since they require 22x smaller UDCI instructions.

11.5. Execution Runtimes. The *VM* column in Table 5 represents the application runtime when executed on the LLVM virtual machine. The runtime of the application depends heavily on the input data which, in the case of the scientific applications, were obtained from the *train* datasets of the SPEC benchmark suite. Due to the unavailability of standard data sets for the embedded applications, custom-made data sets were used. For both application classes, the input data allowed to exercise the most computationally intensive parts of the application for a few or several tens of seconds. The *Nat* column shows the *real* runtime of the application when statically compiled, that is, without the overhead caused by the runtime translation. The *Ratio* column shows the proportion of *Nat* and *VM* and represents the overhead involved with the interpretation during the runtime. For the small embedded applications, the overhead of the *VM* is insignificant (1%). For the large scientific applications, the average overhead caused by the VM equals on average 14%. However, it is important to notice that for some applications like *179.art* or *473.astar*, the *VM* was significantly faster than the statically compiled code by 6% and 2%, respectively. This means that the *VM* optimized the code in a way which allowed to overcome the overhead involved in the optimization as well as the dynamic just-in-time compilation.

12. Experimental Evaluation of the Candidate Identification

In this section, we evaluate and compare the ISE algorithms which have been used for candidate identification as presented in Figure 2. Our evaluation covers the runtime characteristics of the algorithms as well as the number of identified candidates for different architectural constraints, the maximum gain in application performance, and the runtime of our benchmark algorithms when statically compiled to native code and when executing in a virtual machine. The discussion is based on data presented in Table 5 which was obtained for the benchmark applications introduced in the previous section.

12.1. ISE Algorithm Runtimes and Comparison. The average ISE algorithm runtimes are presented in the 5th to 7th columns of Table 5. As stated previously, the MM algorithm has a linear complexity and therefore is the fastest, resulting in a 0.22 s runtime for `444.namd`, which is the largest application. Due to its larger algorithmic complexity, the runtime of the UN algorithm should generally exceed the runtime of SC but this is not the case for applications which include specific types of BBs. For example, it took 3837 ms to process such a specific BB consisting of 55 nodes in the `adpcm` application, which is 99.15% of the overall runtime of the UN algorithm. On contrast, the same BB was analyzed by the SC algorithm in a mere 4.7 ms.

Since both SC and UN have exponential complexity, their runtimes are a few orders of magnitude higher than for MM. In average, MM is 96.94x faster than the SC algorithm.

The identification times for BBs of similar sizes also vary significantly for the same algorithm since the number of candidates that need to be actually considered in a BB depends not only on the total size of the BB but also on the structure of the represented DFG, the number and location of infeasible instructions in the DFG, the architectural constraints, and other factors. For example, it took the SC algorithm 1707 ms to analyze a BB of `433.milc` with 102 instructions, while the analysis of a slightly larger BB in `470.lbm` with 120 instructions took only 76 ms, which is a 22.5x difference in runtime. This example illustrates that it is impossible to accurately estimate the runtime of the exponential ISE algorithms SC and UN in advance when basing the estimation solely on the size of the BB.

It is worth pointing out that for keeping the search space and thus the algorithm runtimes manageable, we had to apply rather tight architectural constraints for the custom instructions (4 inputs, 1 output) in our comparison of ISE algorithm in Table 5. When loosening these constraints, the execution times for the SC and UN algorithms rapidly grow from seconds to many hours.

The overall runtime characterization of the instruction identification algorithms is summarized in Figure 4 which plots the runtime of the different algorithms for varying BB sizes. Each data point represents an average which was computed by running the ISE algorithm 1000 times on each BB. The multitude of data points for a fixed BB size illustrates that the runtime of the same algorithm can vary over several orders of magnitude for BBs which have an equal size but differ in their structure as pointed out above. This effect is particularly strong for larger BBs where many variants of DAG structures exist.

For visualizing the overall behavior of the algorithms we have also added the average runtime for the SC and the UN algorithms for each BB size. It can be observed that the variability for the UN algorithm is larger than for the SC algorithm. Another interesting observation is that although the SC and UN algorithms have a worst-case algorithmic complexity of $O(\exp)$, on average their runtime is only polynomial $O(n^2)$, which can be seen by comparing the blue dotted lines with the red and green lines, respectively.

We are able to fit the runtime of the MM algorithm with a linear polynomial model which is represented with a black line which has an almost ideal characteristic (goodness of the fit: $R^2 = 0.9995$). This means that the runtime of the MM algorithm always depends linearly on the BB size $O(n)$. Unfortunately, the behavior of the other algorithms is not sufficiently regular to perform a meaningful curve fitting with similar quality.

In general, we can say that the MM is the fastest algorithm and outperforms the SC and UN algorithms easily in terms of runtime for small, medium, and large basic blocks. For small BBs of up to 10 instructions, the runtime difference is in the range of up to an order of magnitude; for medium inputs (10^2 instructions), up to two orders of magnitude and for the largest BBs (10^3 and more instructions), a difference of more than three orders of magnitude can be observed. While the runtime of the MM stays on the millisecond time scale even for the largest inputs, the SC and UN algorithms work on a scale of seconds or minutes.

It is important to note that runtime of the exponential algorithms tend to literally explode when these algorithms are constrained less tightly than 4 input 1 output (41), in particular when allowing a larger number of outputs. For instance, when applying an 8-input 2-output (82) constraint, common runtimes are in the order of 10^8 ms, which is three orders of magnitude higher than for the 41 constraint.

In terms of runtime, SC is approximately one order of magnitude faster than UN. However, the runtime for both algorithms grows similarly for increasing BB sizes with the exception of significant outliers for the UN algorithm, for peaks with a runtime difference of three orders of magnitude which can be observed for large BBs. A similar behavior was also found for architectural constraints other than 41.

The results presented here have been obtained using a special *benchmarking mode* of our tool flow where the instruction candidates are identified but not copied to a separate data structure for further processing. Additionally, the time needed to reject overlapping candidates for SC algorithm as well as the time needed for the MM algorithm to validate condition (1) presented in Section 6 was not included. As a result, the runtimes of the candidate identification algorithms will be slightly longer in practice when they are applied as part of the complete tool flow.

12.2. Candidates Found by the ISE Algorithms. The number of candidates found by the ISE identification algorithms is presented in columns 8, 7, and 9 of Table 5. In addition, an overview of all identified candidates as a function of the BB size is shown in Figure 5, while Figure 6 presents a close up of the same data for medium-sized BBs. As illustrated by the *red* line in Figure 5, the SC and the UN algorithms generate an equal number of candidates, given that the same architectural constraints are used and that any overlapping UDCI candidates generated by the SC algorithm are removed.

In general, the total number of subgraphs that can be created from an arbitrary graph G is exponential $\exp(n)$ in the number of nodes of G. For ASIP specialization scenarios, that is, when architectural constraints are applied cf. conditions (1), (2), and (4) [29] has shown that the number of subgraphs is bounded by $n^{(C_{\text{in}}+C_{\text{out}}+1)}$. Thus, for the 4-input/1-output constraints applied in this study, the search space is equal to n^6, which is represented by the gray dotted line found in the upper-left corner of Figure 5. When applying the final constraint condition (3), the search space is significantly reduced from n^6 by at least a power of 4 to n^2, which is presented with the blue dotted line above all results.

The *black* line represents a linear fitting for the MM algorithm, whereas the *red* line shows a second-order polynomial curve fitting for the SC and UN algorithms. The goodness of these fits represented with R^2 parameter is equal to 0.9663 for MM and to 0.9564 for the SC and UN algorithms. Therefore, it is safe to assume that the number of candidates for the 4-input/1-output constraint is limited by a second-order polynomial.

Figure 6 shows that the longer runtimes of the SC and the UN algorithm also result in the identification of more candidates. The difference for small and medium BBs is up to one order of magnitude and increases for even larger BBs.

The data points (number of candidates) were obtained by running the ISE algorithms on the applications presented in Section 11. Thus, each data point is associated with a single BB, and closely located data points tell that there are many BBs of similar sizes.

It can be observed in Figure 6 that there are less data points with large BBs than medium or small BBs. This is a consequence of the distribution of basic block sizes; that is, most BBs found in these applications have sizes of up to 100 instructions. For such BBs, the number of feasible candidates reaches more than 10 when using the MM algorithm and more than 100 when using the SC and UN algorithms. Given that the average application has at least a few dozens (38 for embedded) or hundreds (1709 for scientific) of BBs this results in thousands of feasible candidates that are suitable for hardware implementation. In our experiments, we considered 3328 MM (50724 SC) candidates for the scientific and 68.5 MM (547.75 SC) candidates for the embedded applications. These high numbers are more than

enough since an average ISA consists of around 80 core instructions for X86 platform and around 100 for PowerPC. If one assumes 10% modification to the ISA, it results in a task of selecting less than 10 UDCI instructions from a set of thousands of feasible candidates.

The number of found candidates depends strongly on the architectural constraints that are applied. Tighter constraints, that is, allowing a smaller number of inputs and outputs, lead to a smaller number of candidates for the SC and UN algorithm. This behavior can be seen in Figure 6 where the average number of candidates is plotted as a function of the BB size for various constraints. There are two groups of constraints: the 21,41 and 22,42, between which a rising gap of one order of magnitude is established. Applying the MM algorithm to BBs with a size of 100 instructions leads to more than 10 feasible candidates whereas applying SC or UN leads to more than 100 candidates for the first set of constraints and even two orders of magnitude more candidates (10^4) for the second set of constraints. This validates the second-order polynomial characteristic n^2 of the number of candidates for the SC or UN algorithms.

The similar behavior of the lines representing the average number of identified candidates is caused by the *less or equal* ($\leq$) relationships found in conditions (1) and (2). That is, the less constrained algorithms (like 41) include all candidates of more constrained ones like 21. The area between the *red* and *gray* line corresponds exactly to the number of additional candidates found in less constrained algorithms. Also, the graphs illustrate that the number of candidates depends much stronger on the number of allowable outputs than on the number of allowable inputs.

For BBs with sizes of approximately 75 instructions, we see an interesting decay from which all ISE algorithms suffer. This decay is found only in a concrete benchmark and is the result of a high concentration of illegal instructions in basic blocks of those sizes, for which only a few feasible candidates were found.

Finally, it can be seen that the MM algorithm has a linear characteristic $a \cdot n$ where $a \leq 1$. The SC and UN algorithms also show a linear characteristic with $a \geq 1$ for the case of the 21 and 41 constraints, whereas for the 22 and 42 constraints, the characteristic changes by a power (n^2).

12.3. Achievable Performance Gain. The *Max ASIP-SP Speedup* columns presented in Table 5 describe the upper limit of performance improvement that can be achieved with the Woolcano reconfigurable ASIP architecture and the presented ASIP-SP. These values show the hypothetical best-case potential in which all candidates found by three different ISE algorithms are implemented as custom instructions. In reality, the overheads caused by implementing all possible instructions and the limited hardware resources of the reconfigurable ASIP require pruning of the set of candidates that are evaluated and implemented to a tractable subset. Therefore, the speedup quoted in these columns should be treated only as an upper boundary on the achievable performance.

The ISE algorithms have a lot to offer, reaching a speedup of up to 44.62x for SC algorithm and 18.01x for MM and UN algorithms. The average speedups achieved with MM, SC, and UN are 1.71, 16.39, and 5.20, respectively for the scientific applications and 9.19, 12.04, and 12.03 for the embedded applications. For all applications, the average speedups achieved with MM, SC, and UN are 3.85, 15.15, and 10.02, with the value of median 1.57, 16.22, and 7.85, respectively. These results clearly indicate that the SC algorithm is superior for static systems where identification runtimes are not a major concern.

For a JIT specialization process, one needs to balance the achievable speedup with the identification time. Comparing the ratios of average speedup to identification runtime for embedded applications results in the following ratios: 6.56 (MM), 1.25 (SC), and 0.01 (UN). These figures suggest that the MM algorithm is the most suitable for such systems. In addition, the considerable difference of 0.19 between average speedups for different application sets suggests that the MM algorithm could find better candidates in the smaller applications with more pronounced kernels and that these applications will benefit most from JIT-based systems.

It is important to remember that these results were obtained for the first time for the FPGA-based Woolcano architecture and not as presented in related work for a fixed CMOS ASIP architecture. This distinction is significant since the same hardware custom instructions will achieve significantly higher speedups when implemented in CMOS technology, often by more than one order of magnitude. But at the same time, a fixed architecture will sacrifice the flexibility and runtime customization capabilities of the Woolcano architecture.

13. Runtime Overhead of the ASIP Specialization Process

As elaborated in the previous section, the reconfigurable ASIP architecture is considerably faster than the underlying CPU alone for both benchmark domains. Thus, the overheads of just-in-time software compilation, optimization, and custom instruction generation can be amortized provided that the application will be executed long enough. In this section, we analyze the achievable performance gains by ASIP specialization, presented in Figure 2, and the runtime costs of the three different phases of that process. These figures are used to compute for how long the application needs to be executed until the hardware generation overheads are amortized, that is, when a net speedup is achieved.

13.1. Candidate Search. As described in Section 4, the *Candidate Search* phase is responsible for finding and selecting only the best custom instruction candidates from the software. As this task is frequently very time consuming, we are using our pruning mechanisms introduced in Section 9 to reduce the search space for instruction candidates. The number of selected candidates, after pruning, is indicated in the *can* column of Table 6.

The third column of Table 6 represents the *pruning efficiency* ratio which is defined as the quotient of two terms. The first term is the ratio of the average maximum ASIP

speedup to the runtime of the identification algorithm when no pruning is used. The second term is the same ratio when using the *@50pS3L* pruning mechanism. The pruning efficiency can be used as a metric to describe the relative gain in the speedup-to-identification-time ratio with and without pruning.

The *blk* and *ins* columns represent the number of basic blocks and instructions which have been passed to the identification process. These numbers are significantly lower than the total number of blocks and instructions presented in the *6th* and *7th* columns of Table 4. That is, the pruning mechanism reduced the size of the bitcode that needs to be analyzed in the identification task by a factor of 36.49x and 4.4x for scientific and embedded applications, respectively.

The overall runtime of the data pruning, identification, estimation, and selection is aggregated in the *real* column. The total candidate search time is in the order of milliseconds and thus insignificant in comparison to the overheads involved in the hardware generation.

13.2. Performance Improvements. The column *ASIP ratio* represents the speedup of the augmented hardware architecture when all candidates selected by Candidate Search are offloaded from the software to custom instructions. In contrast to the maximum performance shown in the 11th column in Table 5 which assumes that *all* candidates are moved to hardware, the average speedup drops by 30% from 1.71x–1.20x for scientific applications and by 46% from 9.19x–4.98x for the embedded ones. Comparing the *fft* with the *470.lbm* applications illustrates the main difference between embedded and scientific applications. Both applications have a similar speedup of 2.40x versus 2.53x, respectively, but differ significantly in the number of candidates that need to be translated to hardware to achieve these speedups (14 versus 179 candidates). This correlates with the previously described observation that scientific applications have a significantly larger kernel size.

13.3. Netlist Generation. The tasks discussed in this section are represented by the second phase in Figure 2. The task *Generate VHDL* is performed with the PivPav datapath generator which produces the *structural VHDL* code. The datapath generator traverses the datapath graph of the candidate and matches every node with a VHDL component. This is a constant time operation requiring 0.2 s per candidate. The *extract netlist* task retrieves the netlist files for each hardware component used in the candidate's VHDL description from the PivPav database. This step allows reduction of the FPGA CAD tool flow runtimes, since the synthesis process needs to build only a final netlist for the top module. The next step is to *create the FPGA CAD project* which is performed by PivPav with the help of the *TCL* scripting language. After the project is created, it is configured with the FPGA parameters and the generated VHDL code as well as the extracted netlist files are added. On average this process took 2.5 s per candidate, making this the most time-consuming task of the netlist generation phase. The average total runtime for these three tasks is presented in the *C2V* column of Table 7 and amounts to 3.22 s. As the standard deviation is only 0.10, this time can be considered as constant.

Table 7: Constant overheads involved in the ASIP-SP. *C2V* corresponds to the *Netlist Generation* phase in Figure 2. Syn, Xst, Tra, and Bitgen are the FPGA CAD tool flow processes and correspond to the syntax check, synthesis, translate, and partial reconfiguration bitstream generation processes, respectively, which can be found in the third phase in Figure 2.

	C2V [s]	Syn [s]	Xst [s]	Tra [s]	Bitgen [s]	Sum [s]
Average	3.22	4.22	10.60	8.99	151.00	178.03
Stdev	0.10	0.10	0.23	1.22	2.43	

13.4. Instruction Implementation. Once the project is created it can be used to generate the partial reconfiguration bitstream representing the FPGA implementation of the custom instruction. This step is performed with the FPGA CAD tool flow which includes several steps. First, the VHDL source code is checked for any *syntax* errors. The runtime of this task is presented in the second column of Table 7. On average it takes 4.22 s to perform this task for every candidate. Since the *stdev* is very low (0.10) we can assume that this is a constant time too.

Once the source code is checked successfully the *synthesis* process is launched. Since all the netlists for all hardware components are retrieved from a database there is no need to resynthesize them. The synthesis process thus has to generate a netlist just for the top-level module which on average took 10.60 s. The runtime of this task does not vary a lot since the VHDL source code for all candidates has a very similar structure and changes only with the number of hardware components. After this step all netlists and constraint files are consolidated into a single database with the *translate* task, which runs for 8.99 s on average.

In the next step, the most computationally intensive parts of the tool flow are executed. These are the *mapping* and the *place and route* tasks which are not constant time processes as the previous tasks, but their duration depends on the number of hardware components and the type of operation they perform. For instance, the implementation of the shift operator is trivial in contrast to a division. The spectrum of runtimes for the *mapping* process ranges from 40 s for small candidates up to 456 s for large and complex ones, whereas the *place and route* task takes 56 s–728 s. There is no strict correlation between the duration of these processes; the ratio of *place and route* and *mapping* runtimes vary from 1.4x for small candidates to 2.5x for large candidates. The last step in the hardware custom instruction generation process is the bitstream generation. Our measurements show that this is again a constant time process depending only on the characteristics of the chosen FPGA. Surprisingly, the runtime of this task is substantial. On average, 151 s per candidate are spent to generate the partial reconfiguration bitstream. This runtime is constant and does not depend on the characteristics of a candidate. In many cases, the bitstream creation consumed more time than all other tasks of the instruction

synthesis process combined (including synthesis and place-and-route). The runtime is mainly caused by using the early access partial reconfiguration Xilinx 12.2 FPGA CAD tools (EAPR). In comparison, creating a full-system bitstream that includes not only the custom instruction candidate but also the whole rest of the FPGA design takes just 41 s on average when using the regular (non-EAPR) Xilinx FPGA CAD tools.

In Table 7, we summarize the runtime of the processes which cause constant overheads that are independent of the candidate characteristics. These are the *Candidate to VHDL translation (C2V)*, *Syntax Check (Syn)*, *Synthesis (Xst)*, *Translation (Tra)*, and *Partial Reconfiguration Bitstream Generation (Bitgen)*. The total runtime for these processes is 178.03 s and is inevitable when implementing even the most simple custom instruction. The *Bitgen* process accounts for 85% of the total runtime.

The overall runtime involved in the FPGA CAD Tool Flow execution is presented in the column *Runtime Overheads* in Table 6. The column *const* represents the runtime of constant processes shown in Table 7. The column *map* stands for the mapping process, the column *par* for the *place and route*, and the values in the column *sum* adds all three columns together. These columns aggregate the total runtime involved in the generation of all candidates for a given application. The candidate's partial reconfiguration times were not included in these runtimes since they consume just a fraction of a second [6]. On average it takes less than 50 minutes (49 : 53 min) to generate all candidates for the embedded applications but more than 4 : 30 hours (270 : 28 min) for the scientific applications. One can see that this large difference is closely related to the number of candidates and that *sum* column grows proportionally with the number of candidates. This behavior can be observed for example for the *444.namd* and the *470.lbm* applications, which consist of 179 and 129 candidates, respectively. The total runtime overhead for them is more than 11 hours (678 : 13 min) and 17 hours (1021 : 22 min), respectively and is caused primarily by the high constant time overheads (*const*).

This observation emphasizes the importance of the pruning algorithms, particularly for the large scientific applications. We can observe the difference for the embedded applications where a smaller number of candidates exists. On average, the *const* time drops for the scientific applications from 146 : 34 min to 24 : 28 min, that is, by a factor of 5.99x, which is exactly the difference in the number of candidates *(can)* between the scientific and the embedded applications.

13.5. Break-Even Times. In this section, we analyze the *break-even time* for each application; that is, the minimal time each application needs to execute before the overheads caused by the ASIP-SP is compensated.

A simplistic way of computing the break-even time would be to divide the total runtime overhead (*sum* in Table 6) by the time saved during one execution of the application, which can be computed using the *VM execution time* and the *Max ASIP Speedup* (speedup) (see Table 5). This computation assumes a scenario, where the size of the input data is fixed and the application is executed several times.

We have followed a more sophisticated approach of computing the break-even time, which assumes that more input data is processed instead of multiple execution of the same application. Hence, the additional runtime is spent only in the parts of the code which are *live* while code parts that are *const* or *dead* are not affected. To this end, we use the information about the execution frequency of basic blocks and the variability of this execution frequency for different benchmark sizes which we have collected during profiling; see Section 11.3. The resulting break-even times are presented in the last column of Table 6.

It is evident that there exists a major difference in the break-even times for the embedded and the scientific applications. While the break-even time of the embedded applications is in the order of minutes to a few hours, the scientific applications need to be executed for days to amortize the overhead caused by custom instruction implementation (always under the assumption that *all* candidates are implemented in hardware). The reason for these excessive times is the combination of rather long ASIP-SP runtimes *(>4 : 30 h)* and modest performance gains of 1.2x. As described above, the long runtimes are caused by implementing many candidates. One might expect that this large number of custom instructions should cover a sizable amount of the code and that significant speedups should be obtained, but evidently this is not the case. The reason for this is that the custom instructions are rather small, covering only 6.9 IR instructions on average. Although there are many custom instructions generated, they cover only a small part of the whole computationally intensive kernels of the scientific application, which has a size of 1764 IR instructions on average. Adding more instructions will not solve this issue since every candidate adds an additional FPGA CAD tool flow overhead.

In contrast, the break-even point for embedded applications is reached more easily. On average, the break-even time is five orders of magnitude lower for these applications. In contrast to the scientific applications, the custom instructions for embedded application can cover a significant part of the computationally intensive kernel. This results in reasonable performance gains with modest runtime overheads. For an average-embedded application, a 5x speedup can be achieved, resulting in a runtime overhead of less than 50 minutes and a break-even time of less than 2 hours.

The difference between scientific and embedded applications is not caused by a significant difference in the number of IR instructions in the selected candidates. Scientific applications have on average 7.31 instructions per candidate, while embedded applications have on average 6.5 instructions per candidate.

Since we cannot decrease the size of the computational kernel, we should strive for finding larger candidates in order to cover a larger fraction of the kernel. Unfortunately, this turns out to be difficult because the reason that the candidates are small is that the BBs *(blk)* in which they are identified are also small. The average basic block has only 7.64 (5.94) IR instructions for a scientific (embedded) application (see Table 5).

The pruning mechanism we are using is directing the search for custom instruction to the largest basic blocks;

hence, the average basic block that passes the pruning stage has 155.65 instructions for a scientific and 29.71 for embedded application (see Table 6). However, even these larger blocks include a sizable number of the hardware-infeasible instructions, such as accesses to global variables or memory, which cannot be included in a hardware custom instruction. As a result, there are only 7.31 instructions per candidate in a scientific application which causes high break-even times for them.

This observation illustrates that there are practical limitations for the ASIP-SP when using code that has been compiled from imperative languages.

14. Reduction of Runtime Overheads

In this section, we propose two approaches for reducing the total runtime overheads and in turn also the break-even times: partial reconfiguration bitstream caching and acceleration of the CAD tool flow.

14.1. Partial Reconfiguration Bitstream Caching. As in many areas of computer science, caching can be applied also in the context of our work. Much like virtual machines cache the binary code that was generated on the fly for further use, we can cache the generated partial bitstreams for each custom instruction. To this end, each candidate needs to have a unique identifier that is used as a key for reading and writing the cache. We can, for example, compute a signature of the LLVM bitcode that describes the candidate for this purpose. The cached bitstreams can be stored for example in an on-disk database.

14.2. Acceleration of the CAD Tool Flow. A complementary method for reducing the runtime overheads is to accelerate the FPGA CAD tool flow. There are several options to achieve this goal. One possibility is to use a faster computer that provides faster CPUs and faster and larger memory or to run the FPGA tool concurrently. Alternatively, it may be possible to use a smaller FPGA device, since the *constant* processes of the tool flow depend strongly on the capacity of the FPGA device. We have used a rather large *Virtex-4 FX100* device, therefore switching to a smaller device would definitely reduce the runtime of the tool flow. Another option would be to use a memory file system for storing the files created by the tool flow. As the FPGA CAD tool flow is known to be I/O intensive, this should speed up the tool flow. Finally, we could change our architecture to a more coarse-grained architecture with simplified computing elements and limited or fixed routing. It has been shown that it is possible to develop customized tools for such architectures which work significantly faster [30].

14.3. Extrapolation. In Table 8 we calculate the average *breaking-even time* for the embedded applications when applying these ideas. When the cache is disabled and we do not assume any performance gain from the tool flow, the first value is equal to the *AVG_E* row and the last column in Table 6. One can note also that these values do not scale linearly because we consider the frequency information for basic blocks.

Table 8: The average *breaking-even time* for the embedded applications using a *partial reconfiguration bitstream* cache and a faster FPGA CAD tool flow.

	Faster FPGA CAD tool flow [%]			
Cache hit [%]	0 [h:m:s]	30 [h:m:s]	60 [h:m:s]	90 [h:m:s]
0	01:59:55	01:24:48	00:48:27	00:12:07
10	01:47:44	01:15:25	00:43:06	00:10:46
20	01:32:59	01:05:05	00:37:11	00:09:18
30	**01:28:09**	**01:01:42**	**00:35:15**	**00:08:49**
40	01:13:08	00:51:11	00:29:15	00:07:19
50	01:01:00	00:42:42	00:24:24	00:06:06
60	00:48:50	00:34:10	00:19:32	00:04:53
70	00:35:12	00:24:38	00:14:05	00:03:31
80	00:29:19	00:20:31	00:11:43	00:02:56
90	00:14:07	00:09:53	00:05:39	00:01:24

For this evaluation, we varied the assumed cache hit rate to be between 0%–90%. That is, for simulating a cache with a 20% hit rate, we have populated the cache with 20% of the required bitstreams for a particular application, whereas the selection whose bitstreams are stored in the cache is random. Whenever there is a *hit* in the cache for a given candidate, the whole runtime associated with the generation of the candidate is subtracted from the total runtime; see *sum* column in Table 6. The values in the *Faster FPGA CAD tool flow* columns are decreasing linearly with the assumed speedup.

If we assume that the FPGA CAD tool flow can be accelerated by 30% and that we have 30% cache hits, the average break-even time drops almost by half (1.94x), from *1:59:55 h* to *1:01:42 h*. This means that the whole runtime of the ASIP-SP could be compensated in a bit more than one hour and for the rest of the time the adapted architecture would provide a performance gain by an average factor of 5x. These assumptions are modest values since the *cache hit* rate depends only on the size of the cache, and our Dell T3500 workstation could be easily replaced by a faster one.

15. Conclusion and Future Work

In this work, we have studied the just-in-time ASIP-SP for an FPGA-based reconfigurable architecture. The most significant parts of this process, including the candidate identification, estimation, selection, and pruning mechanisms were not only described with precise formalisms but were also experimentally evaluated. In particular, we discussed and compared characteristics of three state-of-the-art instruction set extension algorithms in order to study the candidate identification mechanism in detail. This study included not only algorithm runtimes, number of found UDCI candidates, their properties, and impact of algorithm constraints on the search space, but more importantly the achievable maximum

performance gains for various embedded computing and scientific benchmark applications.

The study of the ASIP process was performed both separately for every element but more importantly also for the entire ASIP process, where the feasibility and limitations were investigated. The study has shown that for embedded applications an average speedup of 5x can be achieved with a runtime overhead of less than 50 minutes. This overhead can be compensated if the application executes for two hours or for one hour when assuming a 30% cache hit rate and a faster FPGA CAD tool flow. Our study further showed that the larger and more complex software kernels of scientific applications, represented by the SPEC benchmarks, do not map well to custom hardware instructions targeting the Woolcano architecture and lead to excessive times until the break-even point is reached. The reason for this limitation can be found in the properties of the intermediate code generated by LLVM when compiling C code, in particular, rather small basic block sizes with an insufficient amount of instruction level parallelism. Similar results are expected for other imperative languages. Simultaneously, this work has explored the potential of our Woolcano reconfigurable architecture, the ISE algorithms, and pruning mechanism for them as well as the PivPav estimation and datapath synthesis tools.

References

[1] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*, Morgan Kaufmann, San Francisco, Calif, USA, 2006.

[2] M. J. Wirthlin and B. L. Hutchings, "Dynamic instruction set computer," in *Proceedings of the 3rd IEEE Symposium on FPGAs for Custom Computing Machines, (FCCM '95)*, pp. 99–107, IEEE Computer Society, April 1995.

[3] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th International Symposium on Microarchitecture, (MICRO '94)*, pp. 172–180, ACM, New York, NY, USA, November 1994.

[4] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the 27th Annual International Symposium on Computer Architecture, (ISCA '00)*, pp. 225–235, ACM, June 2000.

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

[6] M. Grad and C. Plessl, "Woolcano: an architecture and tool flow for dynamic instruction set extension on Xilinx Virtex-4 FX," in *Proceedings of the 9th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '09)*, pp. 319–322, CSREA Press, Monte Carlo Resort, Nev, USA, July 2009.

[7] J. M. Arnold, "S5: the architecture and development flow of a software configurable processor," in *Proceedings of the International Conference on Field Programmable Technology, (ICFPT '05)*, pp. 121–128, IEEE Computer Society, Kent Ridge Guild House, Singapore, December 2005.

[8] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, 1999.

[9] M. Grad and C. Plessl, "An open source circuit library with benchmarking facilities," in *Proceedings of the 10th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA, '10)*, T. P. Plaks, D. Andrews, R. F. DeMara et al., Eds., pp. 144–150, CSREA Press, Las Vegas, Nev, USA, July 2010.

[10] M. Grad and C. Plessl, "Pruning the design space for just-in-time processor customization," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '10)*, pp. 67–72, IEEE Computer Society, Cancun, Mexico, December 2010.

[11] M. Grad and C. Plessl, "Just-in-time instruction set extension—feasibility and limitations for an FPGA-based reconfigurable ASIP architecture," in *Proceedings of the 18th Reconfigurable Architectures Workshop, (RAW '11)*, pp. 278–285, IEEE Computer Society, May 2011.

[12] M. Wazlowski, L. Agarwal, T. Lee et al., "PRISM-II compiler and architecture," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM '93)*, pp. 9–16, IEEE Computer Society, April 1993.

[13] C. Galuzzi and K. Bertels, "The instruction-set extension problem: a survey," in *Proceedings of the International Conference on Architecture of Computing Systems, (ARCS '08)*, Lecture Notes in Computer Science, no. 4943, pp. 209–220, Springer/Kluwer Academic, Dresden, Germany, February 2008.

[14] R. J. Hookway and M. A. Herdeg, "DIGITAL FX!32: combining emulation and binary translation," *Digital Technical Journal*, vol. 9, no. 1, pp. 3–12, 1997.

[15] V. Bala, E. Duesterwald, and S. Banerjia, "Transparent dynamic optimization," Tech. Rep. number HPL-1999-78, HP Laboratories Cambridge, 1999.

[16] K. Ebcioglu and E. R. Altman, "DAISY: dynamic compilation for 100% architectural compatibility," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 26–37, New York, NY, USA, June 1997.

[17] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: dynamic translation of binaries to FPGA circuits," *Computer*, vol. 41, no. 7, pp. 40–46, 2008.

[18] A. C. S. Beck and L. Carro, "Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility," in *Proceedings of the 42nd Design Automation Conference, (DAC '05)*, pp. 732–737, New York, NY, USA, June 2005.

[19] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209–1229, 2006.

[20] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES '04)*, pp. 69–78, Washington, DC, USA, September 2004.

[21] C. A. William, W. Fornaciari, L. Pozzi, and M. Sami, "A DAG-based design approach for reconfigurable VLIW processors," in *Proceedings of the Design, Automation and Test in Europe Conference, (DATE '99)*, pp. 778–779, ACM, Munich, Germany, January 1999.

[22] J. Gong, D. D. Gajski, and S. Narayan, "Software estimation from executable specifications," *Journal of Computer Software Engineering*, vol. 2, pp. 239–258, 1994.

[23] *The PowerPC 405TM Core*, IBM, 1998.

[24] A. Ray, T. Srikanthan, and W. Jigang, "Practical techniques for performance estimation of processors," in *Proceedings of the International Workshop on System-on-Chip for Real-Time Applications, (IWSOC '05)*, pp. 308–311, IEEE Computer Society, Washington, DC, USA, 2005.

[25] B. So, P. C. Diniz, and M. W. Hall, "Using estimates from behavioral synthesis tools in compiler-directed design space exploration," in *Proceedings of the 40th Design Automation Conference*, pp. 514–519, New York, NY, USA, June 2003.

[26] *Floating-Point Operator v5.0*, Xilinx.

[27] N. Maheshwari and S. S. Sapatnekar, *Timing Analysis and Optimization of Sequential Circuits*, Springer/Kluwer Academic Publishers, Norwell, Mass, USA, 1999.

[28] R. Meeuws, Y. Yankova, K. Bertels, G. Gaydadjiev, and S. Vassiliadis, "A quantitative prediction model for hardware/software partitioning," in *Proceedings of the International Conference on Field Programmable Logic and Applications, (FPL '07)*, pp. 735–739, Amsterdam, The Netherlands, August 2007.

[29] P. Bonzini and L. Pozzi, "Polynomial-time subgraph enumeration for automated instruction set extension," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1331–1336, Nice, France, April 2007.

[30] E. Bergeron, M. Feeley, and J. P. David, "Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC/ETAPS'08)*, pp. 178–192, Springer-Verlag, Berlin, Heidelberg, 2008.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 474765, 16 pages
doi:10.1155/2012/474765

Research Article

Combining SDM-Based Circuit Switching with Packet Switching in a Router for On-Chip Networks

Angelo Kuti Lusala and Jean-Didier Legat

Institute of Information and Communication Technologies, Electronics and Applied Mathematics, Université Catholique de Louvain, 1348 Louvain-la-Neuve, Belgium

Correspondence should be addressed to Angelo Kuti Lusala, angelo.kutilusala@uclouvain.be

Received 1 May 2011; Revised 5 September 2011; Accepted 19 September 2011

Academic Editor: Marco D. Santambrogio

A Hybrid router architecture for Networks-on-Chip "NoC" is presented, it combines Spatial Division Multiplexing "SDM" based circuit switching and packet switching in order to efficiently and separately handle both streaming and best-effort traffic generated in real-time applications. Furthermore the SDM technique is combined with Time Division Multiplexing "TDM" technique in the circuit switching part in order to increase path diversity, thus improving throughput while sharing communication resources among multiple connections. Combining these two techniques allows mitigating the poor resource usage inherent to circuit switching. In this way Quality of Service "QoS" is easily provided for the streaming traffic through the circuit-switched sub-router while the packet-switched sub-router handles best-effort traffic. The proposed hybrid router architectures were synthesized, placed and routed on an FPGA. Results show that a practicable Network-on-Chip "NoC" can be built using the proposed router architectures. 7×7 mesh NoCs were simulated in SystemC. Simulation results show that the probability of establishing paths through the NoC increases with the number of sub-channels and has its highest value when combining SDM with TDM, thereby significantly reducing contention in the NoC.

1. Introduction

Real-time applications have grown in complexity and require more and higher-power computing resources. These applications are then suitable to be run on parallel environments such as MultiProcessor Systems-on-Chip "MPSoC" platforms. However, application performance in an MPSoC platform strongly depends on the on-chip interconnection network used to carry communications between cores in the platform. Since Real-time applications generate both streaming and best-effort traffics, there is then a need for the on-chip interconnection network to provide QoS for the streaming traffic and data completion for the best-effort traffic.

Streaming traffic is best handled in circuit-switched network. Since communication resources are prereserved before any data transfer, QoS is thereby intrinsically supported. Circuit switching often leads to a poor usage of communication resources, since reserved resources for a transaction are exclusively used by that transaction. For that reason, it is not suitable for best-effort traffic. Best-effort traffic is well handled in packet-switched network, however, because of its nondeterministic behavior; packet switching is not suitable for streaming traffic. To improve resource utilization in circuit-switched networks, time division multiplexing "TDM" is often used in order to share resources among multiple connections. TDM consists in dividing a channel in time intervals called time slots; multiple connections can therefore use a given channel by assigning a time slot to each connection. In TDM-based circuit-switched networks, consecutive time slots are reserved in consecutive links along the path between a source node and a destination node. Using TDM, a circuit-switched network can then handle both streaming and best-effort traffic. Reserved time slots are used to carry streaming traffic, while unreserved time slots are used to carry best-effort traffic [1]. However, providing QoS and sharing resources between streaming and best-effort traffic are hard and often lead to a complex design with

huge hardware and power consumption overhead [2]. In packet-switched network, streaming and best-effort traffics are handled by either assigning priorities to each type of traffic, with streaming traffic having the highest priority, [3] or by reserving buffers or virtual channels "VCs" for carrying the streaming traffic, while the unreserved buffers are used to carry the best-effort traffic [4]. The first approach, while providing interesting results for varying traffic, cannot provide strong guarantees denoted "Hard QoS" for real-time applications. The second approach also leads to a complex design, with huge area and power consumption overhead depending on the number of buffers per input port.

In order to efficiently handle both streaming and best-effort traffic in an NoC, we propose a hybrid router which combines circuit switching with packet switching in order to separately and independently handle each type of traffic. The hybrid router then consists of two subrouters: a circuit switched subrouter and a packet switched subrouter. The circuit switched subrouter is responsible for handling streaming traffic, while the packet-switched subrouter is responsible for handling best-effort traffic. In this way, we ensure that each type of traffic is efficiently and suitably handled. In order to improve low resources usage inherent to circuit switching, the circuit-switched subrouter uses SDM and TDM techniques. The SDM technique that we use consists in having more than one link between two adjacent routers. In this way, concurrent data streams are physically separated, thereby increasing path diversity in the router and improving throughput. The TDM technique allows sharing physically separate links among multiple connections. We then define an SDM Channel as a set of links. Each link or subchannel is identified by a number. When the SDM-Channel is shared following the TDM technique, the SDM-Channel is thereby denoted SDM-TDM Channel.

Since circuit-switched subrouters are used to carry the streaming traffic, a path which consists of successive links between a source node and a destination node must first be established before transferring streaming traffic. This task is performed by the packet-switched subrouters by reserving an available subchannel in an SDM-Channel or by reserving a requested time slot at any subchannel in an SDM-TDM channel along the path between the source and the destination nodes. The packet-switched subrouter then configures the attached circuit-switched subrouter by indicating the subchannel to use in an SDM-Channel or by indicating the subchannel and the time slots to use in the SDM-TDM Channel for the concerned connection. When the transfer of the streaming traffic is completed, the circuit-switched subrouter notifies the attached packet-switched subrouter to release reserved resources used to carry the concerned streaming traffic.

In the proposed router architecture, each subrouter independently handles traffic. A node or tile, which can be a processing element "PE" or a storage element "SE", is connected to each subrouter as shown in Figures 1 and 2. When a PE needs to transfer best-effort traffic, it directly sends its "normal or data payload" best-effort packet to the attached packet-switched subrouter for routing through the network hop by hop. When the PE needs to transfer streaming traffic, it first sends a "set-up" best-effort packet to the attached packet-switched subrouter. The set-up best-effort packet is responsible for reserving resources, thereby establishing a path between a source and destination nodes. When a set-up packet reaches its destination, an acknowledgment best-effort packet is generated and routed from destination to source through the packet-switched subnetwork. Upon receiving the acknowledgment packet, the source node then starts transferring streaming traffic, which is segmented in packets like cells in asynchronous transfer mode "ATM" networks. When the transfer of the streaming traffic is completed, the source node sends a teardown streaming packet along the established path whose purpose is to release reserved resources used for the concerned streaming traffic.

Since the circuit-switched subrouters and the packet-switched subrouters do not share links, and avoiding the use of the store-and-forwardstrategy, there is then no need to use FIFO buffers in the circuit-switched subrouter to store streaming packets unlike in [5]. This significantly reduces the area and power consumption of the router. Combining SDM and TDM techniques in a router allows taking advantages of the abundance of wires resulting from the increased level of CMOS circuits. We then have two degrees of freedom to optimize the router; one can increase either the number of subchannels in an SDM-TDM Channel or the number of time slots per subchannel. In both cases, the number of available channels increases in the network, thereby increasing the possibilities of establishing paths through the network.

The proposed hybrid router architectures were implemented in Verilog and synthesized on FPGA with different number of subchannels in an SDM-Channel and for different numbers of subchannels and time slots in an SDM-TDM Channel. Synthesis results show that increasing the number of subchannels in an SDM-Channel does not significantly impact the size of the router, while the clock frequency is only slightly reduced. When combining SDM and TDM techniques, increasing the number of subchannels, while maintaining fixed the number of time slots significantly impacts the size of the router, while the maximum clock frequency remains almost constant; increasing the number of time slots while maintaining fixed the number of subchannels does not significantly impact the size of the router while it significantly reduces the clock frequency. In order to evaluate the performance of the proposed architectures in terms of established paths through the network according to the number of set-up requests packets, three 7×7 2D meshes NoCs were simulated in SystemC under random uniform traffic and compared: an SDM-based hybrid NoC, a TDM-based hybrid NoC, and an SDM-TDM-based hybrid NoC. Simulation results show that combining SDM and TDM techniques in a router substantially increases the probability of establishing paths through the network, while this probability is appreciable in the SDM-based NoC and small in the TDM-based NoC.

The rest of the paper is organized as follows. Related work is reviewed in Section 2. Section 3 introduces the proposed router architectures. Section 4 discusses simulation

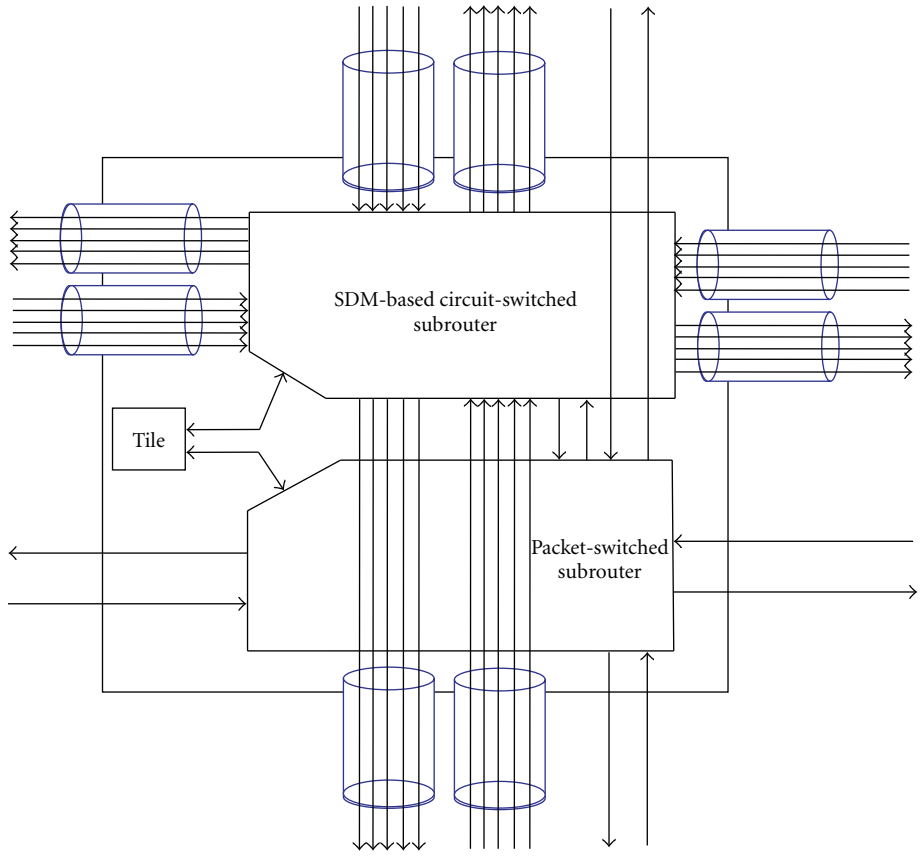

Figure 1: Hybrid SDM-based router architecture.

and synthesis results of the proposed router architectures. Finally, Conclusions are drawn in Section 5.

2. Related Work

Many hybrid NoCs have been proposed in the literature. Some of them deal with topological aspects by combining several topologies in a single NoC [6]; others combine different switching techniques in order to either provide "QoS" for streaming traffic while supporting best-effort traffic or reduce average packet latency in the network [5, 7, 8]. In this paper, we focus on hybrid NoCs which combine different switching techniques.

In [5], ÆTHEREAL NoC is presented. It consists of two disjoint subnetworks: a guaranteed service "GS" subnetwork and a best-effort "BE" subnetwork. The GS subnetwork is circuit-switched, while the BE network is packet-switched. TDM is used in order to share the same links between the BE and the GS subnetworks. Reserved time slots are used to carry the streaming traffic, while the unreserved time slots are used to carry the best-effort traffic. The BE subnetwork is responsible for establishing paths for the streaming traffic through the GS subnetwork by reserving time slots and thus configuring the GS subnetwork. For this purpose, four types of best-effort packets are used: a set-up packet which is responsible for establishing paths through the network by reserving time slots; an ACK packet which is generated when a set-up packet reaches its destination; an NACK packet which is generated when a set-up packet fails and is responsible for releasing reserved time slots in the previous crossed packet-switched subrouters; a teardown packet which is responsible for releasing reserved resources when the streaming traffic transfer is completed. The GS subrouter uses the store-and-forward strategy, while the BE subnetwork uses the wormhole strategy. Despite the fact that TDM is simple to implement, the use of buffers in both GS and BE subrouters and the necessity of a memory device to store the configuration of the shared resources lead to a complex design with a huge area and power consumption overhead [9]. Our proposed hybrid router architecture uses a similar approach by having two distinct subrouters; however, our proposed circuit-switched subrouter is SDM or SDM-TDM based; by avoiding the use of the store and forward strategy, there is therefore no need to use FIFO buffers in the circuit-switched subrouter. However, when combining SDM with TDM, simple registers are required in order to schedule streaming packets to travels through the network in pipeline fashion at the reserved time slots. Furthermore, the two subrouters do not share links; this makes it easy to separately design and optimize each part of the router.

In [7], a hybrid NoC which uses a technique called hybrid circuit switching "HCS" is presented. It consists of a network design which removes the set-up time overhead in circuit-switched network by intermingling packet-switched

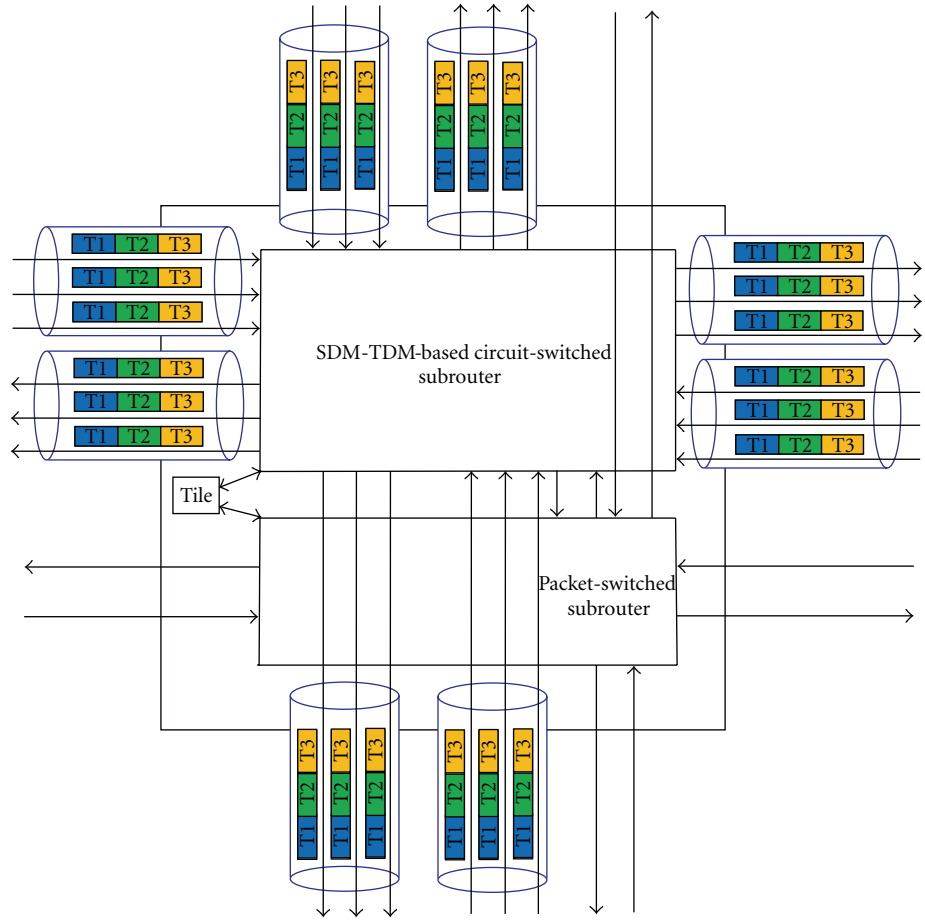

Figure 2: Hybrid SDM-TDM router architecture.

flits with circuit-switched flits. In this architecture, there is no need to wait for an acknowledgment that a circuit has been successfully established; data can then be injected immediately behind the circuit set-up request. If there is no unused resource, then the circuit-switched packet is transformed to a packet-switched packet and buffered; it will then keep its new state until it is delivered. With this approach, it is still difficult to provide hard "QoS" for streaming traffic.

The work presented in [8] is similar to the one presented in [7]. Since in packet switching it is very difficult to predict latency and throughput, sharing the same resource between packet-switched and circuit-switched networks makes it difficult to provide QoS for streaming traffic. In [9] is presented one of the first works using SDM in NoC in order to provide QoS for streaming traffic; however, this NoC does not handle best-effort traffic. In this work, a subset of links constituting an SDM-Channel are allocated to connections according to their bandwidth needs. The authors claim a gain in area and power consumption compared to the TDM approach but with the cost of a huge delay in the SDM switch which significantly limits the scalability of the approach. In the SDM variant that we propose, a connection can only acquire one link among links constituting the SDM-Channel. This significantly reduces the complexity of the switch. Furthermore, we combine SDM with TDM in the circuit-switched subrouter while handling best-effort traffic in a packet-switched subrouter.

3. Proposed Router Architecture

3.1. Router Architecture. The proposed router architecture consists of two major components as illustrated in Figure 1: a packet-switched subrouter and an SDM-based circuit-switched subrouter. The two subrouters are distinct and independently handle traffic. The SDM-based circuit-switched subrouter is responsible for carrying streaming traffic and is configured by the packet-switched subrouter. The SDM-based circuit-switched subrouter notifies the packet-switched, when the transfer of the streaming traffic transfer is completed in order to release reserved resources. The packet-switched subrouter carries best-effort traffic.

The use of SDM technique, by allowing multiple simultaneous connections, mitigates the impact of the poor usage of resources in circuit switching, however the reserved resource (subchannel) is only used by one connection. To improve resource utilization, the SDM technique is combined with the TDM technique as shown in Figure 2. Therefore, a subchannel can be used by multiple connections. As seen previously, an SDM-Channel consists of a set of a given

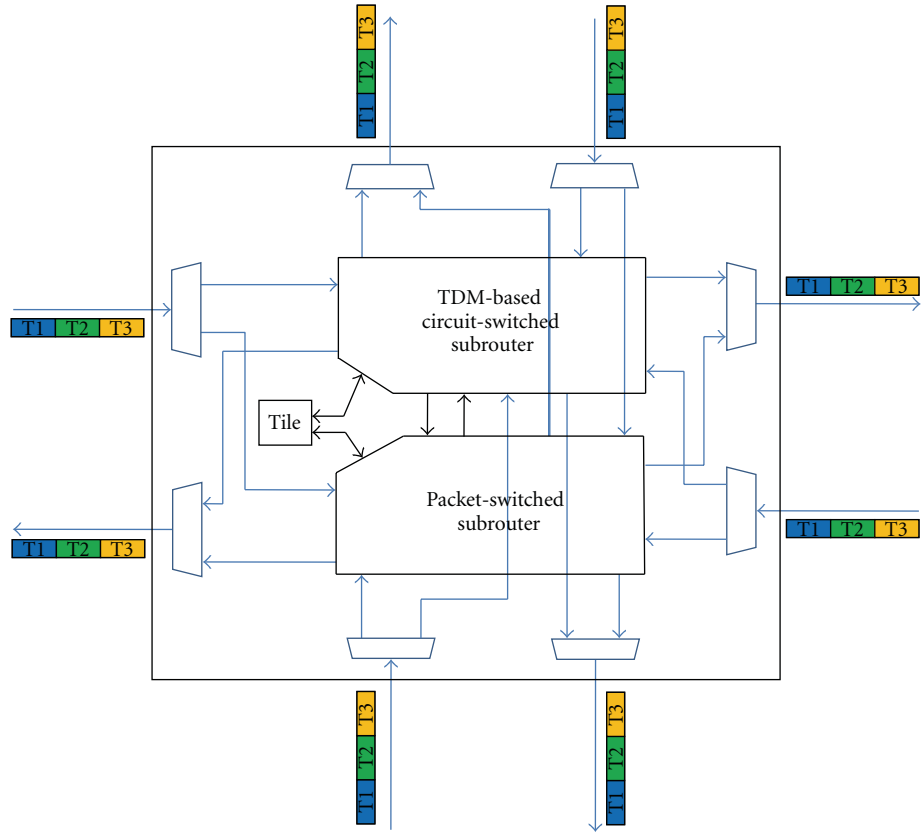

Figure 3: Hybrid TDM-based router architecture.

number of subchannels, while an SDM-TDM Channel is an SDM-Channel shared in time. Each subchannel is n-bits wide. In the SDM-based router, a connection can only acquire one subchannel and exclusively uses it until the end of the transaction, while in SDM-TDM-based router a connection can only acquire one subchannel but uses it at a specific time slot which is assigned to that connection; in the remaining time slots, the subchannel can be used by other connections.

The use of TDM allows sharing links between a circuit-switched subrouter and a packet-switched subrouter as shown in Figure 3. However, the scheduling constraint on time slots reservation which imposes that when a time slot T_i is reserved in a router the time slot $(T_i + 1)$ *modulo* S must be reserved in the next router along the path between a source and destination, and S is the number of time slots, constitutes a bottleneck in TDM-based network since it can limit significantly the number of established paths through the network and best-effort packets can experience a huge delay in the network when all time slots are reserved. Increasing the number of time slots does not solve efficiently this problem while increasing the size of the router. Since SDM allows increasing the probability of establishing paths through the network [10], therefore combining SDM with TDM can efficiently solve the problem of the scheduling on time slots reservation.

To illustrate the benefits of combining SDM and TDM techniques, let us consider the hybrid routers shown above. For the hybrid TDM-based router represented in Figure 3 with 3 time slots, a set-up request packet in any direction should have three possibilities to reserve a time slot; however, the scheduling constraint on time slot reservation imposes the time slot to reserve, thereby reducing the possibilities to choose a time slot from three to one. Let us now consider the SDM-based hybrid router shown in Figure 1 with 3 subchannels. Since there is no constraint on choosing a subchannel, a set-up request packet has three possibilities to choose a subchannel. Finally, let us consider the SDM-TDM-based hybrid router shown in Figure 2 with 3 subchannels in an SDM-Channel and each subchannel shared with 3 time slots. In this case taking in account the scheduling constraint on time slot reservation, there are three possibilities for the set-up request packet to choose the requested time slot. This means that, for the three considered cases, at a given time slot, the probability to establish a path in an SDM, and SDM-TDM-based hybrid NoC is three times greater than in the TDM-based router. However, the SDM-Channel can support up to 3 connections, while the SDM-TDM Channel can support up to fifteen connections.

3.2. Packet-Switched Subrouter. The packet-switched subrouter is responsible for routing best-effort traffic and

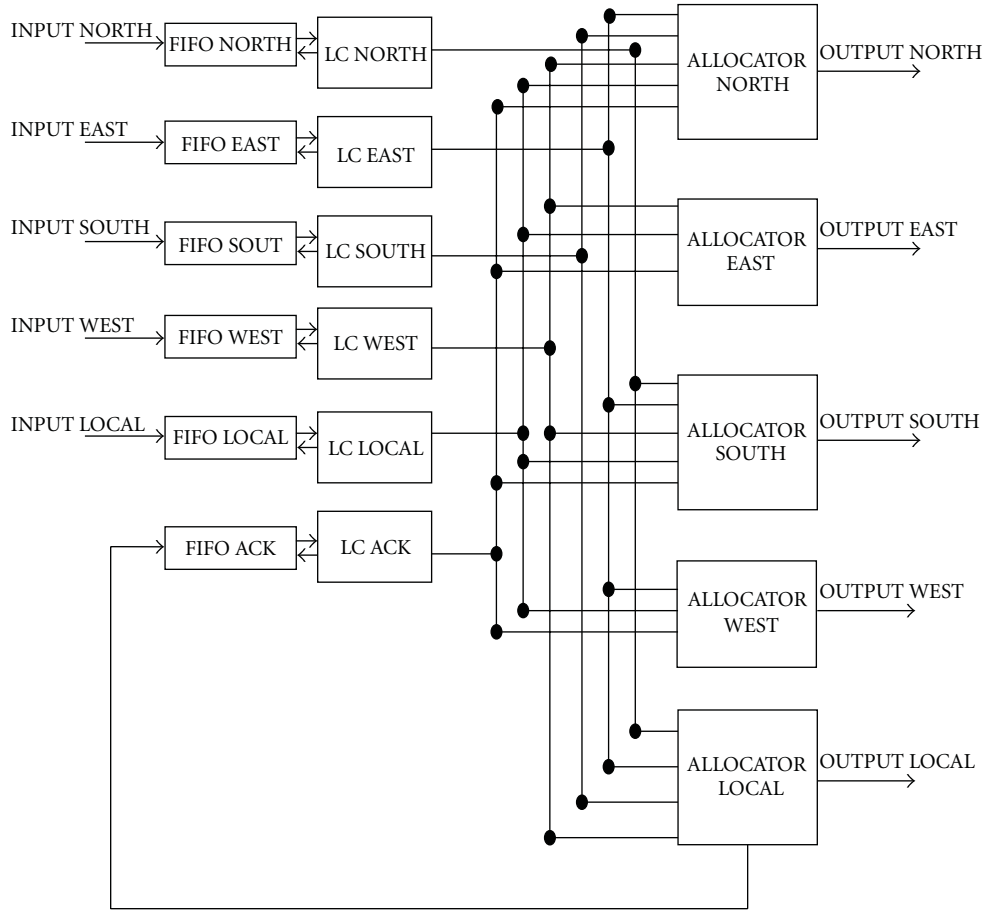

Figure 4: Packet-switched subrouter.

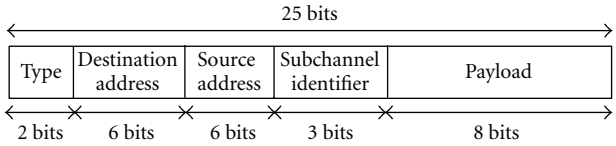

Figure 5: Best-effort packet format SDM-based router.

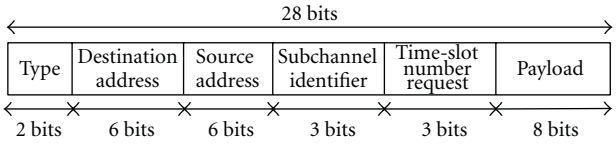

Figure 6: Best-effort packet format SDM-TDM router.

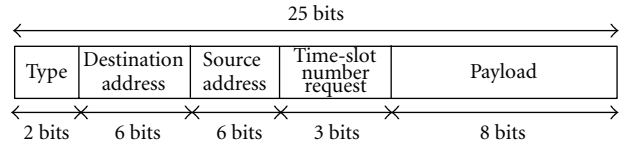

Figure 7: Best-effort packet format TDM-based router.

configuring the attached SDM- or SDM-TDM-based circuit-switched subrouter as shown in Figures 1 and 2. It uses *XY* deterministic routing algorithm with cutthrough as control flow strategy. Routing is distributed so that up to five packets can simultaneously be routed when they request different output channels.

The packet-switched subrouter consists of input FIFO buffers, link controllers, and allocators as shown in Figure 4. The input FIFO buffers store the incoming best-effort packets. The link controllers are responsible for routing the best-effort. Depending to the destination address, they decide to which allocator the fetched packet should be sent. The link controller keeps the fetched packet in a register until it receives a signal from the allocator which indicates that the packet is successfully sent to the output port. This strategy ensures that no packet is lost in the network.

A best-effort packet consists of five fields for SDM- and TDM-based hybrid routers and six fields for the SDM-TDM-based router as shown in Figures 5, 6, and 7, respectively. Two bits indicating the type of the best-effort packet, the destination, and the source addresses are 6-bit wide, allowing building a 7×7 2D mesh NoC, the subchannel identifier and the requested time slot are 3-bit wide, and the payload is 8-bit wide. We define three types of best-effort packets in the

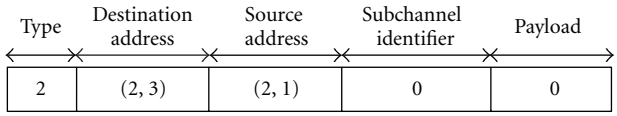

Figure 8: Set-up packet node source (2,1).

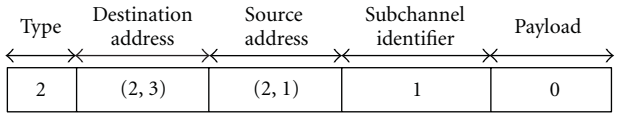

Figure 9: Incoming set-up packet allocator EAST (2,2).

proposed hybrid router.

(i) Set-up request best-effort packet.

(ii) ACK best-effort packet.

(iii) Normal best-effort packet.

The set-up request packet is responsible for reserving resources which are subchannels for the SDM-based router, subchannels and requested time slots for the SDM-TDM-based router, and requested time slots for the TDM-based router. The set-up request packet thereby establishes a path between a source node and a destination node. Its payload is zero and its type is 2. The ACK packet, which is generated when a set-up request packet reaches its destination, is responsible for notifying the source node that the path is successfully established. Its type is 1and its subchannel identifier, time slot request number, and payload fields are zero, respectively. The Normal best-effort packet carries the best-effort payload. Its type is 3and its subchannel identifier and time lot number request fields are zero respectively.

The allocators are responsible for forwarding best-effort packets to the output ports, reserving resources and configuring the attached circuit-switched subrouter. They first check the type of the best-effort packet. If the packet is an ACK or a normal packet, the allocator directly sends it to the output link without modifying it. If the best-effort packet is a set-up request packet, then the SDM-TDM-based router, allocator reserves an available requested time slot at any available subchannel in the SDM-TDM Channel in the concerned direction and builds a new set-up packet by replacing the fields subchannel number and time slot number request of the incoming packet by the number identifier of the reserved subchannel and the time slot number to request in the next hop. For the SDM-based router, the allocator only reserves a subchannel in the SDM-Channel and builds a new set-up packet by replacing the field subchannel identifier of the incoming set-up request packet by the number identifier of the reserved subchannel. In both cases, the value of the subchannel identifier in the incoming set-up request packet and the value of the subchannel in the outgoing set-up request packet are concatenated, and the result is stored in a register which is denoted "*reg_identifier*". Its MSB is the incoming subchannel and its LSB is the outgoing subchannel. Each subchannel has its own

Figure 10: Outgoing set-up packet allocator EAST (2,2).

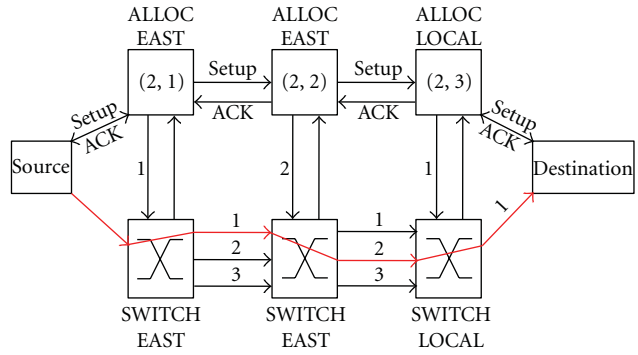

Figure 11: SDM path between source (2,1) and destination (2,3).

reg_identifier. This register helps to retrieve the subchannel to release when a NACK signal is received.

3.2.1. Set-up Path Phase in SDM-Based Router. To illustrate the process of path establishment in the SDM-based router, let us consider an SDM-channel consisting of 3 subchannels; the subchannel identifiers are 1, 2, and 3, respectively. Let us consider a set-up path phase between a source node attached to the router with coordinates (2,1) and a destination node attached to the router with coordinates (2,3) as shown in Figure 11. The set-up request packet from the source node is given in Figure 8; the fields subchannel identifier and payload are zero.

At the router (2,1), the allocator EAST, reserves an available subchannel in the SDM-Channel output port. Suppose that the reserved subchannel is the subchannel 1, the allocator then builds the outgoing set-up request packet with the identifier of the reserved subchannel and concatenates the value of the subchannel identifier in the incoming set-up request packet with the value of the channel identifier in the outgoing set-up request packet. It then stores this value in the *register identifier* of subchannel 1 which is the outgoing subchannel. At the router (2,2), let us assume that subchannel 1 in this allocator is already reserved by another set-up request packet and the remaining subchannels are available. The allocator will then reserve for example subchannel 3. It builds the outgoing set-up request packet (Figure 10) and concatenates the value of subchannel identifier in the incoming set-up packet (Figure 9) with the value of the subchannel identifier in the outgoing set-up request packet and stores this value in the *register identifier* of subchannel identifier 2.

At the router (2,3), the allocator LOCAL reserves the subchannel if it is free. It then generates an ACK packet, which is routed through the packet-switched subnetwork from the destination to the source. Upon reception of

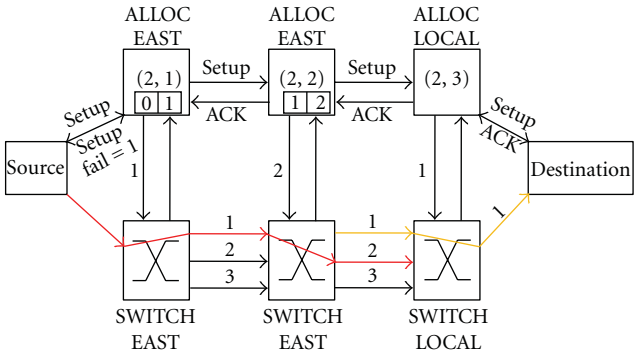

Figure 12: Failed path establishment in SDM-based router.

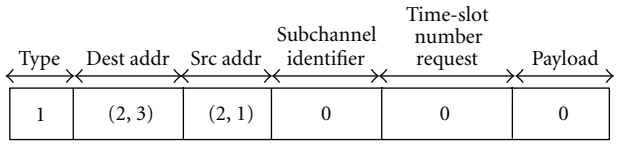

Figure 13: SDM-TDM set-up packet node source (2,1).

the ACK packet, the source node then starts transferring streaming traffic. For simplicity, we represent in Figure 11 this process with only the concerned allocators and switches.

When a set-up request packet fails to reserve a subchannel in a hop, the NACK signal is generated and propagates to all previous crossed packet subrouter in order to release the reserved subchannels by the failed set-up request packet. The NACK signal at the router where it fails is equal to the subchannel value contained in the incoming set-up request packet; it indicates the subchannel to release in the previous packet-switched subrouter. At the previous subrouter, the value of the NACK to propagate is the MSB of the register identifier associated to the subchannel indicated by the NACK value. Figure 12 shows the NACK signals for the considered example when the set-up request packet fails at the allocator LOCAL at router (2,3).

3.2.2. Set-up Path Phase in SDM-TDM-Based Router. Let us now consider an SDM-TDM Channel consisting of 3 subchannels and 3 time slots. The subchannel identifiers are 1, 2, and 3, respectively, and the time slots numbers are also 1, 2, and 3, respectively. We considerm as in the previous example, a set-up path phase between a source node attached to the router with coordinates (2,1) and a destination node attached to the router with coordinates (2,3). The set-up request packet from the source node is given in Figure 13; the fields' subchannel identifier, time slot number request, and payload are set to zero, respectively.

At the router (2,1), the allocator EAST reserves an available time slot in any available subchannel in the SDM-TDM Channel output port and indicates to the source the time slot from which to transfer streaming packet following the relationship $(T_i - 1)$ *modulo* S, where T_i is the reserved time slot and S the number of time slots. Let us suppose that time slot 1 is reserved at the subchannel

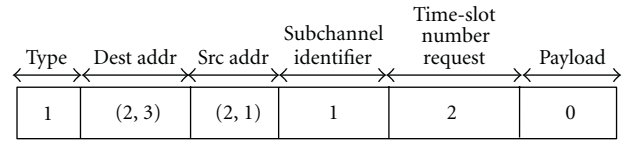

Figure 14: SDM-RDM outgoing set-up packet allocator EAST (2,1).

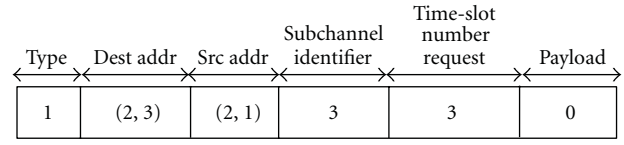

Figure 15: SEM-TDM outgoing set-up packet allocator EAST (2,2).

1, the allocator then builds the outgoing set-up request packet with the reserved subchannel identifier and the time slot to request in the next hop (Figure 14), therefore the time slot number 3 is the time slot at which the source node injects streaming packets in the network. The allocator concatenates the incoming subchannel with the outgoing subchannel in a register denoted "reg_identifier_time1". Since there are three time slots per sub-subchannel, we thereby define reg_identifier_time1, reg_identifier_time2, and reg_identifier_time3 for each subchannel. These registers allow easy retrieval of the subchannel where the specified time slot must be released if a NACK signal is received.

At the router (2,2), let us assume that time slot number 2 at subchannel 1 and time slot number 2 at subchannel 2 are already reserved by other set-up request packets; the allocator will then reserve the time slot number 2 at the subchannel 3. The outgoing set-up request packet is shown in Figure 15. It concatenates the incoming subchannel and the outgoing subchannel in the reg_identifier_time2 associated to the subchannel identifier 3.

At the router (2,3), the allocator LOCAL reserves the requested time slot at the unique subchannel; in this case, it is the time slot number 3. The ACK packet is then generated and routed through the packet-switched subrouter from the destination to the source. Upon reception of the ACK packet, the source node then starts transferring streaming data at the time slot specified by the allocator EAST at router (2,1). Figure 16 shows the established path and the scheduling of time slots.

When a set-up request packet failed to reserve time slot at any subchannel, the NACK signal is sent back and propagated to all previous packet-switched subrouters crossed by the failed set-up request packet. The NACK indicates the subchannel in which the specified time slot number has to be released. For illustration purposes, let us suppose for the considered example that the set-up request packet fails to reserve the time slot number 3 at the allocator LOCAL (2,3). The allocator LOCAL will then issue the NACK signal indicating to the router (2,2) to release time slot number 2 at the subchannel value contained in the incoming set-up packet, which is 3 in our case. At the router (2,2),

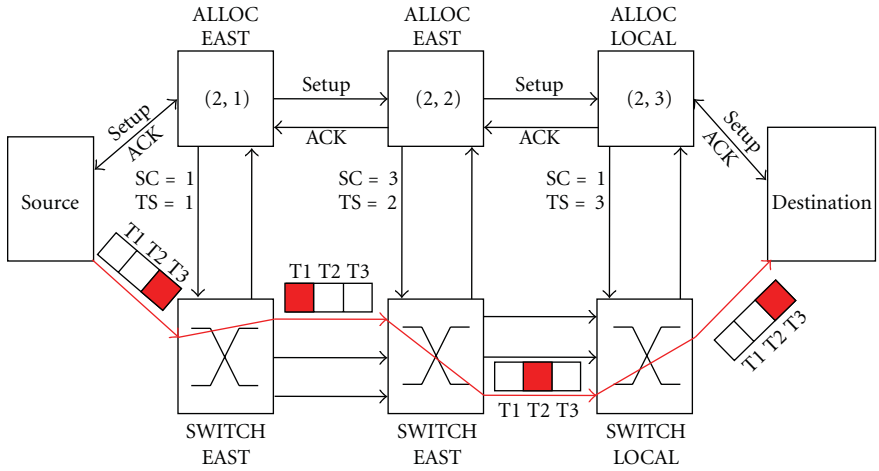

Figure 16: SDM-TDM path between source (2,1) and destination (2,3).

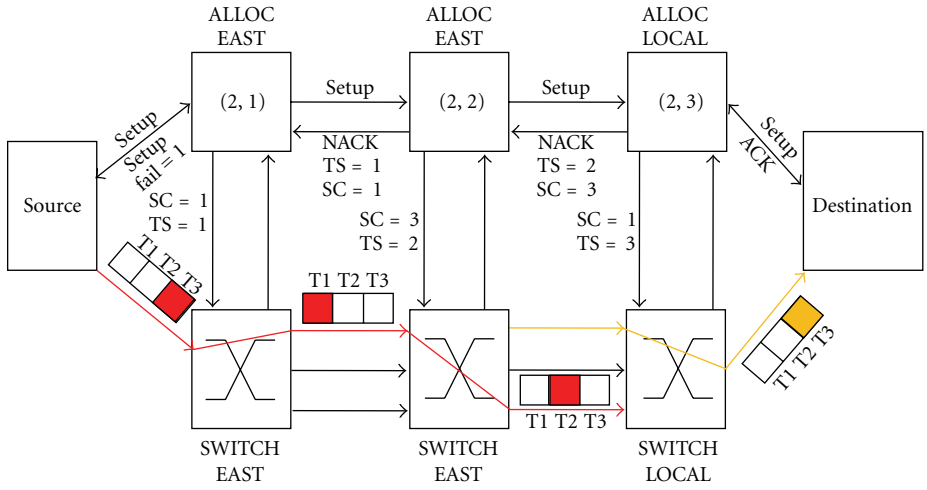

Figure 17: Failed path establishment in SDM-TDM-based NoC.

upon reception of the NACK, it releases specified resources and computes the NACK to send back to the router using the MSB of the reg_identifier_time2 of the subchannel 3. According to this register, the NACK to router (2,1) directs to release the time slot number 1 at the subchannel 1. At router (2,1), the allocator releases the reserved resources and notifies the source node that the set-up request packet failed. This process is shown in Figure 17.

3.3. SDM-Based Circuit-Switched Subrouter. The SDM-based circuit-switched subrouter is responsible for carrying streaming traffic. It has five bidirectional ports. Four bidirectional ports are SDM-based and are used to connect the circuit-switched subrouter to the four adjacent circuit-switched subrouters, and the fifth bidirectional port, which consists of a subchannel, is used to connect the SDM-based circuit-switched subrouter to the local tile as shown in Figure 18. This port is a subchannel since we assume that the local tile cannot receive more than one packet simultaneously. The SDM-Channel consists of a given number of subchannels. Each subchannel is N-bit wide. The streaming traffic is organized in packets like cells in ATM networks. The streaming packet format is shown in Figure 19.

The header indicates the validity of the carried payload. A header value 1 indicates that the carried payload is valid, and a header value 2 indicates that the payload is not valid. The header is used in order to release or maintain reserved resources. When the value of the header is zero, no action is taken. When this value is 1, it means that the transfer of the streaming traffic is ongoing. When a header value 2 is detected, a signal is sent to the attached packet-switched allocator to release the reserved resources. The SDM-based circuit-switched subrouter consists of five switches and header detectors. The switch consists of multiplexers. Since switches are configured by the packet-switched allocators, the use of an XY deterministic routing algorithm in the packet-switched subrouter which prevents best-effort packets to return in the direction where they come from determines the number of input ports of each switch.

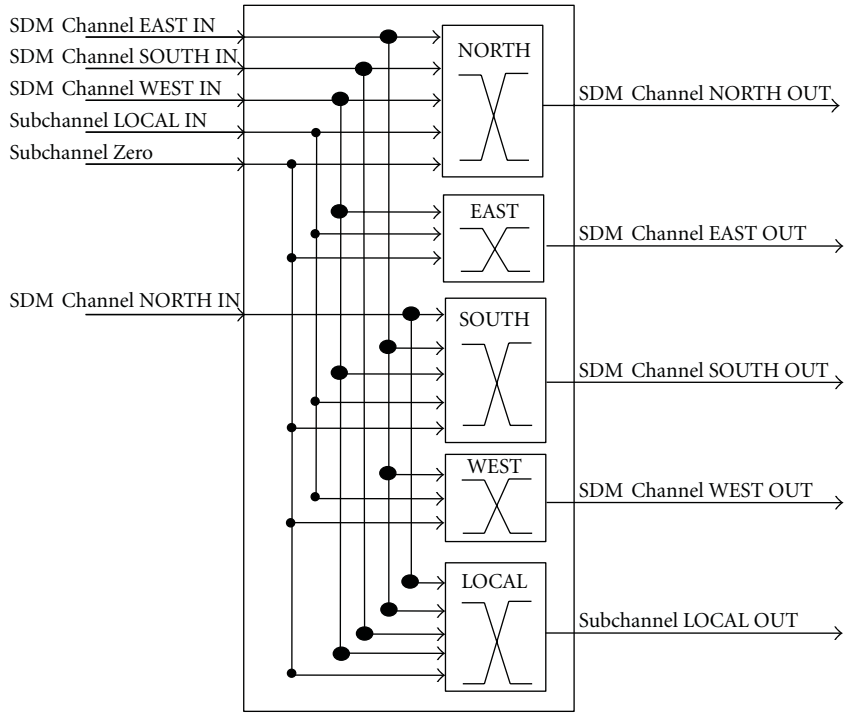

Figure 18: SDM-based circuit-switched subrouter.

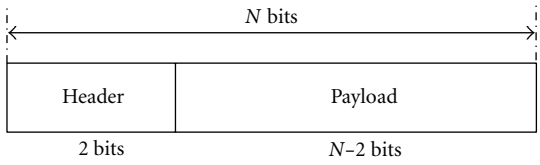

Figure 19: Streaming packet format.

In *XY* deterministic routing algorithm, a packet is routed first in the *X* dimension until it reaches its *X*-coordinate destination, then it begins to be routed in *Y* dimension until it reaches its *Y*-coordinate destination. Since we impose that packets coming from a given direction cannot return in the same direction, following the *XY* deterministic routing algorithm packets coming from EAST direction can only be routed either towards WEST, NORTH, SOUTH, or LOCAL directions, while packets from NORTH can only be routed either towards SOUTH or LOCAL directions. Thus, packets travelling in *X*-direction (From EAST or WEST) and packets from local tile can be routed in four possible directions, while packets travelling in *Y*-direction (From NORTH or SOUTH) can only be routed in two possible directions. This implies that the allocator in EAST direction (Figure 4) can only route packets coming from Input WEST and from input local. The switch attached to this allocator can then carry streaming packets either from the "SDM Channel WEST IN" or from the "subchannel LOCAL IN" as shown in Figure 18. The input port "Subchannel Zero" is used by default for all unreserved output subchannels.

For illustration purposes, let us consider the switch in direction EAST. According to Figure 18, it has three input ports; these are "SDM-Channel WEST IN", "subchannel LOCAL IN", and "Subchannel Zero". We consider an SDM-Channel consisting of 3 subchannels. The bloc diagram of such switch is shown in Figure 20, and its implementation is shown in Figure 21.

The signals "Sel1", "Sel2", and "Sel3" are provided by the allocator EAST of the attached packet-switched subrouter. The signals "Rel1", "Rel2", and "Rel3" are provided by the header extractor which is attached to the three subchannels.

The SDM-based circuit-switched subrouter is entirely combinational. Once a path is established, communication latency is only determined by the serialization time to send the entire streaming message. QoS is then easily provided. Latency and throughput can be configured by inserting pipelines between circuit-switched subrouters. However, each reserved subchannel is only used by one connection; this limits the scalability of the proposed approach. We then combine SDM with TDM in order to share each subchannel among multiple connections.

3.4. SDM-TDM-Based Circuit-Switched Subrouter. The SDM-TDM based circuit-switched subrouter has the same configuration as the SDM-based circuit-switched subrouter, however it contains additional input registers which allow scheduling the streaming packets in their trip through the network as illustrated in Figure 16. The scheduling of time slots reservation ensures that streaming packets are injected in the network in such way that they do not

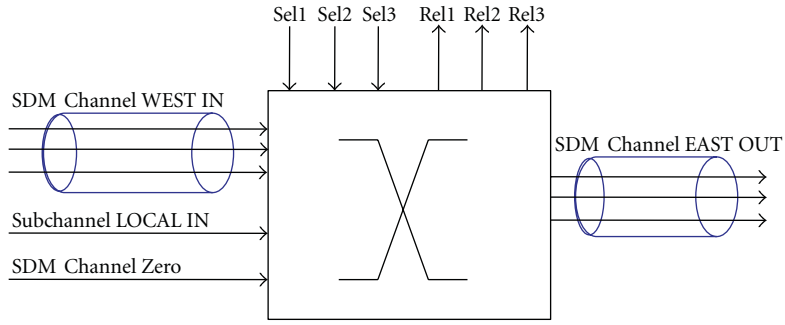

Figure 20: Block diagram SDM switch EAST.

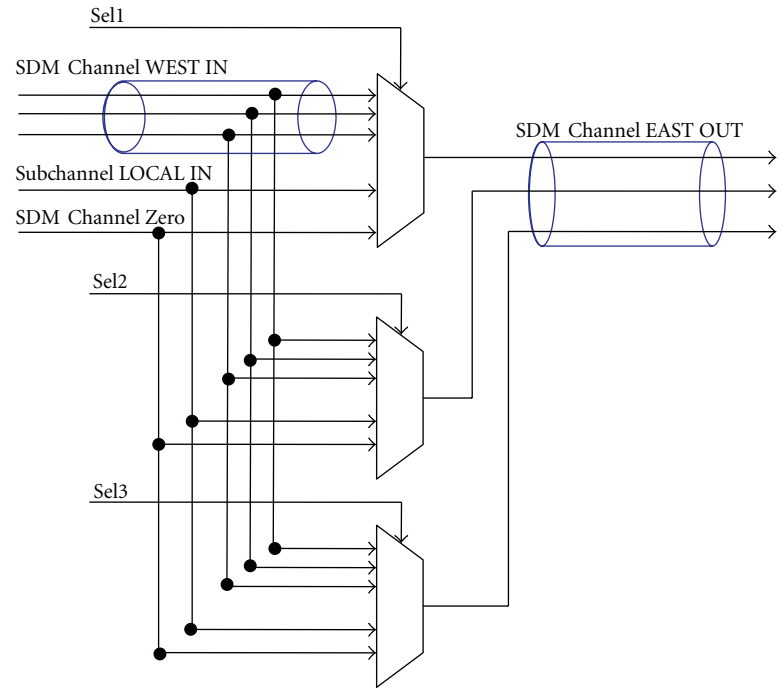

Figure 21: SDM switch EAST implementation.

collide. Figure 22 shows the bloc diagram of the SDM-TDM circuit-switched subrouter, and Figure 23 shows the bloc diagram of the switch in direction NORTH and the attached packet-switched allocator.

4. Results

4.1. Simulation Results. The proposed hybrid routers were implemented in SystemC RTL. 7×7 2D mesh NoCs were built and simulated in SystemC under synthetic traffic. We evaluate the performance of the NoCs in terms of number of simultaneous established connections (paths) through the network when all tiles in the network attempt to establish a path in the network. This worst case scenario leads to a high level of contention to occur in the network. The number of established paths in the network reflects the capacity of the network to face congestion. The fraction of set-up request packets which reach their destination reflects the probability of establishing a path in the network. A higher probability of establishing paths implies a higher number of applications to be run simultaneously in the network, thereby significantly improving the performances of the applications.

Three NoC platforms are compared; the SDM-based hybrid NoC, the SDM-TDM-based hybrid NoC, and the TDM-based hybrid NoC. These platforms are evaluated with the same traffic pattern in order to objectively compare them in terms of established paths according to the number of set-up request packets sent through the network. The destination nodes are generated using a uniform distribution. These simulations were performed with 4-packet deep FIFO buffer per input port for the packet-switched subrouter, while a different number of channels in an SDM-Channel, an SDM-TDM Channel, and TDM Channel are considered.

For the SDM-based NoC, the number of established paths through the network for 3, 4, and 5 subchannels in

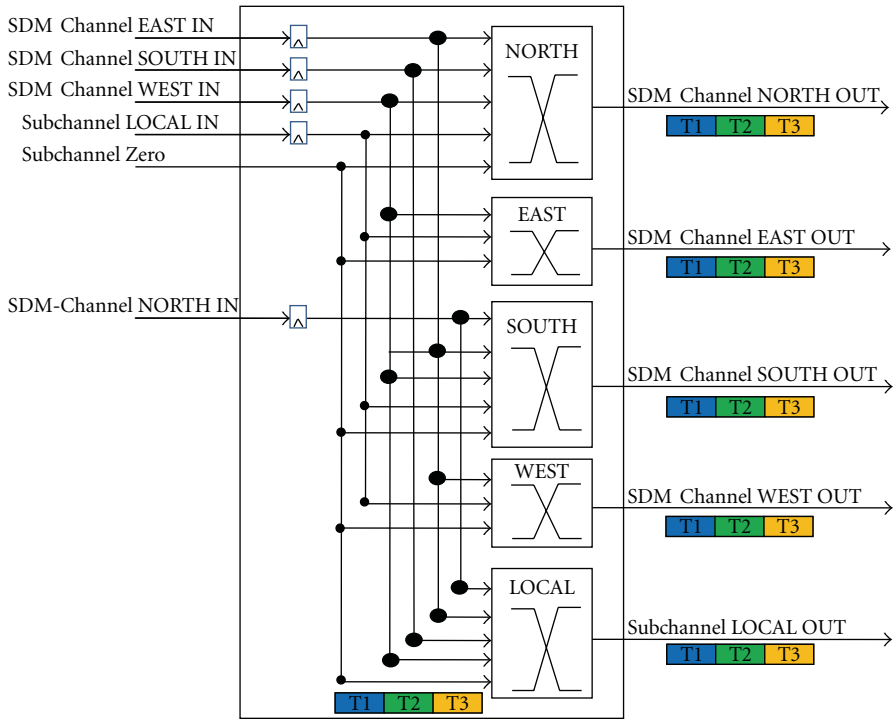

Figure 22: SDM-TDM circuit-switched subrouter.

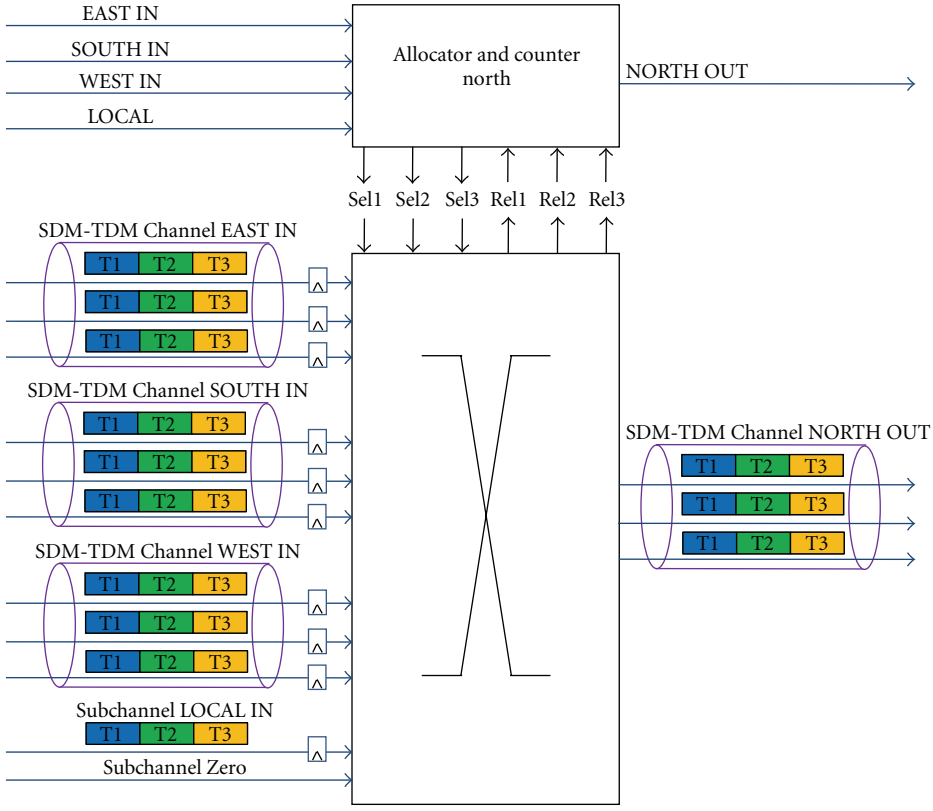

Figure 23: Block diagram SDM-TDM Switch NORTH.

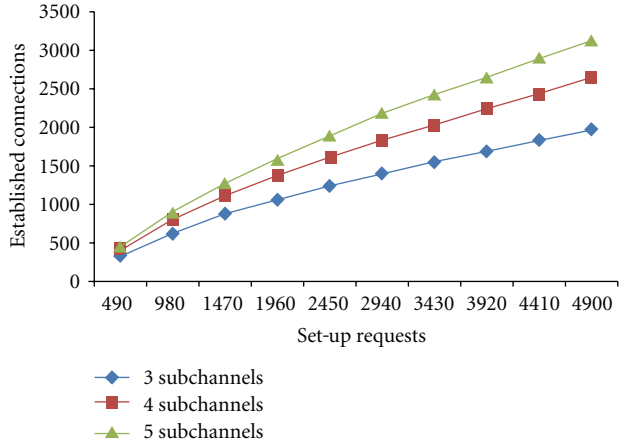

Figure 24: Established connections in 7 × 7 SDM-NoC.

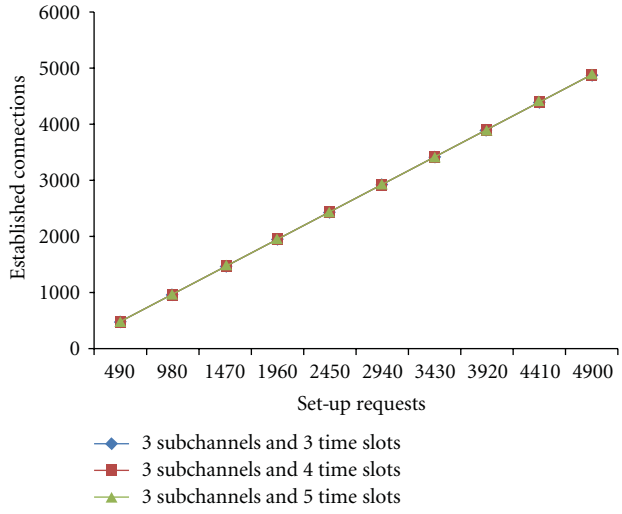

Figure 25: Established connections in 7 × 7 SDM-TDM NoC.

an SDM-Channel is given in Figure 24. For the SDM-TDM-based NoC, the number of established connections for 3 subchannels and 3, 4, and 5 time slots is given in Figure 25. Figure 26 gives the number of established connections in an TDM-based NoC for 3, 4, and 5 time slots in a channel.

For the SDM-based NoC, simulation results show that the number of established connections increases with the number of subchannels in an SDM-Channel. For 3 subchannels in an SDM-Channel, up to 46% of the set-up request packets sent in the network successfully reach their destination; for 4 and 5 subchannels in a SDM-Channel, up to 61% and 72% of set-up request packets sent in the network reach their destination, respectively.

For the SDM-TDM-based NoC, considering only 3 subchannels and 3, 4, and 5 time slots, Figure 25 shows that up to 98% of set-up request packets sent in the network successfully reach their destination in the three cases. For the TDM-based NoC, simulation results in Figure 26 show that for 3, 4, and 5 time slots up to 17%, 22%, and 27%, respectively, of the set-up request sent in the network

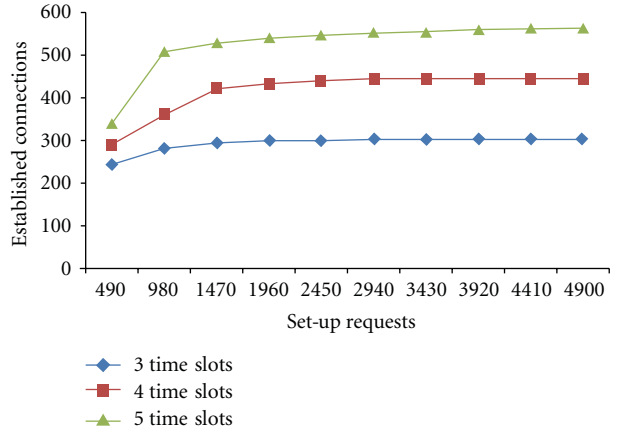

Figure 26: Established connections in 7 × 7 TDM NoC.

reach their destination. The poor performance of the TDM-based NoC related to the number of established paths is essentially due to the scheduling constraint on time slot reservation, which is a bottleneck for TDM-based NoC. Increasing the number of time slots does not efficiently solve the problem while increasing the size of the router. Since there is no constraint on resource reservation in the SDM-based NoC and by offering an increased path diversity, the SDM-based NoC has an appreciable probability of establishing connections through the network, however since each reserved resource is exclusively used by one connection until the end of the transaction, there is still a poor usage of subchannel, although overall the poor usage of resource is mitigated by the number of subchannels in the SDM-Channel. The SDM-TDM NoC solves this problem by allowing increased path diversity, while sharing subchannels among multiple connections, thereby performing the highest probability of path establishment in the network.

The ability of the proposed hybrid routers to handle best-effort traffic is evaluated by means of the average latency and average throughput according to the injection best-effort traffic rate. To evaluate the average latency for the best-effort traffic, we consider that 25 tiles are injecting best-effort traffic, while 24 tiles are transferring streaming traffic.

Figure 27 shows the average latency of the best-effort traffic for the three hybrid NoCs. The TDM-based NoC has the smallest average latency compared to the SDM- and SDM-TDM-based NoCs. This is due to the fact that the TDM-based NoC has the smallest probability of path establishment; it in results a small number of established paths, therefore a small number of ACK best-effort packets, thereby impacting weakly the total number of the best-effort packets in the network. Whereas the SDM-TDM-based NoC allows the highest number of established paths, it results in a higher number of ACK best-effort packets, which significantly impact the total number of best-effort packets in the network, thereby increasing the average latency. However, the three hybrid routers begin to saturate beyond an injection traffic rate of 0.1.

The average time to establish paths through the SDM-based NoC is reported in Table 1. As noticed previously, we

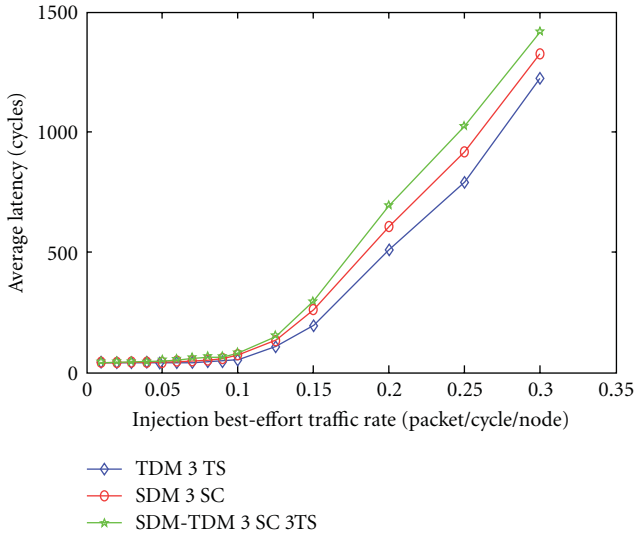

Figure 27: Average latency.

Table 1: Average time to set up a path in a 7×7 SDM-based NoC.

Injection best-effort traffic rate [Packet/Cycle/Node]	Average establishment path time in an SDM-based NoC [Cycles] Number of subchannels		
	3	4	5
0.01	96.18	96.43	96.44
0.02	96.28	96.36	96.48
0.03	96.33	96.42	96.46
0.04	96.47	96.48	96.61
0.05	96.64	96.68	96.77
0.06	96.98	97.11	97.14
0.08	97.47	97.67	97.47
0.1	98.41	98.56	98.59

consider for this experiment that 25 tiles are transferring best-effort traffic, while 24 are transferring streaming traffic. Table 1 shows that the average time to establish paths, which is the average latency before a tile starts to transfer the streaming traffic, is not greatly impacted by the best-effort traffic load. By imposing a minimum Manhattan distance of 5 hops between a given pair of source and destination, the average time to establish paths through an SDM-based NoC is around 96 cycles and does not depend on the number of subchannels in an SDM-Channel.

4.2. Synthesis Results. The proposed hybrid router architectures have also been implemented in Verilog HDL and synthesized in FPGA from Altera. For the SDM- and the SDM-TDM-based routers, the packet-switched subrouter has a 4-packet deep FIFO buffer per input port. For the SDM-based router, the packet-switched ports are 25-bit wide and the subchannels in SDM-Channels are 18-bit wide (2 control bits + 16 bits payload).

Table 2: Synthesis results SDM router (18 bits per subchannel).

SDM Number of channels	STRATIX III EP3SL340F	
	Total logic element utilization	Frequency [MHz]
3 subchannels	4590	200
4 subchannels	5307	190
5 subchannels	6432	186

Table 3: Synthesis results SDM router (10 bits per subchannel).

SDM Number of channels	STRATIX III EP3SL340F	
	Total logic element utilization	Frequency [MHz]
3 subchannels	3867	200
4 subchannels	4648	190
5 subchannels	5392	186

Table 4: Synthesis results TDM router.

TDM Number of channels	STRATIX III EP3SL340F	
	Total logic element utilization	Frequency [MHz]
3 time slots	5934	152
4 time slots	6381	123
5 time slots	6895	113

Table 5: Synthesis results SDM-TDM router (3 SC, 3, 4, 5 TS).

SDM-TDM Number of channels	STRATIX III EP3SL340F	
	Total logic element utilization	Frequency [MHz]
3 subchannels and 3 time slots	6057	160
3 subchannels and 4 time slots	7013	150
3 subchannels and 5 time slots	7841	145

Table 6: Synthesis results SDM-TDM router (3 TC, 3, 4, 5 SC).

SDM-TDM Number of channels	STRATIX III EP3SL340F	
	Total logic element utilization	Frequency [MHz]
3 subchannels and 3 time slots	6057	160
4 subchannels and 3 time slots	7652	159
5 subchannels and 3 time slots	9194	158

Table 2 reports synthesis results for 3, 4, and 5 subchannels in an SDM-Channel. Since the packet-switched and the circuit-switched subrouters separately handle traffic, they can be designed and optimized separately. Thus, Table 3 shows how reducing the width of subchannels from 18 bits to 10 bits impacts the overall size of the router.

For the TDM-based router, the packet-switched router has an 8-packet deep FIFO per input port, while the circuit-switched has a 4-packet deep FIFO buffer per input port. The ports are 25-bit wide. Synthesis results for the TDM-based router are reported in Table 4 for 3, 4, and 5 time

Table 7: Synthesis results 4 × 4 SDM-based NoC.

Number of channels	4 × 4 SDM NoC				
Subchannels	Combinational	Combinational with register	Registers	Total logic utilization	Frequency
3	19564	29499	18696	67759/270400 (25%)	124 MHz
4	24953	33032	18978	76963/270400 (28%)	100 MHz
5	30001	36473	19036	85531/270400 (32%)	82 MHz

Table 8: Synthesis results 4 × 4 SDM-TDM based NoC.

Number of channels		4 × 4 SDM-TDM NoC				
Subchannels	Time Slots	Combinational	Combinational with register	Registers	Total logic utilization	Frequency
3	3	24383	38134	22771	85288/270400 (32%)	85 MHz
4	3	32996	47811	22862	103669/270400 (38%)	78 MHz
5	3	37362	55222	22942	111526/270400 (43%)	70 MHz

slots, respectively. For the SDM-TDM-based router, synthesis results are reported for a various number of subchannels and time. Packet-switched ports are 28-bit wide, and the subchannels are 18-bit wide. Synthesis results are reported in Tables 5 and 6.

Results from Tables 2, 4, 5, and 6 show that for a given number of channels (subchannels and time slots) the SDM-based router has better performances in terms of maximum clock frequency and the total logic used in the FPGA compared to the two other hybrid routers. This is due to its simplicity, since the critical path is confined in the packet-switched subrouter, while the circuit-switched subrouter is entirely combinational. Increasing by one, the number of the subchannels in an SDM-Channel results in the increase of 16% in the router size, while the clock frequency is slightly reduced.

Furthermore, results from Table 2 show the impact of reducing the width of the subchannels from 18 bits to 10 bits on the size of the router, while the clock frequency is not impacted. Thus, optimization for a high clock frequency concerns only the packet-switched subrouter, while optimization of the size of the router concerns essentially the circuit-switched subrouter.

For the TDM-based hybrid router, Table 4 shows that it has the smallest clock frequency compared to the two other hybrid routers; this is essentially due to the fact that channels are shared between the two subrouters and the use of buffers in both subrouters, which increases the complexity of the router, thereby lengthening the critical path and increasing the size of the router. Increasing by one, the number of time slots leads to an increase of 7% in the router size, while the clock frequency is significantly reduced.

However, the size of the TDM-based router grows slower than the sizes of the SDM- and SDM-TDM-based routers when increasing the number of the channels by one. It means that, compared to the SDM-based router, there is a subchannels number threshold from which the size of the SDM-based router becomes greater than the size of the TDM-based router.

For the SDM-TDM based subrouter, it has the highest overhead in the total logic used in the FPGA; this is the cost of combining the two techniques in a single router. However, it offers appreciable clock frequencies compared to the TDM-based router, since the packet-switched and the circuit-switched subrouters do not share the same channels and independently handle traffic. Furthermore, the circuit-switched subrouter has just simple registers instead of FIFO buffers. This eases the control of the critical path in the design of the router. Optimization can be done separately in order to either reduce the size or increase the clock frequency of the router. The SDM-TDM approach gives more flexibility since it allows optimization in either space or time. Table 5 shows that keeping the number of subchannel fixed while increasing the number of the time slots implies an increase by 16% in the size of the router, whereas results from Table 6 show that maintaining constant the number of time slots while increasing the number of the subchannels leads to an increase of 20% in the size of the router, while the clock frequency remains practically constant.

Thus, there is a tradeoff between an optimal number of subchannels and the number of time slots according to the constraint on the clock frequency and the area of the router.

The proposed hybrid routers were used to build complete 4 × 4 2D mesh NoC on the Stratix III EP3SL340F FPGA device. Synthesis results for the SDM-based NoC, SDM-TDM-based NoC, and TDM-based NoC are shown in Tables 7, 8, and 9, respectively. These results show the impact of the interconnecting links on the frequency and area.

The impact of the interconnecting links on the frequency and the area of the TDM-based NoC is small compared to the SDM- and SDM-TDM-based NoCs. This is essentially due to the fact that links in the TDM-based router are shared between the best-effort and the streaming traffic, thereby reducing the link overhead between routers. However, The SDM-based NoC and the SDM-TDM based NoC, although the interconnecting link overhead, take advantages of the abundance of wires resulting from the high-level integration of CMOS circuits. The impact of interconnecting link can be mitigated by reducing the width of subchannels as shown in Tables 2 and 3, thereby reducing the area of the complete NoC.

Table 9: Synthesis results 4 × 4 TDM-based NoC.

Number of channels	4 × 4 TDM NoC				
Time slots	Combinational	Combinational with register	Registers	Total logic utilization	Frequency
3	18733	38540	29961	87234/270400 (32%)	80 MHz
4	20604	40040	30781	91425/270400 (34%)	72 MHz
5	21881	45031	31776	98688/270400 (37%)	66 MHz

For the total logic utilization in the FPGA, the SDM-TDM-based NoC has the highest percentage of resource utilization, while the SDM-based NoC has the smallest percentage of resource utilization. The total logic utilization of the NoC is not directly proportional to the router size, since for the 4 × 4 2D mesh, only 4 routers, located in the center of the mesh, are fully connected, while the routers at the edges have port in either one or two directions that are not connected. These unconnected ports are removed, thereby reducing the size of these routers.

5. Conclusions

In this paper, a hybrid router architecture which combines an SDM-based circuit switching with packet switching for on-chip networks is proposed. Since real-time applications can generate both streaming and best-effort traffic, instead of handling both traffics in a complex packet-switched or circuit-switched router, we propose to separately and efficiently handle each type of traffic in a suitable subrouter. The SDM-based circuit-switched subrouter is responsible for handling streaming traffic, while a packet-switched subrouter is responsible for handling the best-effort traffic. Handling the streaming traffic in a circuit-switched subrouter, QoS in terms of minimum throughput, and maximum latency is easily guaranteed.

The SDM approach used in the circuit-switched subrouter allows increased path diversity, improving thereby throughput while mitigating the low resources usage inherent to circuit switching. To improve usage of resources in the proposed router architecture, the SDM technique is combined with TDM technique, thereby allowing shared subchannels among multiple connections. The proposed hybrid router architectures were implemented in SystemC RTL and Verilog. 7 × 7 2D mesh NoCs were simulated in SystemC and compared to a TDM-based NoC. Simulation results show that increasing the number of subchannels in SDM-Channel or in an SDM-TDM Channel increases the probability of establishing connections in the network. Furthermore, by combining the SDM with the TDM, the NoC offers the highest probability of establishing paths through the network. Synthesis results on an FPGA show that increasing the number of subchannels in an SDM-channel has a slight overhead in router area, but does not greatly impact the maximum clock frequency compared to the TDM-based hybrid NoC. However, when SDM and TDM techniques are combined in a single router, the size of the router significantly increases according to the number of subchannels and time slots in an SDM-TDM Channel while reaching an appreciable clock frequency. Combining SDM and TDM in a single router offers more flexibility since optimization can be made either in space or in time. There is thus an opportunity to take advantage of partial dynamic reconfiguration in order to dynamically add additional subchannels or time slots in an SDM-TDM Channel in presence of heavy traffic and congestion.

References

[1] G. De Micheli and L. Benini, *Networks On Chips: Technology and Tools* , Morgan Kaufman, 2006.

[2] M. A. A. Faruque and J. Henkel, "QoS-supported on-chip communication for multi-processors," *International Journal of Parallel Programming*, vol. 36, no. 1, pp. 114–139, 2008.

[3] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *Journal of Systems Architecture*, vol. 50, no. 2-3, pp. 105–128, 2004.

[4] N. Kavaldjiev, G. J. M. Smit, P. G. Jansen, and P. T. Wolkotte, "A virtual channel network-on-chip for GT and BE traffic," in *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, pp. 211–216, March 2006.

[5] K. Goossens, J. Dielissen, and A. Radulescu, "ÆTHEREAL network-on-chip concepts," *IEEE Design and Test of computers*, vol. 22, no. 5, pp. 414–421, 2005.

[6] S. Bourduas and Z. Zilic, "A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing," in *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS '07)*, pp. 195–202, May 2007.

[7] N. E. Jerger, M. Lipasti, and L. S. Peh, "Circuit-switched coherence," *IEEE Computer Architecture Letters*, vol. 6, no. 1, pp. 5–8, 2007.

[8] M. Modarressi, H. Sarbazi-Azad, and M. Arjomand, "A hybrid packet-circuit switched on-chip network based on SDM," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 566–569, April 2009.

[9] A. Leroy, P. Marchal, A. Shickova, F. Catthoor, F. Robert, and D. Verkest, "Spatial division multiplexing: a novel approach for guaranteed throughput on NoCs," in *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (CODES+ISSS '05)*, pp. 81–86, September 2005.

[10] A. K. Lusala and J.-D. Legat, "A hybrid router combining SDM-based circuit switching with packet switching for On-Chip networks," in *Proceedings of the International conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 340–345, Quintano Roo, Mexico, December 2010.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 850487, 11 pages
doi:10.1155/2012/850487

Research Article

A Fault Injection Analysis of Linux Operating on an FPGA-Embedded Platform

Joshua S. Monson, Mike Wirthlin, and Brad Hutchings

Department of Electrical and Computer Engineering, Brigham Young University, 459 Clyde Building, Provo, UT 84602, USA

Correspondence should be addressed to Joshua S. Monson, jsmonson@gmail.com

Received 1 May 2011; Revised 28 July 2011; Accepted 1 September 2011

Academic Editor: Claudia Feregrino

An FPGA-based Linux test-bed was constructed for the purpose of measuring its sensitivity to single-event upsets. The test-bed consists of two ML410 Xilinx development boards connected using a 124-pin custom connector board. The Design Under Test (DUT) consists of the "hard core" PowerPC, running the Linux OS and several peripherals implemented in "soft" (programmable) logic. Faults were injected via the Internal Configuration Access Port (ICAP). The experiments performed here demonstrate that the Linux-based system was sensitive to 199,584 or about 1.4 percent of all tested bits. Each sensitive bit in the bit-stream is mapped to the resource and user-module to which it configures. A density metric for comparing the reliability of modules within the system is presented. Using this density metric, we found that the most sensitive user module in the design was the PowerPC's direct connections to the DDR2 memory controller.

1. Introduction

Over the last decade, Linux Operating Systems (OSs) have been used on several space-based computing platforms. NASA, for example, sponsored the FlightLinux project which culminated by demonstrating a reliable Linux OS on the UoSat12 satellite [1]. The use of Linux on the UoSat12 provided enough compatibility with ground systems to allow the satellite to be accessible over the Internet. Compatibility with ground systems is only one of the many reasons to use Linux on space-based computing platforms.

Reliable hardware is essential for Linux to operate; however, integrated circuits (ICs) aboard space-based computing platforms are susceptible to failures known as Single-Event Upsets (SEUs). SEUs are random bit flips caused by high-energy particles that collide with the ICs. To ensure correct operation in the presence of radiation, ICs can be specially designed or "hardened." Unfortunately, radiation-hardened ICs are expensive and usually two or three silicon generations behind the state of the art [2]. These factors limit the use of radiation-hardened parts in space-based computing and leave engineers looking for alternatives.

Field Programmable Gate Arrays (FPGAs) are among the state-of-the-art components that are of interest in space-based computing. FPGAs are microchips that contain an array of logic and interconnect that can be programmed and reprogrammed to perform almost any digital function. FPGAs often replace application-specific integrated circuits (ASICs) in space-based computing because designing for FPGAs is faster and less expensive than designing for ASICs. Additionally, reprogrammability allows designers to remotely fix bugs that appear after the platform has launched. These features make FPGAs ideal for space-based platforms [3–5].

The function performed by reprogrammable FPGAs is defined by the values of memory cells on the device known as the bit-stream. Changing the values of the bits in the bit stream may modify the behavior of a logic circuit on an FPGA (until it is reconfigured). For example, a single change in the bit stream has the potential to change the contents of a Look-Up Table (LUT), connect nets together, or completely disconnect a net. In most ICs, SEUs will only corrupt data. In FPGAs, however, SEUs are able to affect data and the logic function performed.

To prevent SEUs from affecting the output of an FPGA design, mitigation techniques are used. Common methods of mitigating and detecting the effects of SEUs are bit-stream scrubbing [6], Triple Modular Redundancy [7] (TMR), Partial TMR (PTMR) [8], and duplicate with compare (DWC) [9]. TMR uses redundant circuits and majority voters to improve the reliability of FPGA designs. PTMR is a reduced form of TMR that has been shown (on small circuits, at least) to reduce area overhead with only a small reduction in reliability. In bit-stream scrubbing, the bit-stream of the FPGA is occasionally rewritten to prevent the accumulation of SEUs. DWC is a method of detecting SEUs by creating a copy of the circuit, comparing the results, and flagging differences when they occur.

In FPGA-Embedded Linux systems using "hard core" processors, important peripherals (such as the memory controller) are implemented in the reconfigurable fabric of the FPGA and are susceptible to SEUs. SEU-induced failures in these components have the potential of crashing the kernel. Understanding each peripheral's likelihood of causing a kernel failure due to an SEU aids in understanding the reliability of the system and in creating a more reliable system at the lowest cost. To gain this understanding, we constructed a test-bed that allows us to simulate SEUs in the FPGA fabric surrounding a "hard core" embedded processor running a Linux kernel using a process known as fault injection [10]. Preliminary results, along with a fault injection analysis, from this test-bed have been previously reported [11]. In this paper, we present new results from a fault injection test using the same design under test. For this new test, we have improved both our fault injection process and analysis. These changes have resulted in a 3X improvement in detected sensitive bits and a decrease (from 8% to less than 1%) in sensitive bits that could not be mapped to part of the design.

2. Previous Work

The development of fast, comprehensive fault injection systems has been the focus of much of the previous work [12–14]. These fault injection systems are able to emulate upsets in FPGA fabric and microprocessor cache lines and special registers. Rather than performing exhaustive tests, these fault injection systems use a probabilistic model to statistically determine the reliability of a design.

The work presented by Sterpone and Violante [15] performed fault injection experiments on the memory image of a Linux microkernel running on a Xilinx Microblaze processor implemented in soft-logic. Their work focused on memory faults during the bootstrapping process but did not examine the sensitivity of the circuitry/logic that implements their system. Essentially, their fault-injection process modified the content of the memories that contain the Linux program. In contrast, our effort injects faults directly into the hardware implementation and analyzes the sensitivity of a full Linux kernel system (rather than a micro kernel) to circuit and logic failures that may be caused by SEUs.

In another work by Johnson et al. [16], they describe the validation of a fault injection simulator using a proton accelerator. Their analysis used the bit-stream offsets of sensitive bits to compute the row and column locations of the sensitive bits. Doing this allowed them to create a "map" of the FPGA showing the locations of sensitive bits. Their fault injection simulator was able to predict the locations of an impressive 97% of the upsets caused by the proton accelerator.

In this work, we also endeavor to improve the verification of fault injection experiments. For our fault injection experiment, we studied the bit-stream to determine the relationships between configuration bits and FPGA resources; this allowed us to take the next logical step and map sensitive bits to FPGA resources and, further, map resources to user-design modules.

Sterpone and Violante also used knowledge of the configuration bit-stream, but rather than examining what happened (as we do) they tried to predict where faults would occur. In [17, 18], they present a reliability analysis tool that would perform a static analysis to predict locations of sensitive bits then they would perform fault injection based on their predictions. They found that their static analyzer could predict the locations of all the sensitive configuration bits in a design mitigated using TMR without being overly pessimistic. They also found that their partial fault injection results matched that of an exhaustive fault injection test.

The similarity between our work and their work is the reliance on architectural and configuration bit-stream knowledge to identify where the problems might occur. A key difference is that their work focused on verifying relatively small systems mitigated by TMR, while our work focuses on the reliability analysis of large, unmitigated FPGA designs. It is likely that their techniques could be used to analyze this Linux system but unfortunately, their system is not available to us and direct comparisons between the two approaches cannot be made.

3. System Architecture

The process of fault injection requires a development board with the ability to feed test vectors into a design under test, monitor the important outputs, and modify the configuration memory of the design. The ideal development board to perform these functions would contain three FPGAs, one to act as the golden, one to act as the device under test, and one to act as the experiment controller. Should the output of the golden be known or simple, a two-FPGA solution would be sufficient. These FPGAs should be connected such that the experiment controller is able to access the configuration memory of the device under test and send (receive) input (output) vectors from both the golden and design under test. Unfortunately, there are very few boards that provide the required connectivity. This is the primary reason we have developed our own fault injection platform.

Our two-FPGA fault injection system consists of a Linux-based host PC, two Xilinx ML410 development boards, and a 124-pin custom connector board. In our system, one ML410 acts as the experiment controller and golden and the other acts as the Device Under Test (DUT). The connector board provides enough connectivity for the controller to

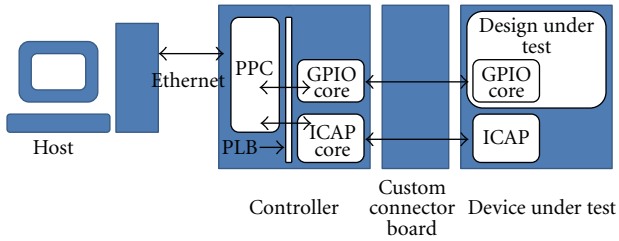

Figure 1: A block diagram of our fault injection system.

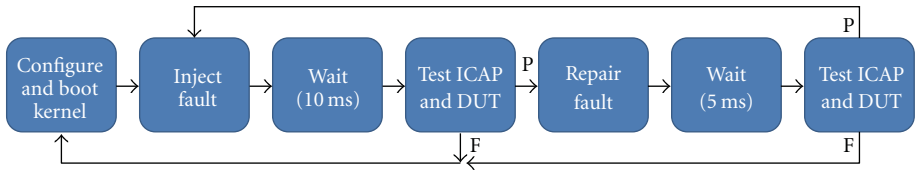

Figure 2: Flowchart of the fault injection routine.

send (receive) input (output) vectors from the DUT and also provides enough pins to fully connect the experiment controller to the Internal Configuration Access Port (ICAP) of the DUT. The ICAP is a module that resides inside the FPGA fabric and provides connections that allow components inside the FPGA to read and write the configuration memory.

As shown in Figure 1, the Linux-based host provides a simplified interface to the fault injection platform over an Ethernet connection. This allows the host to maintain an NFS mounted file system that provides a simple means for transferring software, test vectors, and results to and from the experiment controller. This connection allows experiment software to be developed on the host and then cross-compiled and transferred to the PowerPC 405. The RS-232 serial port acts as the console for the Linux kernel.

All DUT test designs communicate directly with the experiment controller via routing (contained in the DUT test design) that connects the DUT's ICAP to the custom connector board. The experiment controller injects faults through this 400 MB/sec connection. In cases where an injected fault causes the DUT's ICAP interface to fail, the controller can completely reconfigure the DUT using the external configuration interface. The control and communication circuitry for the ICAP is implemented on the experiment controller board to maximize available circuitry in the DUT. The ICAP interface on the experimental-controller board is controlled and configured by software and supports a wide variety of experiments.

4. Description of Fault Injection Experiment

The goal of this fault injection experiment is to measure the sensitivity of an FPGA-based Linux system to upsets in the configuration memory. In this FPGA-embedded system, all inputs and outputs of the processor must travel through the FPGA fabric. For example, the clock, memory interface, STDIN and STDOUT, general purpose I/O, and reset circuitry are implemented in the FPGA fabric. Thus the processor is unable to operate independently of the FPGA fabric, and faults injected into these components may cause the processor or Linux kernel to fail.

In general, fault injection tests are performed by modifying the configuration memory of the FPGA and then testing or observing the design to determine if the modification has caused a failure. The configuration memory is modified by reading a frame (the smallest addressable portion of configuration memory), changing a bit within the frame, and writing the frame back into the device. If the modified configuration memory causes the DUT to fail, the bit is considered "sensitive", otherwise the bit is considered "not sensitive."

In this test, each configuration bit is upset for over 10^6 FPGA clock cycles. However, it is not possible to test each bit in all possible states of the processor, I/O, and operating system. It is possible that some configuration bits tested during fault injection may be tagged as nonsensitive but are in fact sensitive under certain system conditions. While we recognize that some sensitive configuration bits may be tagged as nonsensitive, this form of fault injection provides a good estimate of the "average" sensitivity behavior of the system under test. Previous results from similar fault injection experiments show that the vast majority of sensitive bits are easily detected (i.e., they have a high probability of detection) and the average sensitivity of a design is adequately represented in fault injection experiments [16].

This fault injection procedure is carried out by the controller as illustrated in Figure 2. Before starting the first test, the controller reconfigures the DUT and waits for the Linux kernel to boot and the test program to start. After the DUT has booted, the controller injects a single fault into the configuration memory of the DUT. After 10 ms, the DUT is tested five times over an interval of 1 ms before the fault is classified. If the DUT fails to respond to any of the tests, the bit is classified as sensitive and the DUT is reconfigured and rebooted. On the other hand, if the DUT successfully responds to all five tests, the bit is classified as not sensitive.

Table 1: Time breakout of a nonsensitive bit test.

Configure FPGA	4 sec
Boot Linux Kernel	16 sec
Inject fault	.226 ms
10 ms wait	12.5 ms
Test ICAP	.105 ms
Test DUT	17.9 ms
Repair fault	.120 ms
5 ms wait	8.9 ms
Test ICAP	.104 ms
Test DUT	17.9 ms
Avg. test time (non-sensitive bit)	59.7 ms

To prevent the accumulation of faults, the hardware and software must be returned to a correct state before beginning the next test. This can be done by performing a full FPGA reconfiguration, recopying the original Linux image into memory and rebooting the kernel. While this is the ideal approach, the process of reconfiguring, copying the Linux memory image, and booting the kernel takes 20 seconds. Since 98% of the more than 13 million configuration bits tested are not sensitive, rebooting the DUT after each bit would cause the test to be prohibitively long. The alternative to rebooting the DUT is to repair the fault and move onto the next test. This approach does not guarantee that the system state is error free and in fact it may be possible for the repair of a fault to induce an error. To ensure that the system is at least working correctly, we subject the DUT and ICAP to another set of tests after a 5 ms wait period. If a fault is found, it is attributed to the bit most recently tested and the entire DUT is rebooted.

Sensitive configuration bits are detected in the DUT using a simple two-phase hand shaking protocol over the general purpose I/O connections (GPIO) provided by the custom connector board. The controller initiates a test of the DUT by inverting its GPIO bit. The DUT responds successfully to the test by inverting its GPIO bit to match the controller's GPIO bit. The DUT implements this protocol with a small test program that runs in the background of the Linux kernel. The pseudo-code for this test program is shown in Listing 1. The test program accesses the GPIO module to see if the controller inverted its bit; if the controller has inverted its bit, the test program responds by inverting the DUT's GPIO bit to match. At the end of each iteration, the test program is supposed to sleep for 200 μs and wake up and repeat the process; however, time measurements of the test program reveal that the program really sleeps for close to 4 ms. Using the top program, we found that the Linux kernel spends 99% of its time in the idle state and 1% of its time for user and system processes.

Table 1 shows the average time for each phase of the test of a single non-sensitive bit. The total testing time of a non-sensitive bit is almost 60 ms with the majority of the time occupied receiving 5 test responses from the DUT. Note that the wait times are slightly longer than initially intended because of the use of the `usleep()` function.

```
dut_test_program ( )
{
     int   dut_gpio = 0;
     int   controller_gpio;
     init_gpio ( );
     while ( 1 )
     {
          controller_gpio = read_gpio ( );
          if( gpio_value != dut_gpio)
          {
             dut_gpio = controller_gpio;
             set_gpio (dut_gpio);
          }
          usleep (200);
     }
     close_gpio ( );
}
```

Listing 1: Pseudo-code of DUT test program.

To convince ourselves that the 5 and 10 ms wait times imposed after faults were injected were sufficient to detect the vast majority faults, we performed additional tests with longer wait times. A subset of the configuration memory (4 different configuration blocks) was tested with an additional 15 ms (2X) and 30 ms (3X) of wait time. Table 2 summarizes the number of sensitive configuration bits found in each test. As Table 2 shows, increasing the length of the test did not significantly increase the number of sensitive bits found. Additionally, we found 97.8% percent of the sensitive bits were identical for all three tests which suggests that the 10 ms wait time is adequate.

Although fault injection is a proven and effective way to emulate SEUs, there are a few FPGA components that fault injection cannot test. For example, reading and writing the block rams (BRAMs) via the ICAP may cause side effects that artificially increase the bit-sensitivity count. Some LUTs in the Virtex 4 architecture contain additional circuitry that allows them to be used as 16-bit shift registers (SRL16s) or distributed RAM memories (LUT RAMS). The dynamic memory elements of SRL16s and LUT RAMs are part of the configuration memory and can be corrupted during reads and writes. To prevent this, Xilinx has included the GLUTMASK as a configuration option in the Virtex 4 architecture. When set, the GLUTMASK prevents both configuration memory reads and writes from modifying the contents of SRL16s and LUT RAMs. IOB (input/output buffer) configuration blocks should also be avoided; these bits control FPGA I/0 pins and arbitrarily flipping these bits may cause board or device damage. All of these situations were taken into account during the design of our experiment.

5. Sensitive Bit Density Metric

When designing for FPGAs, the goal is often to squeeze as much performance out of the device as possible. Reliability

Table 2: Number of sensitive configuration bits detected in original and 2X and 3X wait time tests.

Block	Original test	2X wait times	3X wait times
1	495	494	494
2	478	456	462
3	494	494	494
4	577	579	569

requirements often make this goal difficult to achieve because of the area overheads of mitigation techniques such as TMR.

A way to reduce the area cost of reliability is to use lesser mitigation techniques such as PTMR or DWC. While the area costs of these methods are lower they may have other costs such as lower performance or reliability. In some systems, costs may also be reduced by applying different mitigation techniques to different modules in the design. How should an engineer decide which mitigation techniques should be applied to each module to make the most efficient use of the FPGA?

One method would be to perform a fault injection test and determine which modules contain the most sensitive bits and apply more aggressive mitigation techniques to the modules that contain the most sensitive bits.

While this method is simple and straightforward it may not lead to the most efficient use of area. For example, consider a fault injection test of a design containing a large module that is moderately resilient to injected faults and a small module that is unable to tolerate a single fault at any time. After the test, it may be found that the large module, because of its size, contained more sensitive configuration bits than the small module. However, because the large module masks a portion of its sensitive bits while the small module does not, redundancy techniques would be more efficiently applied to the small module even though it has fewer sensitive bits. By using more redundancy techniques on the smaller module, more sensitive bits will be mitigated per unit area than using the same mitigation technique on the larger module.

5.1. A New Metric. To deal with this issue, we present a metric that takes into account both the number of sensitive bits and size of the module. This metric is called the sensitive bit density metric. This metric was first introduced by us in [11] and was measured in sensitive bits per unmitigated resource. The resources considered were nets and instances. Unfortunately, nets can vary in length while instances can vary in configuration, thus using nets and instances obscures the size of the sensitive cross-section of configuration bits. In this paper, we have improved the accuracy of the sensitive bit density metric by measuring it in sensitive bits per configuration bit. This directly links the metric to the sensitive cross section of the design. Modules with a higher sensitive bit density metric will be more efficiently mitigated by redundancy techniques than modules with lower sensitivity bit metrics.

To calculate the sensitive bit density metric, we must know the number of sensitive configuration bits and the bit-area of each module in the design. Fault injection is used to estimate the number of sensitive configuration bits of each module in the design. Ideally, the bit-area of the each module is calculated by adding up the configuration bits that must have a specific value for the design to work properly. To make our estimate of design bit-area for modules in our Linux system, we have added up all the bits in used routing resources and the bits required to set components of slices that are explicitly used.

One might ask, why do not all of the bits actually demonstrate sensitivity during a test? It is because sensitivity is a dynamic phenomenon that depends, to some extent, on system behavior. In our Linux system, there are three issues that reduce the number of detections of sensitive bits: masking, hiding, and kernel resiliency.

5.2. Masking, Hiding, and Kernel Resiliency. Masking occurs when the current operating mode of the circuit does not allow the invalid signal to propagate and cause a system failure. A well-known form of masking is TMR. In TMR, majority voters mask the effects sensitive bits that reside in one of the three redundant copies of the circuit. In our system, many modules are addressed over the processor local bus (PLB). If these components are not used by the processor, they cannot cause a failure in the Linux kernel and are thus masked from causing a failure.

Hiding occurs when the SEU causes the value of the affected signal to be correct during the fault injection test. For example, consider a circuit with a low asserted reset signal and assume that the reset is asserted infrequently. Suppose an SEU strikes a portion of the routing causing an open which results in a "stuck at" 1 fault. This circuit will continue to operate correctly until a reset is required. During the time the circuit is operating correctly, we would say that this sensitive bit was hidden.

The kernel is said to be resilient to faults that are propagated into the processor but that do not cause a kernel failure. In our opinion this is most likely to occur when a fault appears in a data or instruction word before it is consumed by the processor. A data word may not effect the operation of the kernel at all, while a fault could appear in an unused field of an instruction word.

6. Analysis of Results

Our results will be presented using three different analyses. The general overview analyzes the overall number of sensitive bits and describes the reliability of our unmitigated Linux System. In the FPGA resource analysis, the sensitive bits analyzed in the general overview are mapped to the resources they control. In the user circuit analysis, the sensitive resources are mapped to the modules that contain them (memory controller, UART, etc.) and a metric for comparing the sensitivity of modules is explained.

These three analyses are interesting because they provide a designer with data that answers questions such as what is

Table 3: Device logic utilization.

Resource	Used	Available	Utilization
Slice registers	6,271	50,560	12%
4-input LUT	5,688	50,560	11%
Block RAMs	53	232	23%
SRL16 shift registers	281	50,560	.5%
LUT RAMs	0	50,560	0%

the general sensitivity of the DUT? What resources contain the most sensitive configuration bits? Which modules in the DUT are most sensitive to SEUs? Answers to these questions could aid in the directed application of mitigation techniques and possibly save design area or suggest FPGA architecture improvements.

In addition, these analyses allow those performing fault inject experiments to verify their results by confirming that sensitive bits actually have the ability to effect the design. This is done by mapping sensitive bits to actual resources in the FPGA and verifying that they belong to or can affect a component in the DUT.

Our analysis relied heavily upon the open source tool RapidSmith [19] developed at Brigham Young University. RapidSmith is a Java API that is able to both parse Xilinx design files and interface with part databases. In this project, RapidSmith was one of the key components that allowed us to do sensitivity analysis in both the logic and interconnect portion of the design.

6.1. General Overview Analysis. This subsection provides the reader with a general idea of the vulnerable cross section of our design. Specifically, it presents the design utilization, the bits in the bit-stream that were and were not tested, and the number of sensitive configuration bits found in the design.

Table 3 shows the device utilization of the DUT. While 23% of the BRAMs are instantiated less than half of them are actually used by the Linux kernel, making their effective utilization about 10%. The BRAMs not used by the Linux kernel are inserted by Xilinx's Embedded Development Kit (EDK) and are used to boot the PowerPC. Overall, the device utilization is between 10% and 12%.

Table 4 gives a summary of the bits that were and were not tested. The majority of these bits were BRAM and SRL16 bits. The number of sensitive configuration bits contributed by these resources was estimated as discussed in the next paragraph. We did not attempt to estimate the number of sensitive-masked bits or the IOB bits.

Table 5 presents the number of sensitive configuration bits found in the DUT and ICAP circuitry. The SRL16 bits were estimated by assuming that all content bits in the shift register would be sensitive. The same assumption was made for all BRAMs used by the Linux kernel (some were instantiated by EDK but not used by the kernel). The IOB bits were not tested, and no attempt was made to estimate the effect of their sensitivity on the device.

The Mean Time Between Failure (MTBF) is a common parameter used in reliability analysis. The equation for estimating the MTBF is presented in (1). The calculation requires that we know the configuration bit upset rate, λ_{bit}, and the number of sensitive configuration bits in the design, N_{bits}.

Table 4: Summary of bits tested/not tested.

Bits tested	13,757,308	**66**%
Bits not tested	7,140,228	**34**%
Mask file bits (not tested)	821,636	4%
IOB bits (not tested)	944,640	4%
BRAM content (not tested)	5,373,952	25%
Total bits	20,960,512	**100**%

Table 5: Summary of sensitive bits.

	Sensitive bits	SRL16 bits	BRAM content	Total
DUT	119,110	4,496	58,880	182,486
ICAP	1,329	0	0	1,329

The configuration bit upset rate is $\lambda_{\text{bit}} = 2.78 \times 10^{-7}$ upsets per day [20]. This failure rate is the same for commercial- and radiation-tolerant Virtex 4 devices. The difference between the commercial- and radiation-tolerant devices is the use of a thin epitaxial layer to remove single-event latch-up (SEL) and to significantly increase the total ionizing dose (TID) of the device.

N_{bits} is the number of sensitive configuration bits found in the design. To estimate this value, we use the number of sensitive configuration bits we found during our fault injection experiment, 119,110. Additionally, upsets in BRAM and SRL16 content will also cause failures in the DUT. Since these components were not tested we estimate them by assuming that all bits in these elements are sensitive. We choose this approach because it provides a slightly pessimistic lower bound on the MTBF. The actual measured faults plus the estimated BRAM and SRL16 content bits brings N_{bits} to 182,486.

Kernel failures can also be caused by upsets in the processor. Since our focus was on faults in the reconfigurable fabric, we did not perform any fault injection into the processor. Additionally, we do not know the number of state elements within the processor so we cannot form a pessimistic guess. Thus, our MTBF only indicates the average time between failures caused by the reconfigurable fabric. Using these numbers for λ_{bit} and N_{bits}, we calculate the MTBF of our system due to faults induced by the reconfigurable fabric to be 19.7 days while in the IDLE operating mode:

$$\begin{aligned} \text{MTBF} &= \frac{1}{\lambda} = \frac{1}{\lambda_{\text{bit}} \times N_{\text{bits}}} \\ &= \frac{1}{2.78 \times 10^{-7}\,\text{upsets/day} \times 182,486\,\text{bits}} \\ &= 19.7\ \text{days}. \end{aligned} \tag{1}$$

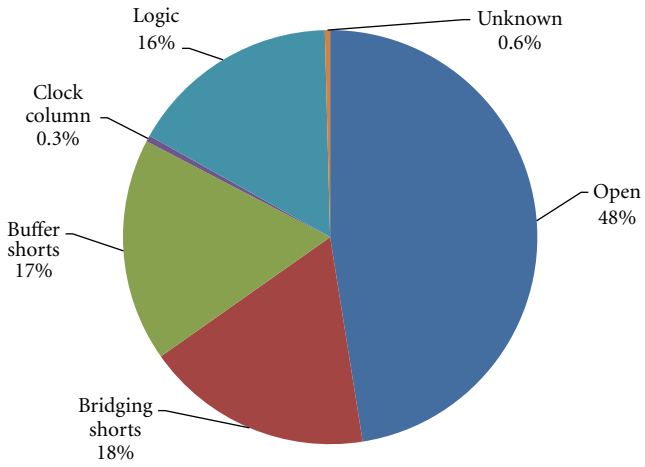

Figure 3: The distribution of sensitive configuration bits among utilized resources.

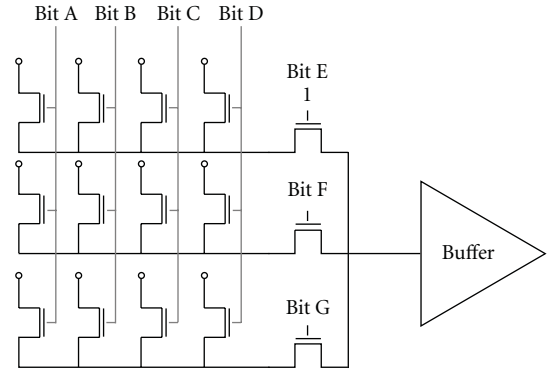

Figure 4: An example of a virtex 4 Routing Switch.

6.2. FPGA Resource Analysis. An FPGA resource analysis provides insight into which resources contained the sensitive configuration bits that cause system failure. This is done by mapping the sensitive bits analyzed in the previous section to the resources that they program. Understanding the resources involved in causing a design to fail could suggest improved resource-level mitigation approaches or future improvements to FPGA architectures.

Failures were placed into three sensitivity categories: Logic, Routing, and Unknown. The routing failures can be subdivided into three categories: open, bridging, and buffer failures. Figure 3 shows the distribution of the different types of failures identified in the design.

6.2.1. Logic Failures. Logic failures are the failures that occurred within the logic elements (slices) of the FPGA and accounted for 19% of all design failures. We were able to directly match most of the sensitive bits that caused these failures to components of slices that were specified as used in the design file. Some of the sensitive bits mapped to components of slices that were specified as unused in the design file; however, we often found that the unused (default) setting of the sensitive bit had relevance to the proper configuration of used slice components. For example, there is a mux at the input of the slice flip-flop set/reset line that allows the designer to choose between the inverted and noninverted version of the reset signal. When a flip-flop is instantiated in a design without a reset, an assumption is made by the bit stream generator about the default bitstream setting of the mux. If the bit that controls the mux is flipped by an SEU it will invert the default reset signal and hold the flip-flop in its reset state.

To compensate for this problem, we used the part database in RapidSmith to determine if an unused logic element's default setting could affect a used logic element. In about 10% of all logic sensitivities, an upset of the default setting was the cause of the failure.

6.2.2. Routing Failures. Routing failures accounted for 80% of all failure in the design. We now describe how routing failures occur, what their effects are, and how we identify them. To understand routing failures, one must first understand the structure of a programmable interconnect point (PIP) in the Virtex 4. While the structure of the Virtex 4 is not completely documented in the literature, the structure of a Virtex II pip is documented in [21, 22]. Our bit-stream studies lead us to believe a similar structure is used in the Virtex 4.

We conjecture that the Virtex 4 PIP (shown in Figure 4) is a two-level mux followed by a level restoring buffer. The select signals on this mux come from configuration bits A–G. Bits A–D are column bits while bits E–G are row bits. The small circles on the pass transistors represent connections from other wires in the switchbox. To properly pass a signal through the switch box, two configuration bits (a row bit and a column bit) must be set to "1" while all of the other bits must be set to "0".

For example, if we wanted to connect the wire coming into the top-left most pass transistor to the wire at the output of the PIP, we would set bits A and E to "1" and all the other bits to "0". Setting the other bits to "0" prevents contention within the PIP. By default, routing bits are set to "0". Since bits A and E are the bits that change from their default conditions we refer to them as primary bits.

Open failures are caused when an SEU sets one of the primary bits to a "0". This results in the disconnection of the input wire from the output wire. Since the PIP is now undriven the output will be driven high by the level restoring buffer.

To describe bridging and buffer failures, we have simplified the PIP in Figure 4 from a 3×4 PIP to the 2×2 PIP in Figures 5–8, where A and C are the primary bits in each PIP and net1 was originally passed through the PIP. Figure 5 shows the simplified PIP under normal (non-SEU) conditions. A lightning bolt on one of the configuration bits indicates that the bit has been changed to its current value by an SEU. The inside of the circles tells whether a buffer or net is driving the connection. The circle on the output describes a common function performed by the PIPs in the given configuration. Please note that these are common functions, not necessarily the function performed by every PIP in the given configuration. In fact, our data suggests that some invalid configurations of PIPs result in no failures at all.

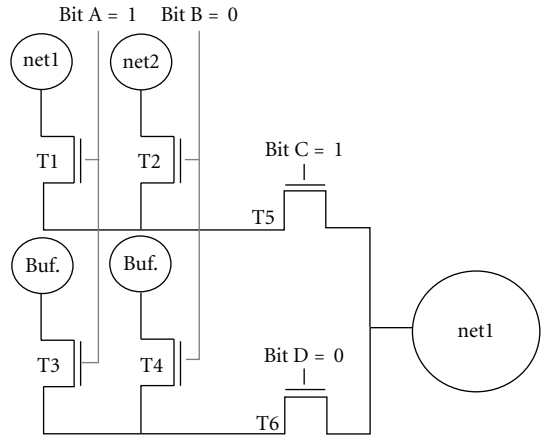

Figure 5: A simplified pip under normal operating conditions. No bits affected by SEUs.

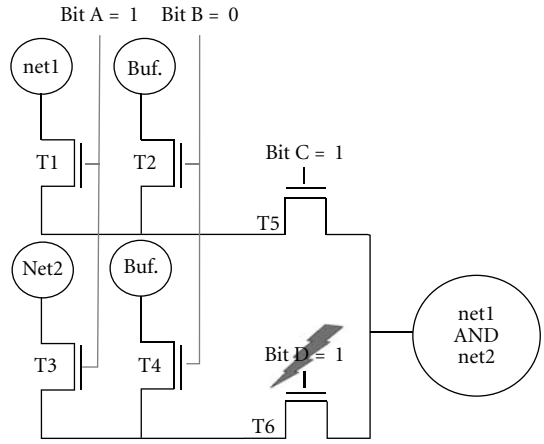

Figure 6: Example of a row bridging short. Bit D was affected by an SEU.

Figure 6 demonstrates a row bridging failure. The primary bit A allows net1 and net2 to drive the inputs of transistors T5 and T6, respectively. Since bit D was affected by an SEU, both row bits are active allowing both net1 and net2 to drive the output of the PIP causing a row bridging failure. Figure 7 demonstrates a column-bridging failure. In this case, net2 is driving the input of transistor T2, the SEU has effected bit B which allows both net1 and net2 to drive the input of T5 causing a column bridging short.

Figures 8 and 9 are examples of buffer failures and are exactly the same as row and column bridging failures with the exception that a buffer rather than a net is driving the connection created by the SEU.

In general, row-based buffer and bridging failures exhibit ANDing behavior. For a bridging failure this means that net1 and net2 are ANDed together. For a buffer failure this means that net1 is ANDed with a level restoring buffer, fortunately, this reduces to net1 being ANDed with a constant "1", which produces no effect on net1. While this ANDing behavior reduces the amount of row buffer failures, it does not always eliminate them. The majority of all row buffer failures

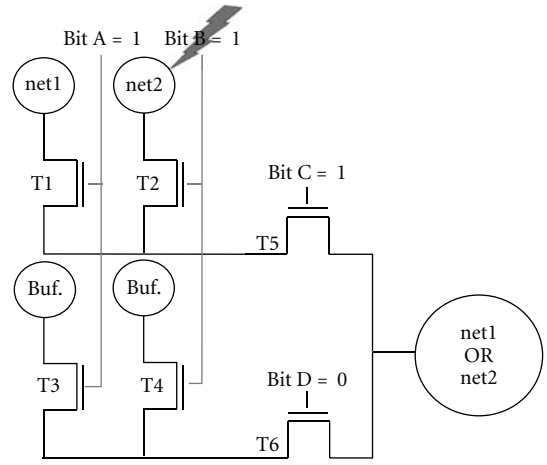

Figure 7: Example of a column bridging short. Bit B was affected by an SEU.

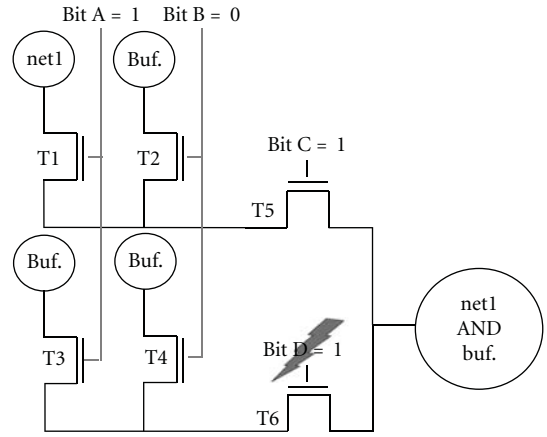

Figure 8: Example of a row buffer short. Bit D was affected by an SEU.

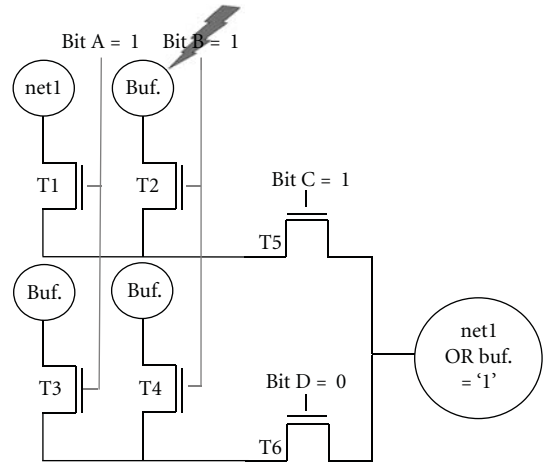

Figure 9: Example of a column buffer short. Bit B was affected by an SEU.

Table 6: Comparison of components.

Module name	Primary bits	Interconnect bits	Logic bits	Bit-Area	# of sens.	Density metric
ICAP	987	3,827	23	3,850	1,329	0.34519482
IPLB	2,439	11,035	2,276	13,311	3,641	0.27353317
DPLB	2,417	11,025	2,163	13,188	2,781	0.21087351
PPC	10,030	43,188	6,219	49,407	8,957	0.1812901
DDR2	122,228	554,433	113,114	667,547	91,398	0.1369162
GPIO	917	4,522	1,688	6,210	850	0.136876
SYSTEMRESET	1,503	6,842	1,370	8,212	728	0.088650756
CLOCKGEN	11,966	48,194	562	48,756	4,145	0.08501518
PLB	18,121	78,556	10,864	89,420	3,449	0.038570788
INTC	2,236	10,642	2,932	13,574	365	0.026889643
IIC	11,558	53,990	13,562	67,552	927	0.0137227615
UART	2,633	12,563	3,425	15,988	203	0.012697022
SystemAce	3,093	14,348	3,808	18,156	113	0.006223838
BRAMCNTRL	11,525	50,616	4,099	54,715	25	4.5691308E-4
JTAGDEBUG	1,728	7,737	0	7,737	0	0.0
MGTPROTECTOR	13,504	64,784	0	64,784	0	0.0

in our fault injection test occurred in PIPs that are driven by the outputs of the slices. This is expected because the default settings of some logic elements in the slice is "0". Instead of the net being ANDed with a constant "1" which has no effect, the net is ANDed with a "0" which is always "0".

Column-based buffer and bridging failures typically result in ORing behavior. For a bridging failure, this means that net1 and net2 are ORed together. For a buffer failure, this means that net1 is ORed with a constant "1" which results in a constant "1" output for the PIP. In our fault injection test, column-based failures occurred twice as often as row-based failures for both bridging and buffer failures.

Once we understand the structure of PIPs and the ways that failures can occur, identifying these failures from fault injection results is straight forward. If the sensitive bit is a primary bit, then we know an open failure has occurred. If the sensitive bit is not a primary bit, the driver (buffer or net) of the connection created by the SEU will classify the failure as a buffer or bridging failure. We can further classify buffer and bridging as row or column failures by identifying whether the sensitive bit is a row bit or column bit.

Unknown failures accounted for less than 1% of all failures in the design. These were failures for which the sensitive bits that caused the failures could not be mapped to part of the design. Even though we do not know why these sensitive bits have caused design failure, we are confident in our fault injection results due to the fact that we were able to diagnose over 99% of the sensitive configuration bits that were found.

6.3. User-Circuit Reliability Analysis. The user-circuit reliability was analyzed on a module by module basis. In this analysis, the sensitive resources identified in the FPGA resource analysis are mapped to the modules that contain them. This information can be used to make decisions on the amount of redundancy used to increase the reliability of a system on a module by module basis. The primary method we have chosen in comparing modules is the sensitive bit density metric described in Section 5. This metric provides guidance on which modules would be most efficiently mitigated using redundancy techniques.

Table 6 provides the number of sensitive bits, bit area, and sensitive bit density metric for each component. The bit area is broken down into primary bits, interconnect bits (which also include primary bits), and logic bits. Recall from the previous subsection that primary bits are interconnect configuration bits that result in open failures. The Interconnect Bit column is the total of all interconnect configuration bits for the specified component. The logic bits column is the total number of bits used to configure the logic elements of the component. The bit area is the sum of the interconnect and logic bits for the component.

It is interesting to note that the majority of the bit area of each component is composed of nonprimary interconnect bits. The components with the highest densities of sensitive bits have far fewer sensitive configuration bits than the number of configuration bits specified in the interconnect bit area. This data seems to suggest that many shorting connections (caused by the faults in non-primary bit interconnect) do not affect the output of individual PIPs. This may suggest that our current method overestimates the actual sensitive bit area of a component; however, at the present time we do not know which non-primary configuration bits will cause errors and which ones will not; therefore, we must include all of these bits to avoid underestimating the bit area of the component. Further work would need to be done to conclusively show which bits will and will not cause shorting failures.

Since the ICAP component on the DUT is almost purely interconnect, it is comforting to see that the number of

sensitive ICAP bits found exceeds the number of primary bits. In fact, our test caught 959 of the 987 faults introduced into the primary bits of the ICAP. Twenty-six of the twenty-eight remaining primary bits exhibited the hiding behavior discussed in Section 5. The other two bits exhibited masking behavior. All 28 bits belonged to a state machine that initializes the ICAP immediately after configuration. After initialing the ICAP the nets in the state machine hold values of "1" that do not ever change. A fault in one of the primary bits of a PIP causes the PIP's output to be pulled high leaving the signal at its correct value and exhibiting hiding behavior. The fact that all the possible open failures within the ICAP were caught demonstrates that our tests of the ICAP during fault injection were sufficient to catch errors on all the interconnect of the ICAP module.

The first group of components we discuss are the modules that are on the main processor data path. These components occupy 4 of the first 5 rows of Table 6. These components are the IPLB, DPLB, PPC, and DDR2. The IPLB and DPLB are the connections between the PowerPC and the DDR2 memory controller. The PPC is the module that is used to instantiate PowerPC processor. The PPC defines the connectivity to the IPLB, DPLB, processor local bus (PLB), ground and VCC connections. The DDR2 is a multiported memory controller that has direct connections to the PowerPC through the IPLB and DPLB and also is connected through another port to the PLB.

One thing to notice about Table 6 is that the DDR2 contained 85% of the sensitive bits found in the fault injection test but did not have the highest sensitive bit density metric. The IPLB and DPLB each contained only 2% of the sensitive bits but had the highest density metrics. This suggests that using redundancy techniques on the IPLB and DPLB will result in a more efficient use of area than using redundancy techniques on the DDR2.

The reason the DDR2 has a lower metric than the IPLB and DPLB is because of the second port connected to the PLB. The second port adds more bit-area to the DDR2 component but does not add any sensitive configuration bits because the port is essentially unused. A solution to this problem would be to examine the DDR2 module and determine the sensitive-bit density metric of the port that is used by the processor. The metric is then likely to suggest that using redundancy techniques is likely more efficient for that one port. We also notice that the GPIO module, which the processor uses to respond to test requests also has a high sensitive bit density metric. The reader is referred to Table 6 for the remaining results.

We should also mention that the sensitive bit density metric may change under different operation modes of the system. Our results have only presented the metric from one operation mode. It is possible that the module with the highest metric may change throughout the operation of the design. This means that there are possible gains from using different mitigation techniques as the sensitive bit density changes during the run-time of the design. A possible way to accomplish this would be to use the dynamic partial reconfiguration ability of FPGAs.

7. Conclusion

A reliable Linux OS can be a useful tool on an FPGA-embedded system. Fault injection testing is an important first step in testing the reliability of FPGA-Embedded Linux Systems. Our test-bed provides an effective platform for fault injection and other useful experiments investigating the low-level details of the FPGA. Using different analyses in fault injection may help in identifying the lowest cost SEU mitigation techniques for FPGA-Embedded Linux Systems and provide additional confidence in fault injection results.

Our experiment showed that our Linux FPGA-embedded system was sensitive to 182,515 bits which gave our unmitigated system an MTBF of 19.7 days. We were able to match more than 99% of all failures to design utilized FPGA resources. We also found that in our system routing failures account for 83% of all design failures, while logic failures accounted for the other 16%.

In our FPGA resource analysis, we described the effects and identification of open, bridging, and buffer failures. We found that bridging and buffer failures often impose logic-like behavior on the net that was affected by the failure. Some of these logic-like behaviors affected the operation of the circuit while others did not.

We also suggested the use of the terms hiding and kernel resiliency to describe phenomena in which faults injected into the user circuit did not cause kernel failure. We even found an example of hiding in our ICAP circuitry that demonstrates the phenomena actually exists. Although we did not find an example of kernel resiliency, there are ways it could be evaluated. For example, by performing fault injection testing on the instruction and data memory of the Linux kernel, we may be able to draw correlations between the faults that cause kernel failure and the faults that do not.

In our user-circuit analysis, we showed that modules could be compared using the sensitive bit density metric. This metric helps determine which modules can be mitigated most efficiently using redundancy techniques such as TMR and PTMR. The sensitive bit density metric also indicated that using the same mitigation technique for the whole design may not lead to the most efficient use of FPGA area. Additionally, since the metric may change for a module during the operation of the circuit, it may be profitable to change the mitigation method of a module during run-time using dynamic partial reconfiguration.

Acknowledgments

The authors would like to express their thanks to the Information Sciences Institute-East (ISI) for providing the ML410 boards and custom connector board. They would especially like to thank Neil Steiner for his assistance in the initial stages of their project.

References

[1] B. Ramesh, T. Bretschneider, and I. Mcloughlin, "Embedded linux platform for a fault tolerant space based parallel

computer," in *Proceedings of the Real-Time Linux Workshop*, pp. 39–46, 2004.

[2] D. S. Katz, "Application-based fault tolerance for spaceborne applications," 2004, http://hdl.handle.net/2014/10574.

[3] M. Caffrey, "A space-based reconfigurable radio," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '02)*, T. P. Plaks and P. M. Athanas, Eds., pp. 49–53, CSREA Press, June 2002.

[4] M. Caffrey, K. Morgan, D. Roussel-Dupre et al., "On-orbit flight results from the reconfigurable cibola flight experiment satellite (CFESat)," in *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 3–10, April 2009.

[5] M. Caffrey, K. Katko, and A. Nelson, "The cibola flight experiment," in *Proceedings of the 23rd Annual Small Satellite Conference*, August 2009.

[6] C. Carmichael, M. Caffrey, and A. Salazar, "Correcting single-event upsets through virtex partial configuration," *Xilinx Application Notes*, vol. 1.0, 2000.

[7] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda, "On the optimal design of triple modular redundancy logic for sram-based fpgas," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, pp. 1290–1295, IEEE Computer Society, Washington, DC, USA, 2005.

[8] S. Baloch, T. Arslan, and A. Stoica, ""Probability based partial triple modular redundancy technique for reconfigurable architectures," in *Proceedings of the IEEE Aerospace Conference*, p. 7, 2006.

[9] D. L. McMurtrey, *Using deplication with compare for on-line error detection in FPGA-based designs*, Ph.D. dissertation, Brigham Young University, Provo, Utah, USA, 2006.

[10] F. Lima, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, "A fault injection analysis of virtex fpga tmr design methodology," in *Proceedings of the 6th European Conference on Radiation and Its Effects on Components and Systems*, pp. 275–282, 2001.

[11] J. Monson, M. Wirthlin, and B. Hutchings, "Fault injection results of linux operating on an fpga embedded platform," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, pp. 37–42, 2010.

[12] U. Legat, A. Biasizzo, and F. Novak, "Automated SEU fault emulation using partial FPGA reconfiguration," in *Proceedings of the 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS '10)*, pp. 24–27, 2010.

[13] M. S. Reorda, L. Sterpone, M. Violante, M. Portela-Garcia, C. Lopez-Ongil, and L. Entrena, "Fault injection-based reliability evaluation of SoPCs," in *Proceedings of the 11th IEEE European Test Symposium (ETS '06)*, pp. 75–82, 2006.

[14] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "FPGA-based fault injection for microprocessor systems," in *Proceedings of the 10th Asian Test Symposium*, pp. 304–309, November 2001.

[15] L. Sterpone and M. Violante, "An analysis of SEU effects in embedded operating systems for Real-Time applications," in *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE '07)*, pp. 3345–3349, 2007.

[16] E. Johnson, M. Caffrey, P. Graham, N. Rollins, and M. Wirthlin, "Accelerator validation of an FPGA SEU simulator," *IEEE Transactions on Nuclear Science*, vol. 50, no. 6 I, pp. 2147–2157, 2003.

[17] L. Sterpone and M. Violante, "Static and dynamic analysis of SEU effects in SRAM-based FPGAs," in *Proceedings of the 12th IEEE European Test Symposium (ETS '07)*, pp. 159–164, May 2007.

[18] L. Sterpone and M. Violante, "A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2217–2223, 2005.

[19] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: accelerating FPGA compilation with hard macros for rapid prototyping," in *Proceedings of the 19th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, May 2011.

[20] G. Allen, "Virtex-4VQ dynamic and mitigated single event upset characterization summary," 2009, http://hdl.handle.net/2014/41104.

[21] C. Beckhoff, D. Koch, and J. Torresen, "Short-circuits on fpgas caused by partial runtime reconfiguration," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 596–601, 2010.

[22] S. Srinivasan, A. Gayasen, and N. Vijaykrishnan, "Leakage control in FPGA routing fabric," in *Proceedings of Conference on Asia South Pacific Design Automation (ASP-DAC'05)*, vol. 1, pp. 661–664z, 2005.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 915178, 12 pages
doi:10.1155/2012/915178

Research Article

NCOR: An FPGA-Friendly Nonblocking Data Cache for Soft Processors with Runahead Execution

Kaveh Aasaraai and Andreas Moshovos

Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada

Correspondence should be addressed to Kaveh Aasaraai, aasaraai@eecg.toronto.edu

Received 30 April 2011; Revised 22 August 2011; Accepted 27 August 2011

Academic Editor: Viktor K. Prasanna

Soft processors often use data caches to reduce the gap between processor and main memory speeds. To achieve high efficiency, simple, blocking caches are used. Such caches are not appropriate for processor designs such as Runahead and out-of-order execution that require nonblocking caches to tolerate main memory latencies. Instead, these processors use non-blocking caches to extract memory level parallelism and improve performance. However, conventional non-blocking cache designs are expensive and slow on FPGAs as they use content-addressable memories (CAMs). This work proposes NCOR, an FPGA-friendly non-blocking cache that exploits the key properties of Runahead execution. NCOR does not require CAMs and utilizes smart cache controllers. A 4 KB NCOR operates at 329 MHz on Stratix III FPGAs while it uses only 270 logic elements. A 32 KB NCOR operates at 278 Mhz and uses 269 logic elements.

1. Introduction

Embedded system applications increasingly use soft processors implemented over reconfigurable logic. Embedded applications, like other application classes, evolve over time and their computation needs and structure change. If the experience with other application classes is any indication, embedded applications will evolve and incorporate algorithms with unstructured instruction-level parallelism. Existing soft processor implementations use in-order organizations since these organizations map well onto reconfigurable logic and offer adequate performance for most existing embedded applications. Previous work has shown that for programs with unstructured instruction-level parallelism, a 1-way out-of-order (OoO) processor has the potential to outperform a 2- or even a 4-way superscalar processor in a reconfigurable logic environment [1]. Conventional OoO processor designs are tuned for custom logic implementation and rely heavily on content addressable memories, multiported register files, and wide, multisource and multidestination datapaths. These structures are inefficient when implemented on an FPGA fabric. It is an open question whether it is possible to design an FPGA-friendly soft core that offers the benefits of OoO execution while overcoming the complexity and inefficiency of conventional OoO structures.

A lower complexity alternative to OoO architectures is *Runahead Execution*, or simply *Runahead*, which offers most of the benefits of OoO execution [2]. Runahead relies on the observation that often most of the performance benefits of OoO execution result from allowing multiple outstanding main memory requests. Runahead extends a conventional in-order processor with the ability to continue execution even on a cache miss, with the hope to find more useful misses and thus overlap memory requests. Section 3 discusses Runahead execution in more detail.

Runahead was originally demonstrated on high-end general-purpose systems where main memory latencies are in the order of a few hundred cycles. This work demonstrates that even under the lower main memory latencies (a few tens of cycles) observed in FPGA-based systems today, Runahead can improve performance. However, Runahead, as originally proposed, relies on nonblocking caches which do not map well on FPGAs as they rely on highly associative Content-Addressable Memories (CAMs).

This work presents NCOR (Nonblocking Cache Optimized for Runahead execution) an FPGA-friendly alternative

to conventional nonblocking caches. NCOR does not use CAMs Instead it judiciously sacrifices some of the flexibility of a conventional nonblocking cache to achieve higher operating frequency and thus superior performance when implemented on an FPGA. Specifically, NCOR sacrifices the ability to issue secondary misses, that is, requests for memory blocks that map onto a cache frame with an outstanding request to memory. Ignoring secondary misses enables NCOR to track outstanding misses within the cache frame avoiding the need for associative lookups. This work demonstrates that this simplification does not affect performance nor correctness under Runahead execution.

This work extends the work that introduced NCOR [3] as follows. It quantitatively demonstrates that conventional nonblocking caches are not FPGA-friendly because they require CAMs which lead to a low operating frequency and high area usage. It provides a more detailed description of NCOR and of the underlying design tradeoffs. It explains how NCOR avoids the inefficiencies of conventional designs in more detail. It compares the frequency and area of conventional, nonblocking CAM-based caches and NCOR. Finally, it measures how often secondary misses occur in Runahead execution showing that they are relatively infrequent.

The rest of this paper is organized as follows. Section 2 reviews conventional, CAM-based nonblocking caches. Section 3 reviews Runahead execution and its requirements. Section 4 provides the rationale behind the optimizations incorporated in NCOR. Section 5 presents NCOR architecture. Section 6 discusses the FPGA implementation of NCOR. Section 7 evaluates NCOR comparing it to conventional CAM-based cache implementations. Section 8 reviews related work, and Section 9 summarizes our findings.

2. Conventional Nonblocking Cache

Nonblocking caches are used to extract Memory Level Parallelism (MLP) and reduce latency compared to conventional blocking caches that service cache miss requests one at a time. In blocking caches if a memory request misses in the cache, all subsequent memory requests are blocked and are forced to wait for the outstanding miss to receive data from main memory. Blocked requests may include request for data that is already in the cache or that could be serviced concurrently by modern main memory devices. A nonblocking cache does not block subsequent memory requests when a request misses. Instead, these requests are allowed to proceed concurrently. Some may hit in the cache, while others may be sent to the main memory system. Overall, because multiple requests are serviced concurrently, the total amount of time the program has to wait for the memory to service its requests is reduced.

To keep track of outstanding requests and to make the cache available while a miss is pending, information regarding any outstanding request is stored in Miss Status Holding Registers (MSHRs) [4]. MSHRs maintain the information that is necessary to direct the data received from the main memory to its rightful destination, for example, cache frame or a functional unit. MSHRs can also detect whether a memory request is for a block for which a previous request is still pending. Such requests can be serviced without issuing an additional main memory request. To detect these accesses and to avoid duplicate requests, for every request missing in the cache, the entire array of MSHRs is searched. A matching MSHR means the data has already been requested from memory. Such requests are queued and serviced when the data arrives. Searching the MSHRs requires an associative lookup, which is implemented using a Content-Addressable Memory (CAM). CAMs map poorly to reconfigurable logic as Section 7 shows. As the number of MSHRs bounds the maximum number of outstanding requests, more MSHRs are desirable to extract more MLP. Unfortunately, the area and latency of the underlying CAM grow disproportionately large with the number of MSHRs making large number of MSHRs undesirable.

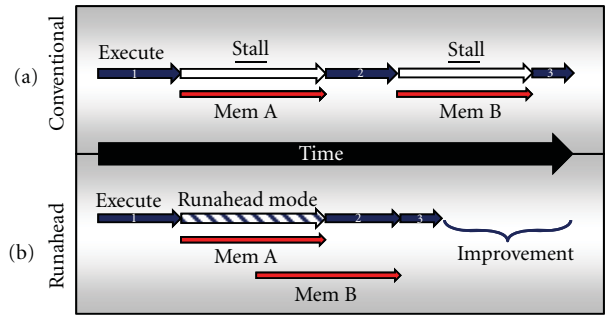

Figure 1: (a) In-order execution of instructions resulting in stalls on cache misses. (b) Overlapping memory requests in Runahead execution.

3. Runahead Execution

Runahead builds on top of a simple in-order processor that maps well onto FPGA fabrics. Runahead improves performance by avoiding the stalls caused by cache misses as Figure 1(a) depicts. A conventional in-order processor stalls whenever a memory request misses in the cache. Even on reconfigurable platforms, a main memory request may take several tens of soft processor cycles to complete limiting performance. Main memory controllers, however, support multiple outstanding requests. Runahead exploits this ability and improves performance by requesting multiple data blocks from memory instead of stalling whenever a request is made. For Runahead to be advantageous the following concerns must be addressed: (1) how can the processor reasonably determine which addresses the program will soon require and (2) how much additional functionality is needed to allow the processor to issue multiple requests.

Runahead shows that it is relatively straightforward to address both issues without overly complicating existing in-order processor designs. As Figure 1(b) shows, upon encountering a cache miss, or a *trigger miss*, instead of stalling the pipeline, the processor continues to execute subsequent independent instructions. This is done with the hope of finding more cache misses to overlap with the trigger miss. Effectively, Runahead uses the program itself to predict near-future accesses that the program will perform and overlaps their retrieval.

When a cache miss is detected, the processor creates a checkpoint of its architectural state (e.g., registers) and enters the *Runahead* execution mode. While the trigger miss is pending, the processor continues executing subsequent independent instructions. Upon the delivery of the trigger miss data, the processor uses the checkpoint and restores all architectural state, so that the results produced in Runahead mode are not visible to the program. The processor then resumes normal execution starting immediately after the instruction that caused the trigger miss.

While all results produced during Runahead mode are discarded, all valid memory requests are serviced by main memory and the data requested is eventually placed in the processor cache. If the program subsequently accesses some of this data, performance may improve as this data was prefetched (i.e., requested earlier from memory). Provided that a sufficient number of instructions independent of the trigger miss are found during runahead mode, the processor has a good chance of reaching other memory requests that miss. As long as a sufficient number of useful memory requests are reached during Runahead mode, performance improves as the processor effectively prefetches these into the cache.

Performance trade-offs with Runahead are complex. On one side, the memory accesses that were initiated during Runahead mode and that fetch useful data effectively prefetch data for subsequent instructions and reduce overall execution time. On the other side, memory accesses that bring useless data pollute the cache and consume memory bandwidth and resources, for example, they may evict useful data from the cache or they may delay another request. Section 7 shows that in practice Runahead execution improves performance.

4. Making a Nonblocking Cache FPGA-Friendly

Runahead execution is conceptually an extension to a simple in-order processor. The simplicity of its architecture is one of the primary reasons that makes Runahead suitable for reconfigurable fabrics. However, for Runahead to be feasible on these fabrics, the extensions must come with low overhead. These extensions include checkpointing and nonblocking caches. Aasaraai and Moshovos proposed a fast, low-cost checkpointing mechanism for out-of-order execution on FPGAs [1]. The same solution could be used in Runahead execution. As Section 7 shows, conventional nonblocking cache designs based on MSHRs do not map well on FPGAs. Accordingly there is a need to design a low-cost nonblocking cache suitable for FPGAs. This work observes that Runahead execution does not need the full functionality of a conventional nonblocking cache and exploits this observation to arrive to an FPGA-friendly nonblocking cache design for Runahead execution.

Conventional nonblocking caches that use MSHRs do not map well on reconfigurable fabrics. The primary reason is that MSHRs use a CAM to perform associative searches. As Section 7 shows MSHRs lead to low clock frequencies and high area usage. In addition to MSHRs, the controller of a nonblocking cache is considerably more complex compared to the one in a blocking cache. The controller is responsible for a wide range of concurrent operations resulting in large, complex state machines. This work presents the Nonblocking Cache Optimized for Runahead execution, or NCOR. NCOR is an FPGA-friendly design that revisits the conventional nonblocking cache design considering the specific needs of Runahead execution. NCOR does away with MSHRs and incorporates optimizations for the cache controller and data storage.

4.1. Eliminating MSHRs. Using the following observations, NCOR eliminates the MSHRs.

(1) As originally proposed, Runahead executes *all* trigger-miss-independent instructions during Runahead mode. However, since the results produced in Runahead mode are later discarded, the processor can choose not to execute some of these instructions as it finds necessary. This option of selective execution can be exploited to reduce complexity by avoiding the execution of instructions that require additional hardware support. One such instruction class is those that cause *secondary misses*, that is, misses on already pending cache frames. Supporting secondary misses is conventionally done via MSHRs, which do not map well to FPGAs.

(2) In most cases servicing secondary misses offers no performance benefit. There are two types of secondary misses: redundant and distinct. A redundant secondary miss requests the same memory block as the trigger miss while distinct secondary miss requests a different memory block that happens to map to the same cache frame as the trigger miss. Section 7 shows that distinct secondary misses are very small fraction of the memory accesses made in Runahead mode.

Servicing a redundant secondary miss cannot directly improve performance further as the trigger miss will bring the data in the cache. A redundant secondary miss may be feeding another load that will miss and that could be subsequently prefetched. However this is impossible as the trigger-miss is serviced first which causes the processor to switch to normal execution. Distinct secondary misses could prefetch useful data, but as Section 7 shows, this has a negligible impact on performance.

Based on these observations the processor can simply discard instructions that cause secondary misses during runahead mode while getting most, and often all, of the performance benefits of runahead execution. However, NCOR still needs to identify secondary misses to be able to discard them. NCOR identifies secondary misses by tracking outstanding misses within the cache frames using a single *pending* bit per frame. Whenever an address misses in the cache, the corresponding cache frame is marked as *pending*. Subsequent accesses to this frame would observe the *pending* bit and will be identified as secondary misses and discarded by the processor. Effectively, NCOR embeds the MHSRs in the cache while judiciously simplifying their functionality to reduce complexity and maintain much of the performance benefits.

4.2. Making the Common Case Fast. Ideally, the cache performs all operations in as few cycles as possible. In particular, it is desirable to service cache hits in a single cycle, as hits are expected to be the common case. In general, it is desirable to design the controller to favor the frequent operations over the infrequent ones. Accordingly, NCOR uses a three-part cache controller which favors the most frequent requests, that is, cache hits, by dedicating a simple subcontroller just for hits. Cache misses and all noncacheable requests (e.g., I/O requests) are handled by other sub-controllers which are triggered exclusively for such events and are off the critical path for hits. These requests complete in multiple cycles. The next section explains the NCOR cache controller architecture in detail.

5. NCOR Architecture

Figure 2 depicts the basic structure of NCOR. The cache controller comprises *Lookup*, *Request*, and *Bus* components. NCOR also contains *Data*, *Tag*, *Request,* and *Metadata* storage units.

5.1. Cache Operation. NCOR functions as follows.

(i) *Cache Hit*. The address is provided to *Lookup* which determines, as explained in Section 5.2, that this request is a hit. The data is returned in the same cycle for load operations and is stored in the cache during the next cycle for store operations. Other soft processor caches, such as those of Altera Nios II, use two cycles for stores as well [5].

(ii) *Tirgger Cache Miss*. If *Lookup* identifies a cache miss, it sends a signal to *Request* to generate the necessary requests to handle the miss. *Lookup* blocks the cache interface until *Request* signals back that it has generated all the necessary requests.

Request generates all the necessary requests directed at *Bus* to fulfill the pending memory operation. If a dirty line must be evicted, a write-back request is generated first. Then a cache line read request is generated and placed in the *Queue* between *Request* and *Bus*.

Bus receives requests through the *Queue* and sends the appropriate signals to the system bus. The *pending bit* of the cache frame that will receive the data is set.

(iii) *Secondary Cache Miss in Runahead Mode*. If *Lookup* identifies a secondary cache miss, that is, a miss on a cache frame with *pending* bit set, it discards the operation.

(iv) *Secondary Cache Miss in Normal Mode*. If *Lookup* identifies a secondary cache miss in normal execution mode, it blocks the pipeline until the frame's *pending* bit is cleared. It is possible to have a secondary miss in normal execution mode as a memory access initiated in Runahead mode may be still pending. In normal execution processor cannot discard operations and must wait for the memory request to be fulfilled.

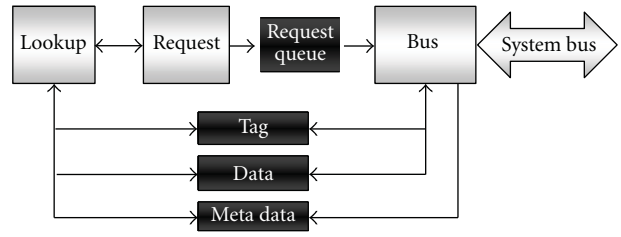

Figure 2: Nonblocking cache structure.

The following subsections describe the function of each NCOR component.

5.2. Lookup. *Lookup* is the cache interface that communicates with the processor and receives memory requests. *Lookup* performs the following operations.

(i) For cache accesses, *Lookup* compares the request address with the tag stored in the *Tag* storage to determine whether this is a hit or a miss.

(ii) For cache hits, on a load, *Lookup* reads the data from the *Data* storage and provides it to the processor in the same cycle as the *Tag* access. Reading the *Data* storage proceeds in parallel with the *Tag* access and comparison. Stores, on the other hand, take two cycles to complete as writes to the *Data* storage happen in the cycle after the hit is determined. In addition, the cache line is marked as dirty.

(iii) For cache misses, *Lookup* marks the cache line as pending.

(iv) For cache misses and non-cacheable requests, *Lookup* triggers *Request* to generate the appropriate requests. In addition, for loads, it stores the instruction metadata in the *MetaData* storage. *Lookup* blocks the processor interface until *Request* signals it has generated all the necessary requests.

(v) For cache accesses, whether the request hits or misses in the cache, if the corresponding cache line is *pending*, *Lookup* discards the request if the processor is in Runahead mode. However, if the processor is in normal execution mode, *Lookup* stalls the processor.

5.3. Request. *Request* is normally idle waiting for a trigger from *Lookup*. When triggered, it issues the appropriate requests to *Bus* through request *Queue*. *Request* performs the following operations.

(i) Waits in the idle state until triggered by *Lookup*.

(ii) For cache misses, *Request* generates a cache line read request. In addition if the evicted line is dirty, *Request* generates a cache line write-back request.

(iii) For non-cacheable requests, depending on the operation, *Request* generates a single read or write request.

(iv) When all necessary requests are generated and queued, *Request* notifies *Lookup* of its completion and returns to its idle state.

5.4. Bus. *Bus* is responsible for servicing bus requests generated by *Request*. *Bus* receives requests through the request *Queue* and communicates through the system bus with the main memory and peripherals. *Bus* consists of two internal modules.

5.4.1. Sender. *Sender* sends requests to the system bus. It removes requests from the request *Queue* and, depending on the request type, sends the appropriate signals to the system bus. A request can be of one of the following types.

(i) *Cache Line Read*. Read requests are sent to the system bus for each data word of the cache line. The critical word (word originally requested by the processor) is requested first. This ensures minimum wait time for data delivery to the processor.

(ii) *Cache Line Write-Back*. Write requests are sent to the system bus for each data word of the dirty cache line. Data words are retrieved from *Data* storage and sent to the system bus.

(iii) *Single Read/Write*. A single read/write request is sent to the memory/peripheral through the system bus.

5.4.2. Receiver. *Receiver* handles the system bus responses. Depending on the processor's original request type, one of the following actions is taken.

(i) *Load from Cache.* Upon receipt of the first data word, *Receiver* signals request completion to the processor and provides the data. This is done by providing the corresponding metadata from the *MetaData* storage to the processor. *Receiver* also stores all the data words received in the *Data* storage. Upon receipt of the last word, it stores the cache line tag in the corresponding entry in the *Tag* storage, sets the valid bit, and clears both dirty and *pending* bits.

(ii) *Store to Cache*. The first data word received is the data required to perform the store. Receiver combines the data provided by the processor with the data received from the system bus and stores it in the *Data* storage. It also stores subsequent data words, as they are received, in the *Data* storage. Upon the receipt of the last word, *Receiver* stores the cache line tag in the corresponding entry in the *Tag* storage, sets both valid and dirty bits, and clears the *pending* bit.

(iii) *Noncacheable Load*. Upon receipt of the data word, *Receiver* signals request completion to the processor and provides the data. It also provides the corresponding metadata from the *MetaData* storage.

5.5. Data and Tag Storage. The *Data* and *Tag* storage units are tables holding cache line data words, tags, and status bits. *Lookup* and *Bus* both access *Data* and *Tag*.

5.6. Request Queue. Request *Queue* is a FIFO memory holding requests generated by *Request* directed at *Bus*. Request *Queue* conveys requests in the order they are generated.

5.7. MetaData. For outstanding load requests, that is, load requests missing in the cache or non-cacheable operations, the cache stores the metadata accompanying the request. This data includes Program Counter and destination register for load instructions. Eventually when the request is fulfilled this information is provided to the processor along with the data loaded from the memory or I/O. This information allows the loaded data to be written to the register file. *MetaData* is designed as a queue so that requests are processed in the order they were received. No information is placed in the *MetaData* for Stores as the processor does not require acknowledgments for their completion.

6. FPGA Implementation

This section presents the implementation of the nonblocking cache on FPGAs. It discusses the design challenges and the optimizations applied to improve clock frequency and minimize the area. It first discusses the storage organization and usage and the corresponding optimizations. It then discusses the complexity of the cache controller's state machine and how its critical path was shortened for the most common operations.

6.1. Storage Organization. Modern FPGAs contain dedicated Block RAM (BRAM) storage units that are fast and take significantly less area compared to LUT-based storage. This subsection explains the design choices that made it possible to use BRAMs for most of the cache storage components.

6.1.1. Data. Figure 3 depicts the *Data* storage organization. As BRAMs have a limited port width, the entire cache line does not fit in one entry. Consequently, cache line words are spread, one word per entry, over multiple BRAM entries. This work targets the Nios-II ISA [5] which supports byte, half-word, and word stores (one, two, and four, bytes resp.). These are implemented using the BRAM *byte enable* signal [6]. Using this signal avoids two-stage writes (read-modify-write) which would increase area due to the added multiplexers.

6.1.2. Tag. Figure 3 depicts the *Tag* storage organization. Unlike cache line data, a tag fits in one BRAM entry. In order to reduce BRAM usage, we store cache line status bits, that is, valid, dirty, and *pending* bits, along with the tags.

Despite the savings in BRAM usage by storing cache line status bits along with the tags, the following problem arises. *Lookup* makes changes only to the *dirty* and *pending* bits and should not alter *valid* or *Tag* bits. In order to preserve *valid* and *Tag* bits while performing a write, a two-stage write could be used, in which bits are first read and then written back. This read-modify-write sequence increases area and complexity and hurts performance. We overcome this problem by using the *byte enable* signals. As Figure 3 shows, we

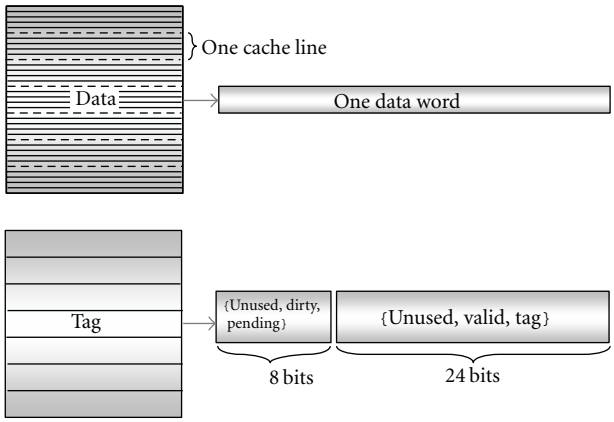

Figure 3: The organization of the *Data* and *Tag* storage units.

store *valid* and *Tag* bits in the lower 24 bits and *dirty* and *pending* bits in the higher eight bits. Then using the *byte enable* signal, *Lookup* is able to change only the *dirty* and *pending* bits.

6.2. BRAM Port Limitations. Although BRAMs provide fast and area-efficient storage, they have a limited number of ports. A typical BRAM in today's FPGAs has one read and one write port [6]. Figure 4 shows that both *Lookup* and *Bus* write and read to/from the *Data* and *Tag* storages. This requires four ports. Our design uses only two ports based on the following observations: BRAMs can be configured to provide two ports, each providing both write and read operations over one address line. Although *Lookup* and *Bus* both write and read to/from the *Data* and *Tag* at the same time, each only requires one address line.

6.2.1. Tag. For every access from *Lookup* to the *Tag* storage, *Lookup* reads the *Tag*, *valid*, *dirty,* and *pending* bits for a given cache line. *Lookup* also writes to the *Tag* storage in order to mark a line dirty or *pending*. However, reads and writes never happen at the same time as marking a line dirty (for stores) or pending (for misses) happens one cycle after the tag and other status bits are read. *Bus* only writes to the *Tag* storage, when a cache line is retrieved from the main memory. Therefore, dedicating one address line to *Lookup* and one to *Bus* is sufficient to access the *Tag* storage.

6.2.2. Data. For every *Lookup* access to the *Data* storage, *Lookup* either reads or writes a single, or part of a, word. However, *Bus* may need to write to or read from the *Data* storage at the same time. This occurs if *Bus* is sending words of a write-back request while a previously requested data is being delivered by the system bus. To avoid this conflict, we restrict *Bus* to send a write-back data word only when the system bus is not delivering any data. Forward progress is guaranteed as outstanding write-back requests do not block responses from the system bus. This restriction minimally impacts cache performance as words are sent as soon as the system bus is idle. In Section 7.9 we show that even in the worst case scenario, impact on performance is marginal. With this modification, dedicating one address line to *Lookup* and one to *Bus* is sufficient for accessing the *Data* storage.

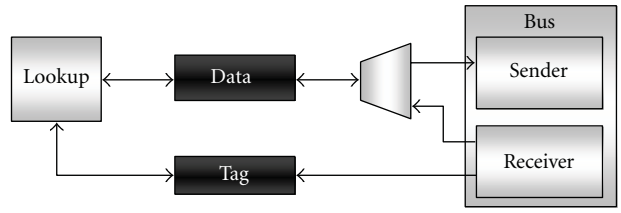

Figure 4: Connections between Data and Tag storages and Lookup and Bus components.

6.3. State Machine Complexity. The cache controller is responsible for looking up in the cache, performing loads and stores, generating, bus requests and handling bus transactions. Given the number of operations that the controller handles, in many cases concurrently, it requires a large and complex state machine. A centralized cache controller can be slow and has the disadvantage of treating all requests the same. However, we would like the controller to respond as quickly as possible to those requests that are most frequent, that is, requests that hit in the cache. Accordingly, we partition the controller into subcomponents. One could partition the controller into two components of CPU-side and bus-side, as Figure 5(a) shows. The CPU-side component would be responsible for looking up addresses in the cache, performing loads and stores, handling misses and non-cacheable operations, and sending necessary requests to the bus-side component. The bus-side component would communicate with the main memory and system peripherals through the system bus.

Due to the variety of operations that the CPU-side component is responsible for, we find that it still requires a non-trivial state machine. The state machine has numerous input signals, and this reduces performance. Among its inputs is the cache hit/miss signal, a time-critical signal due to the large comparator used for tag comparison. As a result, implementing the CPU-side component as one state machine leads to a long critical path.

Higher operating frequency is possible by further partitioning the CPU-side component into two subcomponents, *Lookup* and *Request*, which cooperatively perform the same set of operations. Figure 5(b) depicts the three-component cache controller. The main advantage of this controller is that cache lookups that hit in the cache, the most frequent operations, are handled only by *Lookup* and are serviced as fast as possible. However, this organization has its own disadvantages. In order for the *Lookup* and *Request* to communicate, for example, in the case of cache misses, extra clock cycles are required. Fortunately, these actions are relatively rare. In addition, in such cases servicing the request takes in the order of tens of cycles. Therefore, adding one extra cycle delay to the operation has little impact on performance. Figure 6 shows an overview of the two state machines corresponding to Lookup and Request.

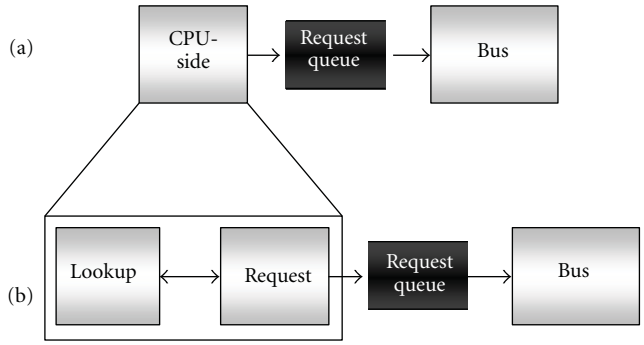

Figure 5: (a) Two-component cache controller. (b) Three-component cache controller.

6.4. Latching the Address. We use BRAMs to store data and tags in the cache. As BRAMs are synchronous rams, the input address needs to be available just before the appropriate clock edge (rising in our design) of the cycle when cache lookup occurs. Therefore, in a pipelined processor, the address has to be forwarded to the cache from the previous pipeline stage, for example, the execute stage in a typical 5-stage pipeline. After the first clock edge, the input address to the cache changes as it is forwarded from the previous pipeline stage. However, the input address is further required for various operations, for example, tag comparison. Therefore, the address must be latched.

Since some cache operations take multiple cycles to complete, the address must be latched only when a new request is received. This occurs when *Lookup* 's state machine is *entering* the *lookup state*. Therefore, the address register is clocked based on the *next state* signal. This is a time-critical signal, and using it to clock a wide register, as is the case with the address register, negatively impacts performance.

To avoid using this time-critical signal we make the following observations. The cache uses a latched address in two phases: in the first cycle for tag comparison and in subsequent cycles for writes to *Data* storage and request generations. Accordingly, we can use two separate registers, *addr_always* and *addr_lookup* one per phase. At every clock cycle, we latch the input address into *addr_always*. We use this register for tag comparison in the first cycle. At the end of the first cycle, if *Lookup* is *in* the lookup state, we copy the content of *addr_always* into *addr_lookup*. We use this register for writes to the cache and request generation. As a result, the *addr_always* register is unconditionally clocked every cycle. Also, we use *Lookup*'s *current state* register, rather than its *next state* combinational signal, to clock the *addr_lookup* register. This improves the design's operating frequency.

7. Evaluation

This section evaluates NCOR. It first compares the area and frequency of NCOR with those of a conventional MSHR-based nonblocking cache. It then shows the potential performance advantage that Runahead execution has over an in-order processor using a nonblocking cache.

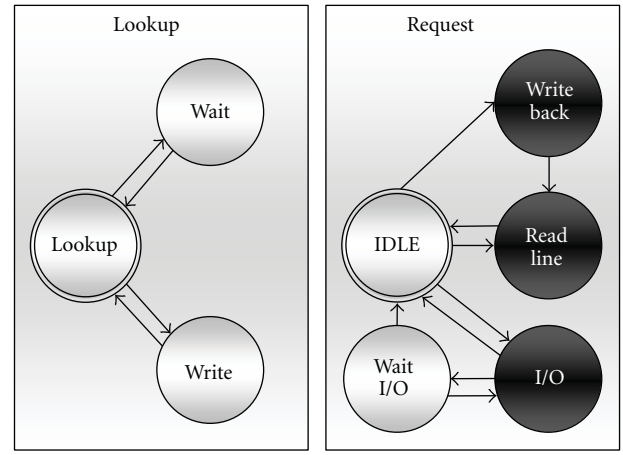

Figure 6: Lookup and Request state machines. Double-lined states are initial states. Lookup waits for Request completion in the "wait" state. All black states generate requests targeted at the Bus controller.

7.1. Methodology. The processor performance is estimated using a cycle-accurate, Nios II full system simulator capable of booting and running the uCLinux operating system [7]. The simulated processor models include a 5-stage in-order pipelined processor, 1- to 4-way superscalar processors, and Runahead processors. Table 1 details the simulated processor microarchitectures.

This evaluation uses benchmarks from the SPEC CPU 2006 suite which is typically used to evaluate desktop system performance [8]. These benchmarks are representative of applications that have unstructured ILP. The assumption is that in the future, soft-core-based systems will be called upon to run demanding applications that will exhibit sufficiently similar behavior. Those benchmarks that the uCLinux's toolchain cannot compile or that require a floating-point unit are excluded. Each benchmark run uses a reference input. These inputs are stored in main memory and are accessed through the block device driver (Ram-disk). Measurements are taken for a sample of one billion instructions after skipping several billions of instructions so that execution is past initialization.

Area and frequency are measured over Verilog implementations. Quartus II v10.0 did synthesis and place-and-routing and the target FPGA was the Altera Stratix III EP3SL150F1152C2. NCOR is compared against a conventional, MSHR-based nonblocking cache.

7.2. Simplified MSHR-Based Nonblocking Cache. NCOR was motivated as a higher-speed, lower-cost and complexity alternative to conventional, MSHR-based nonblocking caches. A comparison of the two designs is needed to demonstrate the magnitude of these advantages. Our experience has been that the complexity of a conventional nonblocking cache design quickly results in an impractically slow and large FPGA implementation. This makes it necessary to seek FPGA-friendly alternatives such as NCOR. For the purposes of demonstrating that NCOR is faster and smaller than a conventional nonblocking cache, it is sufficient to compare against a simplified nonblocking cache. This is sufficient, as long as

Table 1: Architectural properties of simulated processors.

I-cache size (Bytes)	32 K
D-Cache size (Bytes)	4–32 K
Cache line size	32 bytes
Cache associativity	Direct mapped
Memory latency	26 cycles
BPredictor type	GShare
BPredictor entries	4096
BTB entries	256
Pipeline stages	5
No. of outstanding misses	32

the results demonstrate the superiority of NCOR and provided that the simplified conventional cache is clearly faster and smaller than a full-blown conventional nonblocking cache implementation. The simplifications made to the conventional MSHR nonblocking cache are as follows.

(i) Requests mapping to a cache frame for which a request is already pending are not supported. Allowing multiple pending requests targeting the same cache frame substantially increases complexity.

(ii) Each MSHR entry tracks a single processor memory request as opposed to all processor requests for the same cache block request [4]. This eliminates the need for a request queue per MSHR entry which tracks individual processor requests, some of which may map onto the same cache block. In this organization the MSHRs serve as queues for both pending cache blocks and processor requests. Secondary misses are disallowed.

(iii) Partial (byte or half-word) loads/stores are not supported.

We use this simplified MSHR-based cache for FPGA resource and clock frequency comparison with NCOR. In the performance simulations we use a regular MSHR-based cache.

7.3. Resources. FPGA resources include ALUTs, block RAMs (BRAM), and the interconnect. In these designs interconnect usage is mostly tied to ALUT and BRAM usage. Accordingly, this section compares the ALUT and BRAM usage of the two cache designs. Figure 7 reports the number of ALUTs used by NCOR and the MSHR-based cache for various capacities. The conventional cache uses almost three times as many ALUTs compared to NCOR. There are two main reasons why this difference occurs. (1) The MSHRs in the MSHR-based cache use ALUTs exclusively instead of a mix of ALUTs and blockrams. (2) The large number of comparators required in the CAM structure of the MSHRs require a large number of ALUTs.

While the actual savings in the number of ALUTs are small compared to the number of ALUTs found in high capacity FPGAs available today (>100 K ALUTs) such small

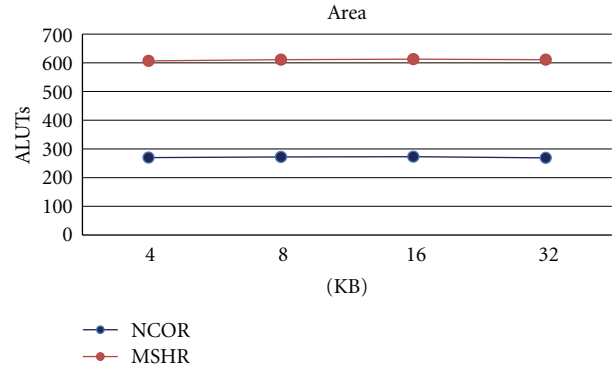

Figure 7: Area comparison of NCOR and MSHR-based caches over various capacities.

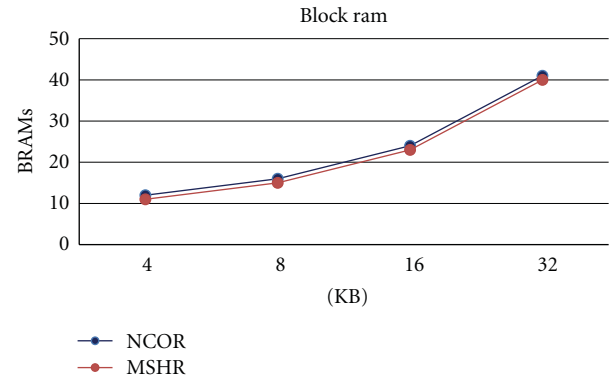

Figure 8: BRAM usage of NCOR and MSHR-based caches over various capacities.

savings add up, for example, in a multiprocessor environment. Additionally, there are designs where low-capacity FPGAs such in, for example, low budget, low power, or military applications.

In NCOR, the bulk of the cache is implemented using BRAMs, hence the low area usage of the cache. The vast majority of the BRAMs contain the cache's data, tag, and status bits. As expected, both caches experience a negligible change in ALUT usage over different capacities, as most of the cache storage is implemented using BRAMs.

Figure 8 shows the number of BRAMs used in each cache for various capacities. Compared to the conventional cache, NCOR uses one more BRAM as it stores pending memory requests in BRAMs rather than in MSHRs.

7.4. Frequency. Figure 9 reports the maximum clock frequency the NCOR and the MSHR-based cache can operate at and for various capacities. NCOR is consistently faster. The difference is at its highest (58%) for the 4 KB caches with NCOR operating at 329 MHz compared to the 207 MHz for the MHSR-based cache. For both caches, and in most cases, frequency decreases as the cache capacity increases. At 32 KB NCOR's operating frequency is within 18% of the 4 KB NCOR. While increased capacity results in reduced frequency in most cases, the 8 KB MSHR-based cache is faster

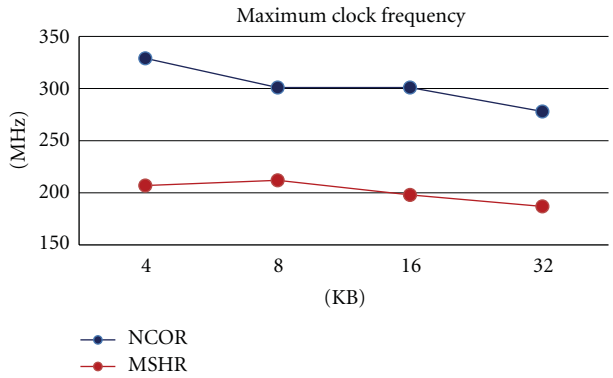

Figure 9: Clock frequency comparison of NCOR and of a four-entry MSHR-based cache over various cache capacities.

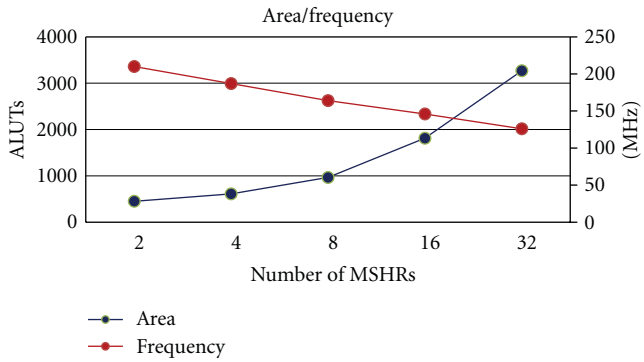

Figure 10: Area and clock frequency of a 32 KB MSHR-based cache with various number of MSHRs. The left axis is ALUTs and the right axis is clock frequency.

than its 4 KB counterpart. As the cache capacity increases, more sets are used, and hence the tag size decreases. Accordingly, this makes tag comparisons faster. At the same time, the rest of the cache becomes slower. These two latencies combine to determine the operating frequency which is at a local minimum at a capacity of 8 KB for the MSHR-based cache. However, as cache capacity continues to grow, any reduction in tag comparison latency is overshadowed by the increase in latency in other components.

7.5. MSHR-Based Cache Scalability. The NCOR studied in this work is capable of handling up to 32 outstanding requests. Supporting more outstanding requests in NCOR basically comes for free as these are tracked in BRAMs. An MSHR-based cache however uses CAMs, hence LUTs for storage. Figure 10 reports how the frequency and area of the MSHR-based cache scale with MSHR entry count. As expected, as the number of MSHRs increases clock frequency drops and area increases. With 32 MSHRs, the MHSR-based cache operates at only 126 MHz and requires 3269 ALUTs.

7.6. Runahead Execution. Figure 11 reports the speedup achieved by Runahead execution on 1- to 4-way superscalar processors. For this comparison, performance is measured

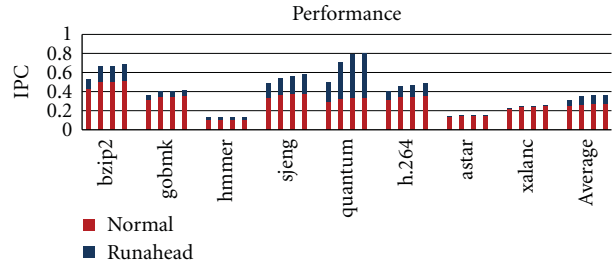

Figure 11: Speedup gained by Runahead execution on 1- to 4-way superscalar processors. The lower parts of the bars show the IPC of the normal processors. The full bars show the IPC of the Runahead processor.

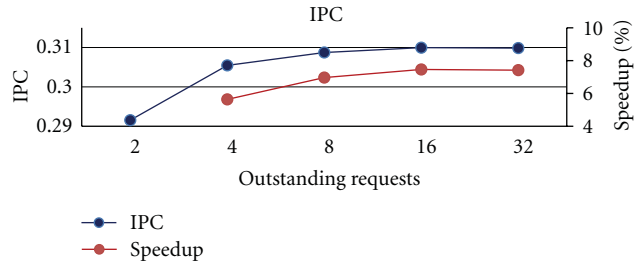

Figure 12: The impact of number of outstanding requests on IPC. Speedup is measured over the first configuration with two outstanding requests.

as the instructions per cycle (IPC) rate. IPC is a frequency-independent metric and thus is useful in determining the range of frequencies for which an implementation can operate on and still outperform an alternative. Runahead is able to outperform the corresponding in-order processor by extracting memory-level parallelism effectively hiding the high main memory latency. For a typical single-issue pipeline (1-way), on average, Runahead improves IPC by 26%.

As the number of outstanding memory requests increases, higher memory level parallelism is extracted, hence higher performance. Figure 12 shows how the IPC scales with increasing the number of outstanding requests. Moving from two outstanding requests to 32, we gain, on average, 7% in IPC. The impact of the number of outstanding requests is even greater as the memory latency increases. We study memory latency impact in Figure 13. When memory latency is lower, increasing outstanding requests marginally increases speedup, that is, by 7%. However with a high memory latency, by moving from two outstanding requests to 32, the speedup doubles, that is, from 26% to 54%.

Next we compare the speedup gained with NCOR to that of a full-blown MSHR-based cache. Figure 14 compares the IPC of Runahead execution with NCOR and MSHR-based caches. NCOR achieves slightly lower IPC, less than 4% on average, as it sacrifices memory level parallelism for lower complexity. However, in the case of sjeng, MSHR performs worse. MSHR is more aggressive in prefetching cache lines and in this case pollutes the cache rather than prefetching useful data.

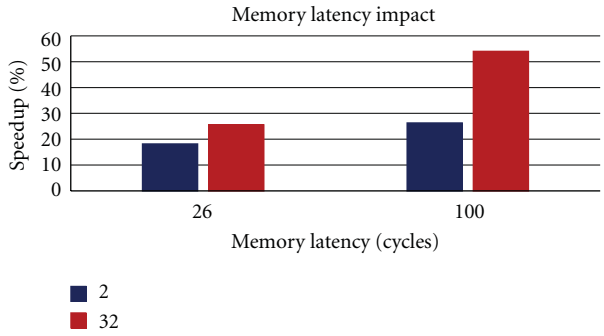

Figure 13: Speedup gained by Runahead execution with two and 32 outstanding requests, with memory latency of 26 and 100 cycles.

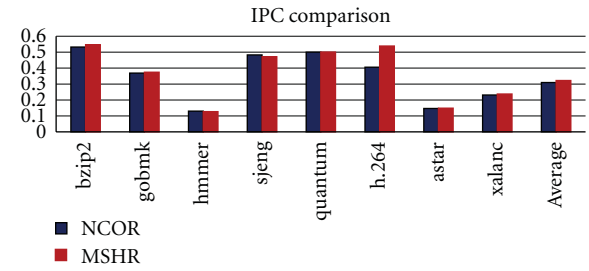

Figure 14: Performance comparison of Runahead with NCOR and MSHR-based cache.

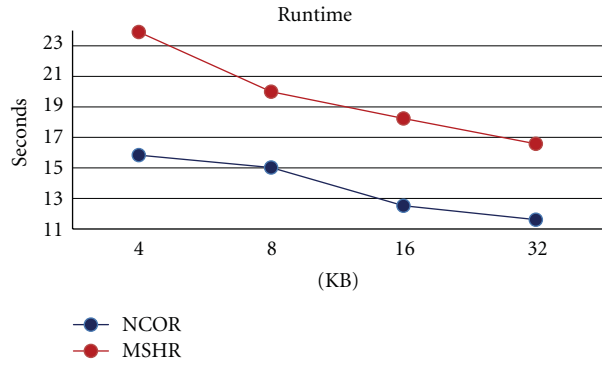

Figure 15: Average runtime in seconds for NCOR and MSHR-based cache.

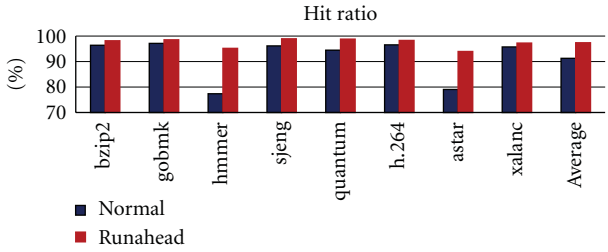

Figure 16: Cache hit ratio for both normal and Runahead execution.

Finally we compare NCOR and MSHR-based caches based on both IPC and their operating frequency. Figure 15 compares the two caches in terms of runtime in seconds over a range of cache sizes. NCOR performs the same task up to 34% faster than MSHR. It should be noted that NCOR with 4 KB capacity performs faster than a 32 KB MSHR-based cache.

7.7. Cache Performance. This section compares cache performance with and without Runahead execution. Figure 16 reports hit ratio for a cache with 32 KB capacity with and without Runahead execution. Runahead improves cache hit ratio, by as much as 23% for hmmer and by about 7% on average. We also report the number of cache Misses Per Kilo Instructions (MPKIs) in Figure 17. Runahead reduces MPKI, on average by 39% as it effectively prefetches useful data into the cache.

7.8. Secondary Misses. Runahead execution tied with NCOR achieves high performance even though the cache is unable to service secondary misses. This section provides additional insight on why discarding secondary misses has little effect on performance. Figure 18 reports, on average, how many times the cache observes a secondary miss (only misses to a different memory block) while in Runahead mode. The graph shows that on every invocation of Runahead mode only 0.1 secondary misses are seen, on average over all benchmarks. Even if the cache was able to service secondary misses, it would have generated only 10 memory requests every 100 times that it switches to Runahead mode. Therefore, discarding secondary misses does not take away a significant opportunity to overlap memory requests. Even for hmmer which experiences a high number of secondary misses, Runahead achieves a 28% speedup as Figure 11 reports. This shows that nonsecondary misses are in fact fetching useful data.

7.9. Write-Back Stall Effect. In Section 6.2.2 we showed that BRAM port limitation requires NCOR to delay write-backs in case the system bus is responding to an earlier cache line read request. Unfortunately in order to study the impact on IPC we need a very accurate DDR-2 model in software simulations which our infrastructure does not include. Alternatively, we study the most pessimistic scenario in which all write-backs coincide with data return of pending cache line reads, resulting in write-back stalls. Figure 19 shows that, even in this worst case scenario, still Runahead execution with NCOR is effective, and on average less than 2% performance is lost.

8. Related Work

Related work in soft processor cache design includes work on automatic generation of caches and synthesizable high performance caches, including nonblocking and traversal caches. To the best of our knowledge, NCOR is the first FPGA-friendly nonblocking data cache optimized for Runahead execution.

Runahead execution requires a checkpoint mechanism for the processor's architectural state. Aasaraai and Moshovos

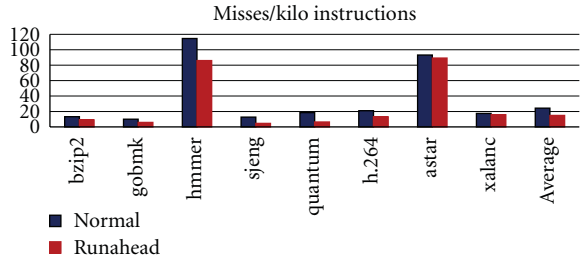

Figure 17: Number of misses per 1000 instructions executed in both normal and Runahead execution.

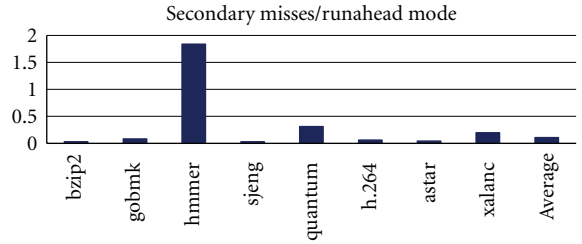

Figure 18: Average number of secondary misses (misses only to different cache blocks) observed per invocation of Runahead executions in a 1-way processor.

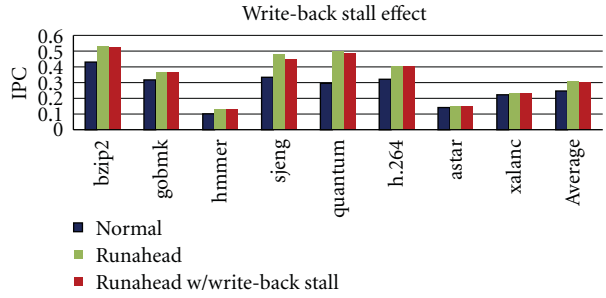

Figure 19: IPC comparison of normal, Runahead and Runahead with worst case scenario for write-back stalls.

proposed CFC, a copy-free checkpointing mechanism which can be used to support Runahead execution [1]. CFC avoids data copying by adding a level of indirection to the checkpointing mechanism. CFC achieves superior performance compared to the conventional method of copying data. CFC, and checkpointing in general, is used in various architectures, for example, out-of-order execution, transactional memory, and Runahead execution.

Yiannacouras and Rose created an automatic cache generation tool for FPGAs [9]. Their tool is capable of generating a wide range of caches based on a set of configuration parameters, for example, cache size, associativity, latency, and data width. The tool is also useful in identifying the best cache configuration for a specific application.

PowerPC 470S is a synthesizable soft-core implementation that is equipped with nonblocking caches. This core is available under a nondisclosure agreement from IBM [10]. A custom logic implementation of this core, PowerPC 476FP, has been implemented by LSI and IBM [10].

Stitt and Coole present a traversal data cache framework for soft processors [11]. Traversal caches are suitable for applications with pointer-based data structures. It is shown that, using traversal caches, for such applications performance may improve by as much as 27x. Traversal caches are orthogonal to NCOR.

9. Conclusion

This work presented NCOR, an FPGA-Friendly nonblocking data cache implementation for soft processors with Runahead execution. It showed that a conventional nonblocking cache is expensive to build on FPGAs due to the CAM-based structures used in its design. NCOR exploits the key properties of Runahead execution to avoid CAMs and instead stores information about pending requests inside the cache itself. In addition, the cache controller is optimized by breaking its large and complex state machine into multiple, smaller, and simpler subcontrollers. Such optimizations improve design operating frequency. A 4 KB NCOR operates at 329 MHz on Stratix III FPGAs while it uses only 270 logic elements. A 32 KB NCOR operates at 278 Mhz using 269 logic elements.

Acknowledgments

This work was supported by an NSERC Discovery grant and equipment donations from Altera Corp. K. Aasaraai was supported by an NSERC-CGS scholarship.

References

[1] K. Aasaraai and A. Moshovos, "Towards a viable out-of-order soft core: copy-free, checkpointed register renaming," in *the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, Prague, Czech Republic, September 2009.

[2] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the International Conference on Supercomputing*, pp. 68–75, July 1997.

[3] K. Aasaraai and A. Moshovos, "An efficient non-blocking data cache for soft processors," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs*, December 2010.

[4] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 81–87, 1982.

[5] Altera Corp., "Nios II Processor Reference Handbook v10.0," 2010.

[6] Altera Corp, "Stratix III Device Handbook: Chapter 4. TriMatrix Embedded Memory Blocks in Stratix III Devices," 2010.

[7] Arcturus Networks Inc, "uClinux," http://www.uclinux.org/.

[8] Standard Performance Evaluation Corporation, "SPEC CPU 2006," http://www.spec.org/cpu2006/.

[9] P. Yiannacouras and J. Rose, "A parameterized automatic cache generator for FPGAs," in *Proceedings of Field-Programmable Technology (FPT)*, pp. 324–327, 2003.

[10] IBM and LSI, "PowerPC 476FP Embedded Processor Core and PowerPC 470S Synthesizable Core User's Manual," http://www-03.ibm.com/press/us/en/pressrelease/28399.wss.

[11] G. Stitt and J. Coole, "Traversal caches: a framework for FPGA acceleration of pointer data structures," *International Journal of Reconfigurable Computing*, vol. 2010, Article ID 652620, 16 pages, 2010.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 716984, 13 pages
doi:10.1155/2012/716984

Research Article

Dynamic Circuit Specialisation for Key-Based Encryption Algorithms and DNA Alignment

Tom Davidson, Fatma Abouelella, Karel Bruneel, and Dirk Stroobandt

ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium

Correspondence should be addressed to Tom Davidson, tom.davidson@ugent.be

Received 30 April 2011; Revised 30 August 2011; Accepted 3 September 2011

Academic Editor: Marco D. Santambrogio

Parameterised reconfiguration is a method for dynamic circuit specialization on FPGAs. The main advantage of this new concept is the high resource efficiency. Additionally, there is an automated tool flow, *TMAP*, that converts a hardware design into a more resource-efficient run-time reconfigurable design without a large design effort. We will start by explaining the core principles behind the dynamic circuit specialization technique. Next, we show the possible gains in encryption applications using an AES encoder. Our AES design shows a 20.6% area gain compared to an unoptimized hardware implementation and a 5.3% gain compared to a manually optimized third-party hardware implementation. We also used *TMAP* on a Triple-DES and an RC6 implementation, where we achieve a 27.8% and a 72.7% LUT-area gain. In addition, we discuss a run-time reconfigurable DNA aligner. We focus on the optimizations to the dynamic specialization overhead. Our final design is up to 2.80-times more efficient on cheaper FPGAs than the original DNA aligner when at least one DNA sequence is longer than 758 characters. Most sequences in DNA alignment are of the order 2^{13}.

1. Introduction

Parameterised configurations are a new concept for dynamic circuit specialization that uses FPGA reconfiguration. It was developed to use run-time reconfiguration (RTR) to dynamically specify the design [1]. This concept is implemented in the *TMAP* tool flow, an alternative to the normal FPGA tool flow. The *TMAP* tool flow allows us to automatically make a run-time reconfigurable design, based on the original design. Its principles and advantages are discussed in section 2. Because this is a new technique, there is still a lot of exploration needed to fully understand how it should be used and what the potential gains are for different applications. This paper shows that in at least two fields, key-based encryption and DNA alignment, substantial gains can be made using parameterised configurations. The *TMAP* tool flow allows us to check very quickly whether or not a certain implementation is suitable for dynamic circuit specialization or not. In Section 3 we will discuss the similarities between this tool flow and hardware/software partitioning that does not use run-time reconfiguration. There, we will also show that using parameterised configurations extends the hardware/software boundary.

To explain how parameterised configuration can be used in encryption applications, we start with a straightforward implementation of the Advanced Encryption Standard. AES is an encryption algorithm detailed by the NIST in [2] and is explained in more detail in Section 4. In this section we will also detail the design decisions for our own AES implementation, k_AES. Next, we will explain how exactly the *TMAP* tool flow is used on this application to make the design run-time reconfigurable. In Section 5 we will discuss the result of applying *TMAP* to k_AES. We show a 20.9% area gain for the k_AES implementation, compared to the normal FPGA tool flow. In addition, we will compare this result with four other, manually optimized, designs, Crypto-PAn [3], Avalon_AES [4], AES128 [5], and AES_core [6]. We will show results. We will also use *TMAP* on the manually optimized designs, and discuss why this does not result in significant gains. Finally, allowing a design to be dynamically specialized will always introduce a specialization overhead, we will discuss this overhead for the AES implementation in this section.

In Section 6 we will expand our results for the AES algorithm to two other key-based encryption algorithms. For the TripleDES [7] encryption algorithm we see a 27.2% LUT-area gain for using parameterisable configuration. The second algorithm is the RC6 [8] encryption algorithm, here we have a 72.7% LUT-area gain.

In addition to the encryption algorithms we will also discuss an example of a specialised string matching algorithm, a DNA aligner. DNA alignment is discussed in Section 7. In this case, we followed a different path than with the AES application. We started with an existing DNA aligner, changed the control flow slightly, and applied the *TMAP* tool flow. The resulting RTR aligner does not show a reduction in the number of LUTs, but does reduce the number of used BRAMs significantly. It changes the resource tradeoff of the design. The RTR aligner can be implemented on a much cheaper FPGA. We will discuss the details of the RTR aligner and its advantages in comparison with the original design in Section 8. In Section 8.3 we will discuss the test-setup and experimental results for the RTR aligner, with concrete numbers on reconfiguration time as measured on an actual FPGA.

In the case of the DNA aligner, the specialisation overhead has a large impact on the design. We will discuss this overhead in detail in Section 9. In this section we will also discuss how to reduce this specialisation overhead using several optimizations.

2. Parameterised Configurations and *TMAP*

FPGAs can be configured to implement any function, as long as there are enough FPGA resources available. The functions on the FPGA are completely controlled by memory. This memory is called the configuration memory. An FPGA configuration of a design describes what values the configuration memory should have to implement the design. Since the configuration memory consists of SRAMs, this memory can be overwritten at run time, and thus the functionality on an FPGA can be changed at run time. This is why FPGAs can be used for run-time reconfigurable implementations of applications and why they are a useful platform for dynamic circuit specialization.

Dynamic circuit specialisation strives to specialise a circuit, at run time, for the specific inputs at that time. This is a form of constant propagation in hardware. This specialisation results in a circuit that is both faster and smaller than the original circuit. Each specialisation takes time, both to generate the specialised configuration and to configure the FPGA. This means that in practice, dynamic circuit specialisation is only useful when some of the inputs of the circuit change less frequently than others. We will call these slowly changing inputs parameters from now on.

There has already been a lot of research on run-time reconfiguration on FPGAs. Most vendors even have their own tool flow for run-time configuration. The Xilinx Modular design flow is one example [9]. This design flow allows the user to use the FPGA hardware in a time-multiplexed way. If one wants to implement two or more applications that never need to be available at the same time, then this design flow can be used to implement both applications on the same FPGA area. This can be done for completely different applications or for different specialisations of the same circuit. At run time, when a new application is needed, the appropriate configuration is retrieved from memory and the FPGA will be (partially) reconfigured.

This design flow could be used for dynamic circuit specialisation, but only in some very specific cases. The problem with this approach is that it scales exponentially with respect to the number of parameter bits. A solution for this could be to generate the specialised circuit at run time. However running the full FPGA tool flow at run time introduces an overhead of seconds to hours, a much too large overhead for most run-time reconfigurable applications.

Parameterised configuration is not the only method or concept for dynamic circuit specialisation. The existing techniques, for both dynamic circuit specialisation and run-time reconfiguration in general, fall in two main categories based on how they approach placement and routing.

The first group avoids placement and routing at run time and so has trouble realising LUT-area gains. Reference [10] is an example of this. They bypass logic cells that become unnecessary after run-time constant propagation. This results in a circuit with better timing behaviour, but this does not decrease the number of LUTs the application needs. Our technique, parameterised configurations does show good area gains.

The second group implements some form of placement and routing at run time. This group of solutions does show LUT-area gains. The problem with this approach is that placement and routing stay NP-complete problems, even if they are simplified. Solving NP-complete problems at run time is very hard because of the limited time available. Our approach takes care of all NP-complete algorithms at compile time. Reference [11] shows work in developing a JIT hardware compiler, aimed at solving simplified place and route problems at run time. Reference [12] is a paper that also falls in this category. They simplify the routing problem by reducing the complexity of the routing architecture. The paper presents a global dynamic router for connecting reconfigurable modules. This global dynamic router allows the connections between modules to be dynamically changed, not the connections within these modules.

The parameterisable configuration concept strives to solve the problems with the existing techniques. A conventional FPGA configuration consists of LUT truth table bits and routing bits. A parameterised configuration is an FPGA configuration where some of the LUT truth table bits are expressed as Boolean functions instead of Boolean values. The Boolean functions are functions of the parameters we discussed earlier. If the parameters change, then the new configuration is generated by evaluating the Boolean functions, based on the current parameter values. By evaluating the Boolean functions a fully specified FPGA configuration can be generated very quickly. This generation only involves evaluating Boolean expressions, and can be done fast enough for use at run time. Actual measurements of this evaluation time will be discussed in Section 9.

A parameterised configuration consists of two parts. A template, the part of the configuration that has fixed Boolean values, and the Boolean functions, the part of the design that is dependent on the parameters. In [1, 13] the *TMAP* tool flow is presented. This tool flow is able to generate parameterised configurations automatically, based on VHDL. This tool flow is an adaptation of the conventional FPGA tool flow for the generation of configurations. Both tool flows are shown in Figure 1.

The conventional tool flow is shown in Figure 1(a). The synthesis tool converts a VHDL description of the design to a gate level circuit. The technology mapper maps this circuit on an LUT-circuit. This LUT-circuit is then given to the placer and the router to place the design on an actual FPGA.

There are several changes for the *TMAP* tool flow, as seen in Figure 1(b). First, the VHDL code is now annotated. These annotations show which input signals should be considered parameters. This annotated VHDL file is then given to an adapted technology mapper, the *TMAP* technology mapper. This mapper also maps the design to an LUT-circuit, but the truth tables of these LUTs can now be dependent on the parameter values, instead of being fixed in a conventional FPGA tool flow. The result is that logic that is only dependent on parameter values will now be expressed as Boolean functions. These Boolean functions determine the truth tables of some of the LUTs. This means the total number of LUTs will be smaller because some of the functionality, originally expressed in LUTs, will now be incorporated in the Boolean functions. This will lead to LUT-area gains. Both the placement and the routing is done with the same tools as the conventional FPGA tool flow.

It is clear that parameterised configurations can be used to implement dynamic circuit specialisation. As said before, the parameterised configuration consists of two parts, the template and the Boolean functions. The template contains all the LUTs that need to be placed and routed. The area needed to implement this template is smaller than using the original toolflow without parameters. All the logic that is only dependent on the parameters is now expressed in Boolean functions and does not require separate LUTs. However, a template in itself does not contain all the information necessary for a fully working implementation. Some of the truth tables are expressed in the Boolean functions. To get a working specialised implementation, these Boolean functions need to be evaluated, based on the actual parameter values.

The above means that all the NP-complete problems, like place and route, can be done at compile time, because they determine the template which does not change at run time. At run time we only need to evaluate Boolean functions, based on the current parameter values, to specialise the circuit. When a parameter changes, the Boolean functions need to be evaluated and the FPGA needs to be reconfigured according to the new results. This introduces an overhead.

A much more detailed overview of parameterised configurations is given in both [1, 13]. Important for us is that the input of the *TMAP* tool flow is annotated VHDL and its output is a working parameterised configuration. This output also includes the code that needs to be executed to evaluate the Boolean functions. The annotations added to the VHDL are very simple and their only role is to tell the tool which of the input signals are parameters. Any of the input signals of a design or parts of a design can be selected as parameters. It is still the job of the designer to chose the parameters, from that point on no user intervention is needed.

The reconfiguration platform, regardless of reconfiguration technique, always consists of two elements, the FPGA itself and a configuration manager. This configuration manager is responsible for evaluating the Boolean functions and supervising the reconfiguration of the FPGA. The configuration manager is generally a CPU. We will use a PowerPC on the FPGA but that is not the only option.

A different perspective on parameterisable configurations is given in the next section. There we will discuss how using *TMAP* can be seen as executing a hardware/software partitioning. This partitioning extends the hardware/software boundary, because run-time reconfiguration is used.

3. *TMAP* as HW/SW Partitioning

The *TMAP* tool flow is an extended hardware/software partitioning method. It is based on the distinction between the slowly changing inputs, the *parameters*, and swiftly changing inputs (regular inputs). Once we have selected the parameters, the tool (*TMAP*) generates a parameterised configuration of the design. This parameterised configuration can then be split up in a hardware and a software part. The hardware part corresponds to the bits in the parameterised configuration that have Boolean values (0 or 1). This will give us an incomplete FPGA configuration, the template. The software part then consists of Boolean expressions in the parameterised configuration that are dependent on the parameters. Evaluating these Boolean expressions will generate the values to complete the FPGA configuration, and is done in software by the configuration manager.

The similarities between *TMAP* and hardware/software partitioning are discussed in detail in [14]. However, it is important to realize that because run-time reconfiguration is used, the hardware/software boundary can be extended compared to traditional hardware/software partitioning.

3.1. The Hardware/Software Boundary. Using FPGAs and run-time reconfiguration the hardware/software boundary can be extended so a larger part of the design can be implemented in software. This is possible because run-time reconfiguration allows the use of specialized circuits. In a conventional hardware/software partitioning, the hardware part would have to be generic to accommodate all possible parameter values. Using FPGAs and run-time reconfiguration, the hardware part can be optimized for specific parameter values.

In a conventional approach to hardware/software partitioning without run-time reconfiguration, such as [15], we would select the slowly changing inputs of the design. Next we identify the functionalities that are only dependent on these parameters. This way the hardware/software boundary is identified. In the next step the boundary is replaced by

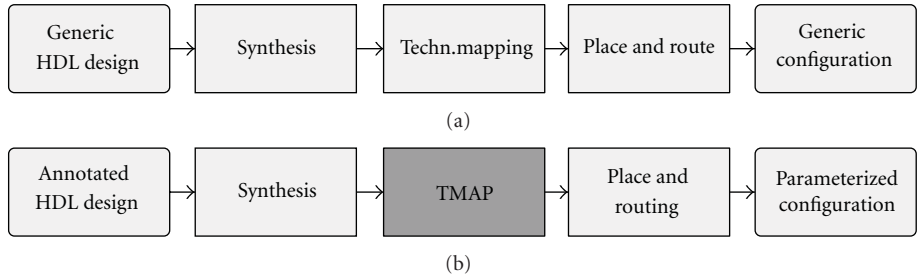

Figure 1: Conventional and *TMAP* FPGA tool flow.

registers and the functionalities that are only dependent on the parameters all get a software equivalent. The actual hardware then consists of the registers and the remainder of the design. The signal values on the boundary will be calculated by the software, and then written to those registers.

When using an approach that includes run-time reconfiguration, several things change. Using the parameterised configuration concept, and the *TMAP* tool flow, we can extend the hardware/software boundary by moving it to the configuration memory instead of adding registers to the design. In addition, because parameterised configurations are used, the hardware will be a specialized circuit that is optimized for specific parameter values. This is in contrast to the generic hardware design of the previous approach. As for the software, in this case it consists of Boolean functions that are generated automatically, based on the hardware functionality that is replaced Figures 2 and 3.

In the next sections of this paper we will discuss in greater detail what the effects are of applying *TMAP* and parameterisable configurations on actual applications. First, in Sections 4 and 5 we will discuss a series of AES implementations, including our own k_AES design. We will explain why using *TMAP* on our design has better gains, compared to other AES implementations. In Section 6 we will also discuss two other encryption algorithms. Finally, we will discuss our tmapped DNA aligner, the RTR aligner in Section 7. Because the specialisation overhead is important in this case, we will give a detailed overview and offer optimisations in Section 9.

4. Advanced Encryption Standard

The complete Advanced Encryption Standard (AES) is described in [2] by the NIST. This document describes the details and mathematical background involved with the different AESs. We will only discuss the hardware implementation of the algorithm, and therefore will only focus on the practical implications of AES.

In its most basic form, AES describes how to encode 128 bits of data, using an encoding scheme based on a specific key value. The Advanced Encryption Standard consists of three separate standards, whose main difference is the length of this key (128, 192, and 256 bits). The three standards are very similar and do not require significantly different hardware implementations. This is why we only discuss the 128-bit AES algorithm in this paper.

The 128-bit AES application can be split up in two parts: data encoding and key expansion. The data encoding explains how the data will be converted into encoded data. For this process we need several values that are key dependent, the round keys, which are generated by the key expansion.

(1) Data Encoding. In the AES algorithm the input data is encoded per 128-bit blocks. We split this data up in 16 bytes, and assign each byte a location in a 4-by-4 state matrix. The encoding part of the AES algorithm consists of a series of bytetransformations that are applied to the state. These transformations are combined in a specific order and grouped in a round, and each round has a round key input. This round key value is different for each round, and all these values are the result of the key expansion.

(2) Key Expansion. The round keys, necessary for the addRoundkey transformation in the different rounds, are derived from the input key. This key is expanded to generate the different round keys. The first round key is the input key, this key is used in the first `addRoundKey()` that is applied to the data input. From the second round key, the round key generation uses some similar transformations as the data encoding process. A more detailed description of the exact transformations used in both the key expansion and the data encoding can be found in [2].

4.1. k_AES. To explain how the *TMAP* tool flow works, we chose an application where the parameter is easy to identify. As described in Section 4, the AES application consists of two main parts: the data encoding and the key generation. Both parts are clearly separated and work, in general, on a different time scale. In most practical applications the key changes much slower than the input data. So, the key input signal is a good parameter choice, we expect a large part, if not all, of the key expansion will be moved to Boolean functions.

To show how parameterised configuration can significantly optimize a design very quickly by making it run-time reconfigurable, we wrote our own AES implementation, *k_AES*. In the next Section 5, this design will be compared with several other AES designs. *k_AES* is almost a direct reflection of the hardware described in the NIST document [2]. The design was deliberately kept fully parallel, and it was therefore very simple to write and test.

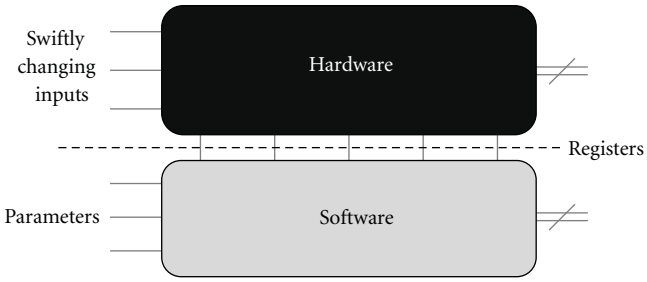

Figure 2: Conventional hardware/software partitioning.

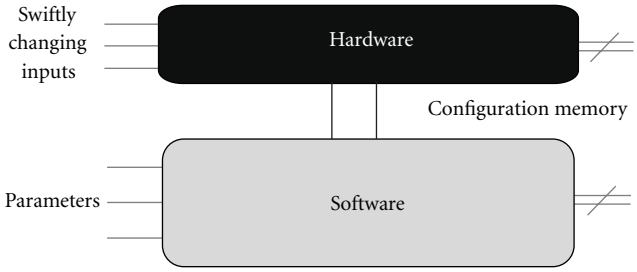

Figure 3: Hardware/software partitioning using *TMAP*.

The design is split up in two main parts, a chain of 10 rounds that implements the encoding process and a chain of 10 k_rounds that take care of the key expansion. The key expansion is combinatorial. The actual transformations are not clock dependent, all the parts are instantiated separately and are only used for calculating one transformation. All this means that we use a lot of area, but conversely encode the data very fast. The throughput is 128 bit of encoded data every clock cycle.

4.2. Making k_AES Run-Time Reconfigurable. Once we have designed our *k_AES* implementation, the next step is to use the *TMAP* tool flow on this design. Since the tool flow is automatic once we have annotated the VHDL code, the main decision designers have to make when using this technique is which signals to select as parameters. Some designs are clearly suited for this approach, like our AES encoder, *k_AES*, and other encryption algorithms that are key based. If a key is used to encode the data, in most cases this key changes very slowly compared to the data.

Once the parameter is selected the *TMAP* tool flow can be used as any other FPGA tool flow is used. It will generate a parameterised configuration in approximately the same amount of time a traditional FPGA tool flow needs to generate an FPGA configuration from a VHDL description. These kind of optimizations are a lot faster than manually optimizing a VHDL implementation by, for example, reducing parallelism in the key expansion. We will discuss the result of applying *TMAP* to *k_AES* in the next section, where we will also discuss several other AES implementations. Applying *TMAP* to these implementations yields much smaller gains. We will briefly discuss the reasons for this observation.

Table 1: The resource usage, in 4 LUTs, of the different AES designs.

	fpga	TMAP	Gain	Throughput (Gbps)
k_AES	45178	35843	20.3%	17.68
Crypto-PAn	37874	37750	0.3%	11.26
Avalon_AES	8448	8448	0%	1.31
AES128	5111	5071	0.7%	0.92
AES_core	5973	5973	0%	0.12

5. Comparison of AES Implementations

In this section we will discuss the results of applying *TMAP* to different AES implementations. Quartus was used as the synthesis tool. As the technology mapper when *TMAP* was not used, we chose the mapper available in ABC [16], called *fpga*. Also, to provide a good basis to compare the different designs, we will only compare LUT-only implementations. This was decided because it is entirely dependent on external factors if you would rather use more BRAMs and less LUTs or the other way around. The DNA aligner in the next section is a good example of *TMAP* influencing the resource tradeoff. This tradeoff will be discussed in much more detail in Section 8.3.

5.1. Area Optimization through Parameterised Configurations. In Section 2 we discussed the concept of parameterised configuration and the *TMAP* tool flow. The result of the *TMAP* tool flow can be seen in Table 1.

The first important result in Table 1 is that, as suggested in Section 4, *TMAP* succeeds in optimizing our original *k_AES* design significantly. The version of *k_AES* mapped by *TMAP* is 20.3% smaller than the design mapped by the original mapper. Basically, the *TMAP* tool flow has removed all the parts of the design that are solely dependent on the parameter, in this case the key input, and has converted those parts to Boolean functions. For the *k_AES* design this means that almost the full key expansion has been moved to software.

However, we do see a difference in clock speeds between both designs. The original *k_AES* has a maximum clock speed of 146.63 Mhz while the tmapped version only has a clock speed of 138.17 Mhz. This is still significantly better than the other AES implementations below, but does mean a 5.7% clock speed decrease. This means we offer up a 5.7% of throughput to reduce the area by more than 20%.

One comment is needed here: the usage of dynamic circuit specialization will introduce a specialization overhead. Each time the key changes, the new FPGA configuration has to be generated and the FPGA itself has to be reconfigured, the total of both types of overhead is called the specialization overhead. The impact of this overhead will be discussed in Section 5.3.

5.2. Comparison to Opencore AES Implementations. To compare our design to other designs, we looked at the publicly available AES implementations found on the opencores.org website. For applying *TMAP* the designs need to be written

in VHDL. We only looked at the "stable" cores that were finished. We found 5 AES implementations that fit these criteria. We had to discarded one implementation because its results with the *fpga* technology mapper generated errors. This leaves us with four AES implementations, the *Avalon_AES* [4], the *Crypto-PAn* [3], the *AES128* [5], and the *AES_core* [6] implementation. The original goals of these applications are less important, and we will only discuss the AES implementations within these applications.

In the first application, "*Crypto-PAn*", an AES-encoding module implements the rounds in a similar way as in our AES implementation, *k_AES*. The main difference is in the key expansion, where the *Crypto-PAn* AES implementation uses a much more complex and more sequential design, involving a state machine. This contrasts to the more parallel *k_AES* implementation, where the key expansion is spread out into 10 k_rounds. For this design, the throughput is the same as for the *k_AES* design, 128 encoded bits every clock cycle. This design can be clocked at 88.043 Mhz.

As Table 1 indicates, we see that, the unoptimized *k_AES* implementation is quite a bit larger than the *Crypto-PAn* design, 45178 LUTs compared to 37874 LUTs. The optimized version of k_AES, however, shows a 5% area gain, compared to the manually optimized *Crypto-PAn* design. From Table 1 it is also clear that both the throughput and the size of the design are worse than the optimized *k_AES* implementation. When comparing *k_AES* and *Crypto-PAn*, the design time and complexity should also be taken into account. The *k_AES* key expansion is significantly more simple and easier to implement and test. As the results show, using *TMAP*, *k_AES* was optimized automatically to the point where it is smaller than more complex and time-consuming implementations.

The second application, "*Avalon_AES*," is even more sequential. Not only in the key-expansion, but also in the encoding part of the AES algorithm. This implementation only consists of one round that is used sequentially to run the full AES algorithm. This design is significantly more complex than the *k_AES*. It also has a much lower throughput. This design needs 10 clock cycles to generate 128 bits of encoded data. The clock speed of this design is 102.76 Mhz. There are significant differences between k_AES and Avalaon_AES. It is very hard to directly compare both designs, because they are both Pareto efficient. Which one is better is decided by external factors.

The third and fourth application, *AES128* and *AES_core*, are both similar to the *Avalon_AES* implementation of the AES algorithm. The *AES128* is a fully sequential AES implementation. It outputs encoded 128 bits of data every 13 clock cycles. This design clocks at 94.20 Mhz. The *AES_core* design is even more sequential, as it works on a byte level instead of the full 128-bit data length. The design size is small, but encrypting 128 bits of data takes more clock cycles. The clock speed of the *AES_core* design is 156.6Mhz. This is the fastest design, but also the one which will take the largest number of clock cycles to encode 128 bits of data.

If we look at the "*TMAP*" column of Table 1, we see that the gains to the other designs for using *TMAP* are a lot less or even nonexistent. The reason is closely linked to how parameterised configurations work. Since we chose the key input as a parameter, all the signals that are dependent on the key input, and not on any other signals, are removed from the design. In the case of a fully parallelised design, like *k_AES*, this results in almost the complete key expansion being moved to software as Boolean functions. In more sequential designs, however, the key expansion is not only dependent on the key input, but also on swiftly changing signals. In these designs, internal timing and hardware reuse make sure that the signals change at a higher rate than the key input. This means a smaller part of the design will be only dependent on the parameters. In essence, a (much) smaller part of the key expansion is moved to software, because most of the hardware used for the key expansion is reused, and thus no longer only dependent on the key input. This is clearly the case in the opencore designs.

These results were attained by comparing 128-bit key AES implementations.

5.3. Specialization Overhead. In the case of AES and similar scenarios the impact of the specialisation overhead is fully dependent on external factors and individual-use cases. In most user-initiated parameter changes an overhead of milliseconds, which is very attainable using *TMAP*, will be unnoticeable to the human user. In this case, dynamic circuit specialization can be used without a large or noticeable impact on the user experience.

In other cases, any delay will be too large. For example, one can easily image a encryption scheme in which the AES key changes for every new 128 bit of input data. In this case the specialization overhead will have a huge impact on the actual operating speed of the design because it will need to be reconfigured very frequently. In most of those cases a delay in the order of milliseconds will never be acceptable.

In Section 9 we will discuss the specialisation overhead further, for the case of the DNA aligner. We will also discuss optimizations to reduce this overhead. But first, in the next section, we will show our results with other key-based encryption algorithms.

6. Other Encryption Algorithms

To check whether our results are transferable to other key-based encoding algorithms, we looked at two other encoding algorithms, "*TripleDES*" [7] and "*RC6*" [8].

The TripleDES algorithm is based on a Data Encryption Standard (DES) encoding/decoding scheme. The data is first encoded with a specific key, then decoded with another key, and finally encoded again with a third key. These separate encoding/decoding steps are done using the DES algorithm. We select the three input keys as parameters for similar reasons as in the AES application. Since a suitable opencore implementation was available for TripleDES [17], we annotated their VHDL description and ran the design through the *TMAP* tool flow without any further adaptations. The result is a 28.7% area gain, from 3584 to 2552 LUTs.

The second algorithm is the RC6 algorithm, it was designed for the AES competition and was one of the finalists. RC6 has a block size of 128 bits and supports key

sizes of 128, 192, and 256 bits. An opencores implementation of this algorithm was also available [18]. Here, we also chose the key as a parameter. Using *TMAP* on this implementation resulted in a 72.7% LUT-area gain, from 4635 to 1265 LUTS.

7. DNA Alignment

DNA alignment is used here as an example of a string matching algorithm, so not a lot of detail on the biological background will be given. The actual algorithm is the Smith-Waterman algorithm, proposed in [19]. The aim of this algorithm is to find the region within two DNA sequences, sequence A and B, where they share the most similarities.

This is done by filling in a score matrix (F). Each element in the score matrix corresponds to one character in sequence A and one character in sequence B. The rows are associated with sequence A, which we will call the vertical sequence from now on. The columns are associated with sequence B, which we will call the horizontal sequence from now on. The element i, j of the score matrix then corresponds to the score for comparing A_i and B_j. The score matrix is filled by solving (1) for each element. Once the matrix is completely filled in, the regions with the most similarities are found by starting from the maximum value in the score matrix and using a trace-back algorithm to find the start of this region,

$$F_{i,j} = \mathrm{Max}\left(F^D_{i,j}, F^V_{i,j}, F^H_{i,j}\right), \tag{1}$$

where

$$\begin{aligned} F^D_{i,j} &= F_{i-1,j-1} + S(A(i), B(j)), \\ F^V_{i,j} &= F_{i-1,j} + \omega, \\ F^H_{i,j} &= F_{i,j-1} + \omega. \end{aligned} \tag{2}$$

F^D takes the matrix element diagonally above and adds a value S. This value S is positive if $A(i)$ and $B(j)$ are the same and negative in case of a mismatch. The values for all possible pairs $A(i)$ and $B(j)$ are described in a substitution matrix. The values in this substitution matrix are based on biological considerations, so we will not go into further detail here. The value taken from this matrix is called the substitution cost.

F^V takes the matrix element directly above and adds a gap penalty, ω. F^H does the same but with the element to the left. The maximum of these three values $(F^D_{i,j}, F^V_{i,j}, F^H_{i,j})$ is then selected and assigned to $F_{i,j}$.

8. RTR Aligner

To get a run-time reconfigurable DNA aligner, we started with an already existing DNA aligner, that was developed for a project (the FlexWare project) funded by the IWT in Flanders and therefore called the Flexware aligner [20]. We will first discuss the structure of this Flexware aligner briefly, then we will discuss the changes and decisions made to turn it into an RTR aligner.

8.1. Flexware Aligner. The Flexware aligner, like most DNA aligners, is based on the observation that the data dependencies in the Smith-Waterman algorithm are west, northwest, and north, as you can see from Figure 4. This means that the algorithm can be HW accelerated by using a systolic array, as shown in Figure 5. Vertical sequence (A) is streamed through the array and each PE calculates an element of the score matrix each clock tick. Once the full vertical sequence has been streamed through the array, the maximum score and its location are known.

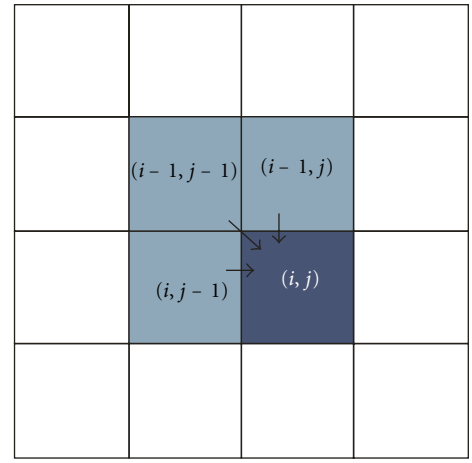

Figure 4: The data dependencies in the Smith-Waterman algorithm [20].

If there are not enough PEs to compare the sequences completely, memory is used to store temporary data. Then, the horizontal sequence is streamed through the array multiple times, each time with different column characters. In Figure 6, you can see the progress of the PEs of this systolic array through the score matrix. Each clock tick a PE calculates the score for its row and column characters.

In general the DNA sequences that are compared are longer (2^{13}) than the number of PEs that fit on an FPGA, at most hundreds on very large FPGAs. This means that more PEs will always speed up the execution.

8.2. RTR Aligner. The parameter selection in the case of the DNA aligner is less straightforward than in the case of AES. However, looking at the inputs of the systolic array we can see that, for each PE, the column character stays constant while the whole vertical sequence is streamed through the systolic array. On the design level the column sequence was chosen as a parameter, for an individual PE this translates to the column character being chosen as a parameter.

Before applying *TMAP* to the Flexware aligner, some changes to the control flow were made so the aligner could be started and stopped without any problems. This is necessary because between configurations the aligner will have to stop working and we need to be able to restart it when the reconfiguration is done. The result of these changes and applying the *TMAP* tool flow is the RTR aligner. We will compare this RTR aligner to the original Flexware aligner in detail below.

In most practical cases, the horizontal sequence will always be longer than the amount of PEs. This means that

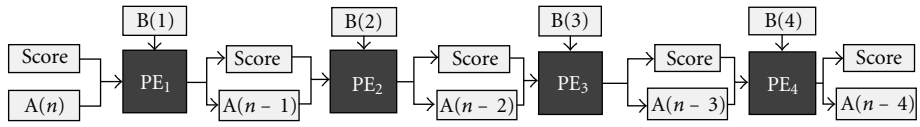

Figure 5: A systolic array.

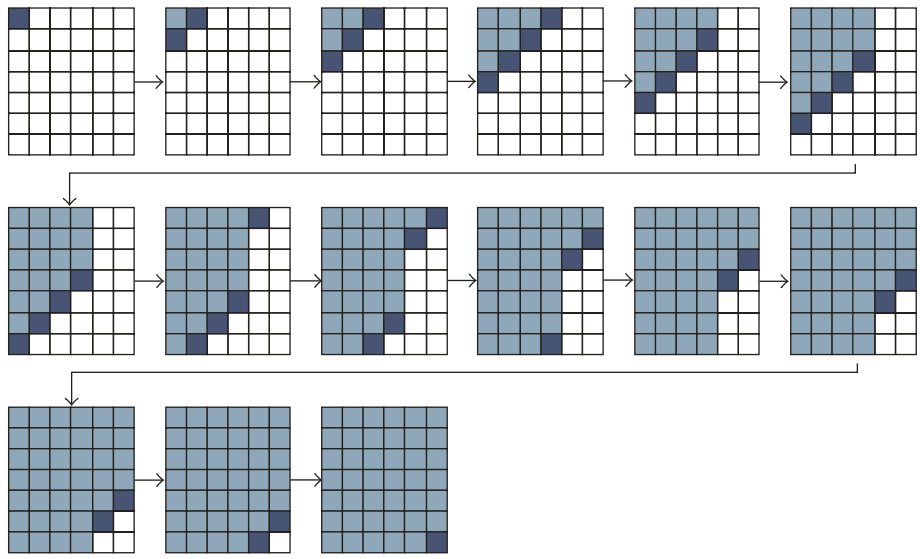

Figure 6: Progress of the PEs in the score matrix during the Smith-Waterman algorithm [20].

the column character of the PEs will change several times during operation. The PEs will need to be reconfigured every time they are started on a new column. This introduces an overhead that will be discussed in Section 9.

From the previous details on the Flexware aligner, and from actual measurements in Figure 8, it is clear that extra PEs are an important factor in the system's overall efficiency. In the original Flexware aligner each PE required its own BlockRAM. This is not the case for the RTR aligner where the number of PEs has no influence on the BRAM usage. In FPGAs with a limited number of BRAMs this can make a huge difference in resource usage. For example, Table 2 shows that the resource usage of the RTR aligner on a Spartan device is a lot better than for the Flexware aligner. It should be noted that only comparing the increase in the number of PEs is not a good comparison. The RTR aligner will be less efficient because it will have to be reconfigured several times. This will be discussed in a lot more detail in the next section, where we will show experimental results based on an actual FPGA implementation.

8.3. Experimental Results. In this section we will compare the operation of the Flexware aligner and the RTR aligner. The next section will discuss the specialization overhead and optimizations based on the data in this section.

As discussed in Section 2, the reconfiguration platform consists of two elements, the FPGA itself and a configuration manager. The configuration manager is generally a CPU.

Table 2: The resource usage in the Flexware and RTR aligner on a Spartan XC3S1600E-4FG484.

	Flexware	RTR	Max. avail.
Slices	3766 (25%)	14708 (99%)	14752
Slice flip flops	2635 (8%)	9515 (32%)	29504
4-input LUTs	6967 (23%)	27382 (92%)	29504
BRAM	36 (100%)	19 (52%)	36
PEs	14	50	—

Our test platform is shown in Figure 7. We used the Virtex II Pro, this FPGA has a PowerPC. The PowerPC is used as the configuration manager. Additionally, the Virtex II Pro has a readily available run-time reconfiguration interface, the HWICAP. We will use this platform to test our design, all the data in this section was collected on the test platform. In all our tests, the OPB/PLB-bus was clocked at 100 Mhz.

8.3.1. Flexware Aligner versus RTR Aligner. In Figure 8 we display the influence of the amount of PEs on the execution time of both the Flexware aligner and the RTR aligner. For the RTR aligner this is the execution time *without the specialization overhead*. As you can see there is a difference between both aligners, but at most it is 18 μs, this is caused by the control flow changes that were made to the RTR aligner to allow for dynamic circuit specialization. It is also clear

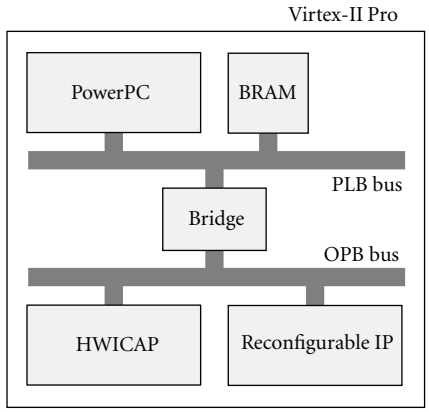

Figure 7: Virtex II Pro run-time reconfigurable platform.

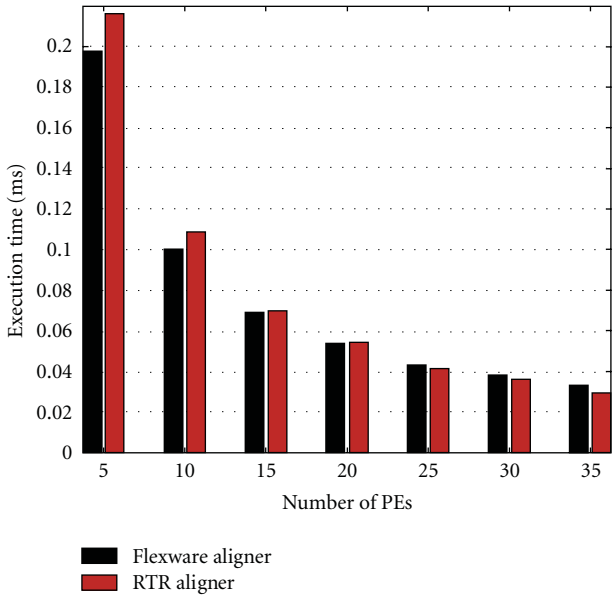

Figure 8: Execution time (ms) for both aligners, without reconfiguration overhead, based on measurements on the test platform when comparing a 188 to a 258 DNA sequence.

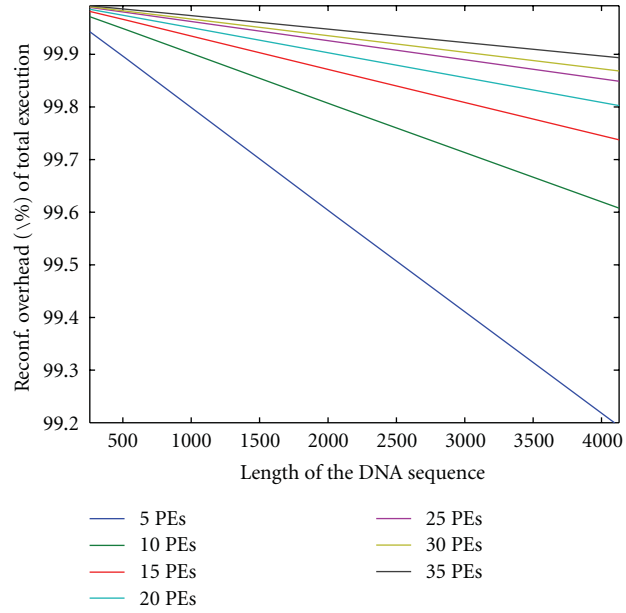

Figure 9: Fraction of the total time spent on reconfiguration, with increasing DNA sequence length, based on measurements on the test platform.

that the relative gain of additional PEs decreases as the total number of PEs in the system increases, for both designs.

For the designs described above the maximum clock speed was measured too. For 4 PEs the RTR aligner clocks at 62 Mhz and the Flexware aligner at 59 Mhz. However, this difference reduces steadily as the area of both designs increases with additional PEs. For designs with 20 or more PEs this difference is less than 1 Mhz. For both designs the clock speed gradually settles down to around 50 Mhz when designs span the complete Virtex II Pro FPGA. There is no clear difference between the two, certainly not for the largest designs.

Now we look at the RTR aligner with the specialisation overhead included. In this case the total execution time is around 3.7 seconds, for any number of PEs, when comparing a 188 to a 258 DNA sequence. It is very clear that this specialization overhead imposes too large a cost on the design, in this case. Using HWICAP reconfiguration makes the system very inefficient compared to the Flexware aligner. In the following optimizations we will reduce this overhead drastically and eventually make the design more efficient than the Flexware aligner. The reason there is no reduction of the total execution time with additional PEs is because the added reconfiguration time of the extra PEs offsets the execution time increase. This will also change with our optimizations.

Figure 9 shows that the fraction of the time spent on actual execution increases with longer DNA sequences, regardless of the number of PEs.

8.3.2. Increasing the DNA Sequence Length. The measurements for Figure 8 were taken comparing sequences of 188 and 258 characters. Those are small sequences in DNA alignment, where sequences in the order of 2^{13} are a lot more common. Increasing the sequence length will impact both designs in a different way.

The Flexware aligner's overall execution time is proportional with the product of the sequence lengths. If one sequence is doubled, the execution time also doubles. If both sequences double the execution time is quadrupled.

Before we can discuss the impact of increasing the sequence length on the RTR aligner, we need to make clear which sequence is chosen as a parameter. Because of the specialization overhead, minimizing the number of reconfigurations is essential. This means choosing the shortlinebreak est sequence as a parameter is clearly the best choice, this can greatly reduce the number of times the PEs need to be reconfigured.

With the shortest sequence as the parameter, increasing the other sequence length will work to the advantage of

the RTR aligner. The number of reconfigurations will stay the same but the time spent on execution will increase, as with the Flexware aligner. This means the relative cost of reconfiguring will decrease.

Another way to see this is that if the nonparameter sequence length increases, a configuration will be used longer before it is changed. From this we can conclude that for the RTR aligner to be working efficiently, the nonparameter sequence needs to be as long as possible. We will go into further detail on this in the next section.

9. Specialisation Overhead

A fundamental property of dynamic circuit specialization is the overhead incurred by this dynamic specialization. There are several possible scenarios, one was discussed with the AES implementation in Section 5.3. In AES the parameter change is decided externally, by a user or a governing program, because its parameter, the key, is an input of the global design.

In the RTR aligner there is a subtle difference. There is also an external parameter decision, a program or user decides which two sequences need to be compared. Internally, however, the application itself will govern the actual parameter change at the input of the PEs. Each time a PE is started on a new column, it will need to be reconfigured.

In this section we will discuss the RTR aligner specialisation overhead in more detail. In the last parts of this section we will also discuss optimizations we have implemented to reduce the specialization overhead and make the RTR aligner more efficient than the Flexware aligner.

In the case of the RTR aligner the specialization overhead has a huge influence on the overall efficiency of the design. This is because the timing of the specialization is a part of the systems normal operating environment. Each comparison of two DNA sequences will entail a certain number of reconfigurations. This number is dependent on the length of the sequences and the number of processing elements. To be clear, each time the last PE finishes its column, all the PEs are reconfigured and a start signal is given to restart the calculations. The next time the last PE has finished its column, a new reconfiguration is done. This goes on until every element of the score matrix has been calculated. The average specialization overhead of one such reconfiguration, which is reconfiguring each PE one time, is displayed in Figure 10. It is clear that if more PEs need to be reconfigured, the specialization takes longer. This increase is invariable of sequence length. Figure 10 also shows that sometimes a larger design will have a smaller specialization overhead. We will discuss this later in the subsection about the reconfiguration overhead.

There are a lot of ways to minimize this specialization overhead, several of the more interesting ones will be applied to the RTR aligner in order to make it more efficient than the Flexware aligner for the case shown in Table 2.

Before we look at optimizing the specialization overhead, we must first determine in which cases it will be used. The RTR aligner uses no BRAMs for its PE implementation, the Flexware aligner uses one BRAM for every PE. Most high-end

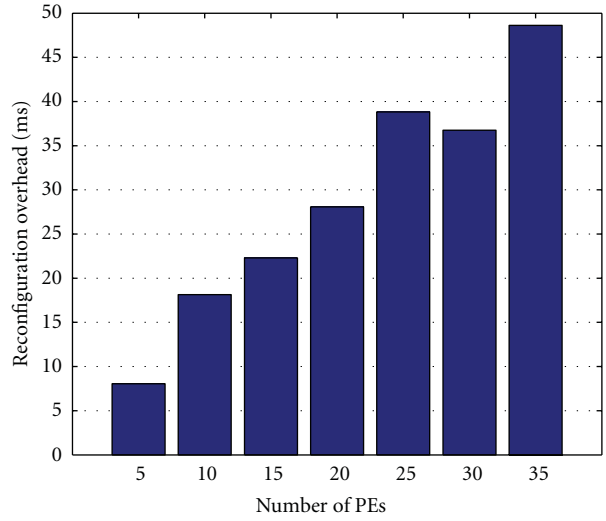

Figure 10: Specialization overhead (ms) needed to reconfigure the PEs of the RTR aligner one time, based on measurements in the test platform.

FPGAs, such as the Xilinx Virtex-line, have enough BRAMs. They generally run out of LUTs before all BRAMs are used by the Flexware aligner. Cheaper FPGAs, however, such as the Xilinx Spartan discussed below quickly run out of BRAMs before all the available LUTs are used, making these FPGAs much less optimal for the Flexware aligner. An example of this is shown in Table 2.

The reason why Spartan implementations are interesting is the price difference between Spartan and Virtex FPGAs. This difference is easily a factor of 10 for equally sized FPGAs.

In [21] we calculated the maximum allowed specialization overhead for the RTR aligner to be as efficient as the Flexware aligner. In the case of the Spartan displayed in Table 2, this resulted in the following upper bound for the specialization overhead of 50 PEs, 455 μ seconds. This is an upper bound for the time needed to reconfigure all the PEs one time. From the measurements of the specialization overhead displayed in Figure 10 it is clear that we already surpass this upper bound for 5 PEs, let alone 50. To improve the efficiency and meet this upper bound, we will discuss several optimizations to the RTR aligner. The SRL optimization discussed below is interesting for any run-rime reconfigurable application that needs to be reconfigured as fast as possible.

Sadly, we have no Spartan XC3S1600E-4FG484 board to test the final design. We will apply the optimisations to the design implemented on the test platform. Based on these results we will discuss the effect of the optimisations on the application. The biggest disadvantage is that the Virtex II Pro does not have enough area to implement 50 PEs. So, in the following we will always discuss the results on the test platform first. Next, we will discuss what the effect of these optimisations is on the use case with the Spartan FPGA.

The first thing we have to realize is that the specialization overhead can be split up in two parts, the evaluation time and the reconfiguration time. The evaluation time is the time

needed for evaluating Boolean functions of the parameters, as described in Section 2. The reconfiguration time is the time needed to write these values to the configuration memory of the FPGA.

From Table 3, we can see that the evaluation time takes about 0.24% of the total specialization overhead and the actual reconfiguration time takes up 99.76% of the total specialization overhead. Also, these numbers are fairly constant when the number of PEs increases.

We can decrease both of these overheads in different ways. Because the reconfiguration time is by far the largest factor in the overhead we will optimize this reconfiguration time first.

9.1. Reconfiguration Time. A large part of the reconfiguration time is introduced by the HWICAP. The way it handles the reconfiguration is not always efficient [22]. It changes the truth tables in three stage. First, it needs to read a full frame from the configuration memory. Then it changes the truth table values. And last, it writes back the full frame. In addition, it writes a padding frame to flush the reconfiguration pipeline [23]. In the Virtex II Pro, each frame contains 320 LUTs with a small header. One can see that changing a single LUT truth table introduces a large overhead. Furthermore, the overhead is dependent on the placement of our design. In a worst case scenario, every LUT of a PE is in a different column. Each PE has 18 LUTs that need to be reconfigured. So, for each PE 18 frames have to be read and 36 have to be written. Figure 10 shows this dependency. If a larger design is placed more efficiently then this can lead to a significant reduction in the total reconfiguration time.

The HWICAP is not the only way to access the configuration memory of the FPGA at run time. In [24] the reconfiguration time is reduced using shift register LUTs (SRLs). An SRL is an LUT whose truth table bits are also arranged as a shift register. This shift register functions like any shift register, with a shift_in, a shift_out, and a clock. Thus, the truth table of an SRL can be changed by shifting in new values through the shift_in input. By connecting the shift_out output of one SRL to the shift_in input of another SRL, a long SRL chain can be formed. This long SRL chain can then be used to change the truth tables of all the LUTs that are part of it.

Such a long SRL chain can then be used to reconfigure the FPGA very quickly, by shifting the new configuration into the truth tables of the connected LUTs. This way the reconfiguration time is reduced to only hundreds or thou-\linebreak sands of clock cycles, instead of hundreds of thousands as with the HWICAP.

The SRL chain(s) can be added on top of the already existing design. The parameterised configurations tool flow determines which truth tables need to be changed. These LUTs are then marked and arranged in an SRL chain that is completely independent of the design. The chain is connected at the input with the configuration manager, it has no output and is only used to change the truth tables of the LUTs. Reference [24] also shows that the impact on the clock frequency of the design of adding these shift register LUT chain(s) is minimal. Important to note is that not every FPGA has this SRL functionality. The Spartan 3 and Virtex II Pro LUTs both have this capability.

Table 3: The specialization overhead, split up in reconfiguration time and evaluation time.

PEs	Reconfiguration time (ms)	Evaluation time (ms)
5	7.9 (99.76%)	0.019 (0.24%)
10	18.0 (99.76%)	0.044 (0.24%)
15	22.1 (99.74%)	0.057 (0.26%)
20	27.8 (99.76%)	0.069 (0.24%)
25	38.5 (99.76%)	0.095 (0.24%)
30	36.4 (99.74%)	0.095 (0.26%)
35	48.2 (99.76%)	0.120 (0.24%)

Table 4: The reconfiguration time, using shift register LUT reconfiguration in the RTR aligner.

PEs	Clock cycles	Theoretical (μs)	Measured (μs)
5	1440	27.91	29.15
10	2880	56.78	59.93
15	4320	84.18	88.93
20	5760	112.31	118.94
25	7200	142.68	151.13
30	8640	172.04	182.23
35	10080	199.83	211.93

The reconfiguration time using SRL reconfiguration can be seen in Table 4. Compared to the reconfiguration overhead using the HWICAP, as seen in both Figure 10 and Table 3, this is a huge decrease. The SRL chain is clocked at the same speed as the design.

The third column shows the theoretical reconfiguration time, assuming one long SRL. We know that we need to reconfigure 18 LUTs in each PE and that reconfiguring 1 bit takes one clock cycle. Based on these numbers we can calculate for each RTR implementation how long the reconfiguration using SRLs should take. In the next column we show our measured results on the test platform. In each instance we see that the measured reconfiguration time is 5% to 6% larger than the expected value. This is because of inaccuracies in our measurements. The reconfiguration time is completely dependent on the SRLs, which are completely deterministic. However, this extra overhead has no impact on our conclusions. The reconfiguration time using SRLs is two orders of magnitude smaller than the run-time reconfiguration overhead using the HWICAP.

In the use case shown in Table 2 we have 50 PEs at a clock speed of 46.25 Mhz. The theoretical reconfiguration time is then 313.04 μs. The upper bound from [21] is 455 μs. We are well below this upper bound, even if we assume that the 6% overhead is not related to inaccuracy in the way we measured. In that case the reconfiguration time is 331.82 μs. Of course, this does not necessarily mean that the total specialisation overhead is below the upper bound. For this to be the case we need to look at the evaluation time too.

9.2. Evaluation Time. The evaluation of the Boolean functions in our run-time reconfigurable platform is done by the PowerPC, shown in Figure 7. However, there are two good reasons to change this in the design on the Spartan. The first reason is that the Spartan has no PowerPC, so we would need to evaluate the Boolean functions on a Microblaze or a custom processor. A second reason to change this is that, in the case of the RTR aligner, the evaluation does not involve any Boolean evaluation. The new truth tables are directly dependent on the parameters, no Boolean expressions are necessary. Because of both these reasons, we propose to use a dedicated hardware block that will function as the configuration manager.

This dedicated hardware block takes a character of the parameter DNA sequence as the input and outputs the bit values that need to be shifted into the SRLs one clock tick at a time. This means, once we have started the dedicated hardware, we can start shifting in the results into the SRL the next clock cycle. The reconfiguration and evaluation can happen at the same time. This will greatly reduce the total specialization overhead. The extra time overhead introduced by the evaluation is then only one clock cycle, compared to the overhead from reconfiguring using SRLs.

Of course the size of this dedicated hardware block will influence how many PEs we can fit on the FPGA. This block uses a very small amount of LUTs (49) but does use one BRAM. In return for adding one BRAM, our total specialization overhead is reduced to the values seen in Table 4, with one additional clock cycle for starting the dedicated hardware block. This is a huge decrease from the specialization overhead shown in Figure 10. We can clearly see the advantages of these optimizations.

An advantage in addition of using the SRLs combined with the dedicated hardware is that we can further reduce our specialization overhead by using two or more SRL chains and an equal number of dedicated hardware blocks. In this case one SRL chain will be split up in two or more SRL-chains that are half as long, this will then reduce the reconfiguration time by half.

Finally, we look back at the upper bound, 455 μs proposed in [21], for the RTR aligner presented in Table 2. Since the Spartan FPGA has SRL capabilities, we can make use of the SRL reconfiguration. Additionally, because we still have unused BRAMs in our RTR aligner, we can use the dedicated hardware block. We do, however, have to remove one PE to make room for the dedicated hardware block. If we lower the number of PEs of the RTR aligner by one, the new upper bound becomes 442.43 μs. For 49 PEs and using both optimizations, our specialization overhead is reduced to 305.14 μs, well below our new upper bound. If we again take an extra 6% of reconfiguration time into account this number becomes 323.45 μs, still below the upper bound.

We can calculate, based on [21] that the RTR aligner will finish its execution earlier than the Flexware aligner on the Spartan XC3S1600E-4FG484, provided the nonparameter sequence is long enough. If we assume the nonparameter sequence is 2^{13} characters, then the RTR aligner will be between 1.15 and 1.29 times more efficient than the Flexware aligner, depending on the length of the parameter sequence.

The RTR aligner is more efficient than the Flexware aligner as long as the nonparameter sequence is longer than 6046 characters. We can decrease this number and increase the overall efficiency by using two or more SRL chains and dedicated hardware blocks. For example, using two SRL chains, and two dedicated hardware blocks, increases the maximum efficiency of the RTR over the Flexware aligner to 1.90. The nonparameter sequence lower bound for efficiency is then lowered to 2824. This trend holds for larger numbers of SRL chains.

Since one PE in the RTR aligner has a size of 369 LUTs, we can fit 7 dedicated hardware blocks in the area used for one PE. This means we can use 7 SRL chains. This improves the maximum efficiency of the RTR aligner to 2.80-times the Flexware aligner for nonparameter sequences longer than 758 characters.

10. Conclusions

In this paper we have shown that parameterised configurations as a method for dynamic circuit specialization and the corresponding *TMAP* tool flow can be used to improve key-based encryption and DNA alignment. In the case of encryption we show an area gain of 20.6% in the case of AES encryption, a gain of 29.3% for a TripleDES implementation, and a 72.7% gain for a RC6 implementation. As an example of a string matching algorithm we chose an existing DNA aligner, which we adapted to the RTR aligner. This RTR aligner runs more efficiently on cheaper FPGAs than the original design. We also showed methods to greatly reduce the specialization overhead introduced by dynamic circuit specialization, and discussed their effects in detail for the RTR aligner. They allow the RTR aligner to be up to 2.80-times more efficient that the Flexware aligner for nonparameter sequences longer than 758 characters.

Acknowledgment

This paper was funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders.

References

[1] K. Bruneel and D. Stroobandt, "Automatic generation of run-time parameterizable configurations," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 361–366, September 2008.

[2] N. I. of Standards and Technology, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," November 2001.

[3] "Opencores.org, the Cryptopan project," http://www.opencores.org/project,cryptopan_core.

[4] "Opencores.org, the Avalon AES project," http://www.opencores.org/project,avs_aes.

[5] "Opencores.org, the AES128 project," http://opencores.org/project,aes_crypto_core.

[6] "Opencores.org, the AES core project," http://opencores.org/project,aes_128_192_256.
[7] D. Coppersmith, D. B. Johnson, and S. M. Matyas, "A proposed mode for triple-DES encryption," *IBM Journal of Research and Development*, vol. 40, no. 2, pp. 253–261, 1996.
[8] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 block cipher," in *Proceedings of the 1st Advanced Encryption Standard (AES) Conference*, p. 16, 1998.
[9] Xilinx Inc., *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, Xilinx Inc., 2002.
[10] N. McKay and S. Singh, "Dynamic specialisation of XC6200 FPGAs by parial evaluation," in *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, from FPGAs to Computing Paradigm*, pp. 298–307, Springer, London, UK, 1998.
[11] E. Bergeron, M. Feeley, and J. P. David, "Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC '08/ETAPS '08)*, pp. 178–192, Springer, Berlin, Germany, 2008.
[12] J. Surís, C. Patterson, and P. Athanas, "An efficient run-time router for connecting modules in FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 125–130, September 2008.
[13] K. Bruneel and D. Stroobandt, "Reconfigurability-aware structural mapping for LUT-based FPGAs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 223–228, December 2008.
[14] T. Davidson, K. Bruneel, and D. Stroobandt, "Run-time reconfiguration for automatic hardware/software partitioning," in *Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 424–429, IEEE Computer Society, Los Alamitos, Calif, USA, 2010.
[15] A. Kalavade and P. A. Subrahmanyam, "Hardware/software partitioning for multifunction systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 9, pp. 819–837, 1998.
[16] "ABC: A System for Sequential Synthesis and Verification," Berkeley Logic Synthesis and Verification Group, http://www.eecs.berkeley.edu/~alanmi/abc/.
[17] N. Seewald and K. Sissell, "Braskem triples income, postpones venezuela projects," *Chemical Week*, vol. 171, no. 22, 2009.
[18] "Opencores.org, the RC6 cryptocraphy project," http://opencores.org/project,cryptography.
[19] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.
[20] F. Project, "Deliverable 2.6: Report on the process of the implementation of the Smith-Waterman algorithm on the FPGA architecture," November 2007.
[21] T. Davidson, K. Bruneel, H. Devos, and D. Stroobandt, "Applying parameterizable dynamic configurations to sequence alignment," in *Proceedings of International Conference on Parallel Computing (ParCo '10)*, p. 8, Lyon, France, 2010.
[22] K. Bruneel, F. M. M. A. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Proceedings of Design, Automation and Test in Europe*, K. Preas, Ed., vol. 4, pp. 964–969, Nice, France, 2009.
[23] O. Blodget, P. James-Roxby, E. Keller, S. Mcmillan, and P. Sundararajan, "A self-reconfiguring platform," in *Proceedings of Field Programmable Logic and Applications*, pp. 565–574, 2003.
[24] B. Al Farisi, K. Bruneel, H. Devos, and D. Stroobandt, "Automatic tool flow for shift-register-LUT reconfiguration: making run-time reconfiguration fast and easy (abstract only)," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '10)*, pp. 287–287, ACM, New York, NY, USA, 2010.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 342625, 14 pages
doi:10.1155/2012/342625

Research Article

A Dynamically Reconfigured Multi-FPGA Network Platform for High-Speed Malware Collection

Sascha Mühlbach[1] and Andreas Koch[2]

[1] *Secure Things Group, Center for Advanced Security Research Darmstadt, Mornewegstr. 32, 64293 Darmstadt, Germany*
[2] *Department of Computer Science, Embedded Systems and Applications Group, Technische Universität Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany*

Correspondence should be addressed to Sascha Mühlbach, sascha.muehlbach@cased.de

Received 30 April 2011; Accepted 22 August 2011

Academic Editor: Marco D. Santambrogio

Malicious software has become a major threat to computer users on the Internet today. Security researchers need to gather and analyze large sample sets to develop effective countermeasures. The setting of honeypots, which emulate vulnerable applications, is one method to collect attack code. We have proposed a dedicated hardware architecture for honeypots which allows both high-speed operation at 10 Gb/s and beyond and offers a high resilience against attacks on the honeypot infrastructure itself. In this work, we refine the base NetStage architecture for better management and scalability. Using dynamic partial reconfiguration, we can now update the functionality of the honeypot during operation. To allow the operation of a larger number of vulnerability emulation handlers, the initial single-device architecture is extended to scalable multichip systems. We describe the technical aspects of these modifications and show results evaluating an implementation on a current quad-FPGA reconfigurable computing platform.

1. Introduction

The significant increase of malicious software (malware) in recent years (see [1]) requires security researchers to analyze an ever increasing amount of samples for developing effective prevention mechanisms. One method for collecting a large number of samples is the use of low-interaction honeypots (e.g., [2]). Such dedicated computer systems emulate vulnerabilities in applications and are connected directly to the Internet, spanning large IP address spaces to attract many different attackers. A number of software applications are available helping in building up honeypot systems. But in addition to having performance limitations in high-speed environments (10+ Gb/s), such software systems also suffer from being compromisable themselves (they can be subverted to attack even more hosts). Given the experience of the Nepenthes research project [3], it is extremely hard to realize an attack surface of millions of IP addresses (such as multiple class B networks) with actively communicating service modules running in software on a single server.

In this context, we have proposed MalCoBox, a low-interaction malware-collection honeypot realized entirely in reconfigurable hardware without any software components in [4]. The core of the MalCoBox system is NetStage, a high-speed implementation of the basic Internet communication protocols, attached to several independent vulnerability emulation handlers (VEH), each emulating a specific security flaw of an application (see Figure 1). We have demonstrated the feasibility of that approach by implementing a prototype on a FPGA platform, fully employing the power of dedicated hardware resources to support 10+ Gb/s network traffic.

With NetStage able to sustain above wire-speed throughput, MalCoBox can easily handle the emulation of multiple class B networks in a single system (Honeynet-in-a-Box), thus avoiding the overhead of administering a honeynet distributed across multiple servers. Beyond the performance aspects, in context of the network security domain, a purely hardware-based approach such as ours has the additional advantage that no general-purpose software programmable

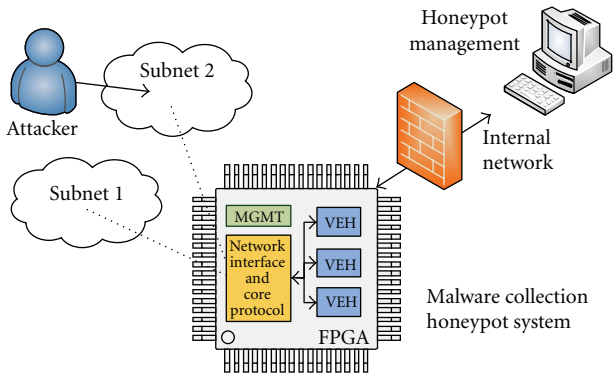

Figure 1: Hardware-based malware collection.

processor is present that could be subverted if the honeypot itself is being attacked.

An important issue for potential MalCoBox users is how the platform can be updated during operation with new or improved vulnerability emulation handlers (VEHs) to react to the changing threat landscape. For an FPGA-based system, the hardware functionality can be altered during operation by using partial reconfiguration (PR). This approach has already been used for network routers in [5]. We now employ the technique in a larger scope to flexibly swap-in new VEHs while the rest of the system stays in operation. The initial discussion presented in [6] is expanded in this work.

Another aspect of great practical interest is the number of different vulnerabilities that can be emulated in *parallel*. The original MalCoBox relied on a single-device implementation of NetStage and was limited to ca. 20 VEHs active in the system. This is a gap to software honeypots, where even the low-interaction variants often support $50 \cdots 100$ different vulnerabilities implemented as scripts in languages such as Perl and Python. While it could be argued that the capacities of individual FPGA chips does increase from each generation to the next (which they do), the larger high-end devices are significantly more expensive per logic cell than the mid-range versions. Thus, it is worthwhile to examine how the MalCoBox capacity can be extended using a multidevice NetStage implementation. This approach has been introduced in [7] and is described in greater detail here.

The paper is organized as follows: Section 2 briefly describes the core architecture components. Section 3 covers details of the ring implementation and elaborates the differences between single-chip and multichip solution. Section 4 continues with a description on the required modifications of the partial reconfiguration strategy. The implementation of the complete system on the BEEcube BEE3 quad-FPGA reconfigurable computing platform [8] is described in Section 5, followed by experimental results given in Section 6. We close with a conclusion and an outlook towards further research in the last Section.

1.1. Related Work. To our knowledge, this is the first implementation of such a honeypot system using pure dedicated hardware blocks. In 2007, Pejović et al [9]. presented an initial concept for a hardware honeypot with RAM-based state machines for the emulations. Unfortunately, they did not give any detailed results on the achievable performance and the possible parallelism. It is likely, however, that the RAM-based state machines could become a bottleneck in high-speed environments due to limited bandwidth. Thus, our architecture contains only dedicated hardware blocks.

In terms of FPGA-based networking, a popular generic platform for research is the Stanford NetFPGA [10], containing an FPGA, four 1G interfaces, and various memories. The platform is the vehicle for a wide spectrum of research, for example, on accelerated switches, routers, and network monitoring devices. Internally, NetFPGA provides a flexible data-path structure into which custom processing modules can be easily inserted. With the widespread use of NetFPGA, a new version supporting 10 G networks is currently being released.

Another related research project is DynaCORE [11]. It consists of a network-on-chip- (NoC-) oriented architecture for a generic reconfigurable network coprocessor, combining general network processing in software with accelerated functions (e.g., encryption) in hardware units. By using current techniques such as partial reconfiguration, the platform can be adapted to different communication situations.

In [12], the authors present a reconfigurable network processor based on a MicroBlaze softcore-CPU extended with multiple exchangeable hardware accelerators. Partial reconfiguration is used to adapt the system to the current network traffic workload. Partial reconfiguration is also employed in [13] to support network virtualization, realizing multiple virtual routers in hardware on a single FPGA.

In contrast to these often packet-oriented approaches, our own research has always been aiming at higher-level Internet (e.g., TCP and UDP) and application protocols. We, thus, have created NetStage [4] as a novel base architecture for our honeypot appliance. NetStage is built around an all-hardware Internet protocol stack implementation (including TCP operation). We have chosen to follow some of the approaches proven useful with NetFPGA, for example, the capability to insert "plug-in" hardware modules at various points in the processing flow. In contrast to DynaCORE, however, we generate a light-weight application specific interconnect between these modules, instead of using a general-purpose, but larger NoC scheme.

2. Key Architecture Components

Figure 2 shows the base NetStage architecture (discussed in greater detail in [4]), including extensions to support dynamic partial reconfiguration (DPR). The architecture provides module slots (Figure 2(a)) into which the partial VEH bitstreams can be loaded. These VEH slots are loosely interconnected with the core system by buffers, allowing all VEHs to have the same external interface (important for DPR). Thus, any VEH may be configured into any of the slots of the same size, with the buffers limiting the impact of brief VEH-level stalls on the system-level throughput.

VEHs share the underlying implementations of the core protocols (IP, TCP, and UDP) in NetStage. These have

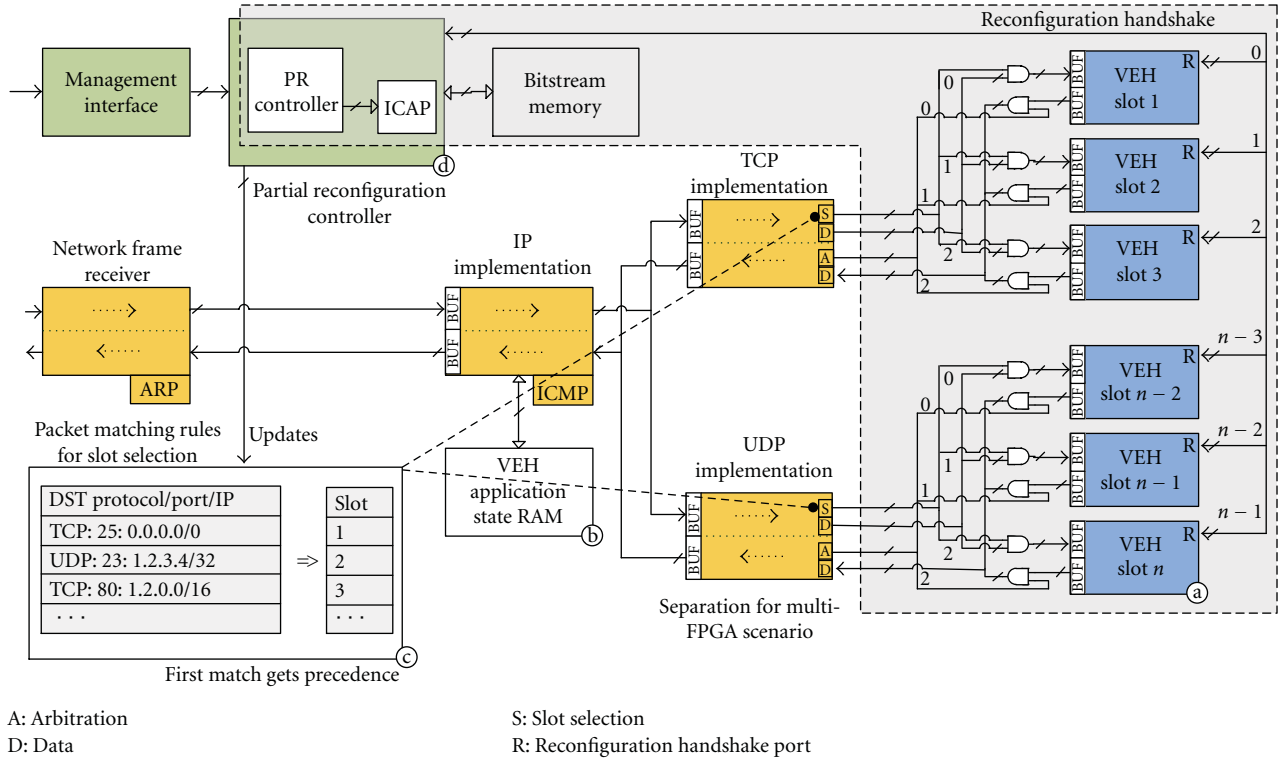

Figure 2: Core architecture of the partially reconfigurable malware collection system.

been very carefully optimized to achieve a throughput of at least 10 Gb/s by using pipeline- and task-level parallelism to keep up with the line rate of the 10 Gb/s external network interface.

In some cases, VEHs have to track session state to generate an appropriate response. NetStage provides a central facility for storing per-connection state (Figure 2(b)). When a packet is passing the IP implementation, the globally maintained state information is attached to the packet in a custom control header which accompanies every packet through the system. The VEH can read this information, act on it, and update the value if necessary. The modified header is written back to the state memory when a response packet (or an empty state write packet) passes the IP implementation on the transmit path. Such a centralized storage is more efficient than attempting to store state in each VEH (which would fragment the capacity of the on-chip memories).

The global VEH application state memory can also be used to save/restore VEH state during reconfiguration to allow the seamless swapping-in of newer (but state-compatible) versions of a VEH.

2.1. Vulnerability Emulation Handler. When a packet has passed through the NetStage core, it will be forwarded to the responsible slot, where the VEH performs the actual malware detection and extraction. Packets are routed to the appropriate slots by means of a routing table (Figure 2(c)) that holds matching rules for the different vulnerability emulations currently active in the system. The table is writable to allow dynamic modification of the actual VEHs used. A basic set of matching rules includes the destination port, destination IP and netmask. The latter allows us to set up separate IP address ranges which use VEHs for different vulnerabilities on the same port (e.g., many handlers will listen on the HTTP port 80).

With the processing speed achievable using reconfigurable hardware, these basic rules could also be extended to directly match packet contents instead of just header fields. Dedicated matching units could search the payloads of packets being forwarded through the NetStage layers and have the matching results available in time for the slot routing decision. However, this would require dynamic reconfiguration of the actual matching units together with the VEHs when they change instead of just writing new values into registers (as in the basic header-only matcher). Since all our current VEHs are selected just based on protocol and port (independently of the payload), we can continue to use the simpler basic approach.

2.2. Management Section. The partial dynamic reconfiguration of VEHs is managed by the partial reconfiguration controller (PRC, Figure 2(d)), which is connected to the FPGAs internal configuration access port (ICAP). On the application side, the PRC is connected to the MalCoBox management interface (either by a PCI express endpoint or a dedicated network link, depending on the selected deployment mode of the system). The PRC is also connected to the individual VEH slots by a number of handshake signals to inform the VEHs about their impending reconfiguration

(for a clean shutdown, etc.) and to check whether the VEH is idle. An attached bitstream memory can hold several partial bitstreams to allow the system to be reconfigured independently of the management station in future implementations.

2.3. Reconfigurable VEHs. The VEHs are dedicated to emulate known application vulnerabilities that may be exploited by malware. As malware is evolving quickly and new vulnerabilities are detected on a regular basis, the VEHs need to be updated as well. For example, when the system is used for research purposes, multiple updates per day would commonly be required if a researcher would like to react to current observations in the network traffic.

In a conventional software-based system, such changes could generally be handled quickly by altering a few lines of code in a script language. However, when using dedicate hardware, even minor changes of the functionality require full logic synthesis, mapping, and place-and-route steps. This effort can be significantly reduced by employing dynamic partial reconfiguration (DPR) to recompile just those VEHs actually affected by the change.

As a secondary effect, DPR also allows the arbitrary combination of VEHs in the system. Otherwise, each specific set of selected VEHs would need to be compiled into its own static configuration.

To support independent partial reconfiguration of any of the VEH slots within the architecture, a wrapper encapsulates the actual VEH implementation module (see Figure 3). This wrapper includes glue logic controlled by the partial reconfiguration controller to disconnect/reconnect all inputs and outputs of the VEH module. This clean separation is essential to avoid introducing errors in the rest of the system when reconfiguring.

The wrapper also contains the send and receive buffers for each module as well as the corresponding buffer management logic. As all the handlers share the same buffer structure, it is more efficient to keep it static than configure it with each VEH. The inputs and outputs of the wrapper are directly connected to the MalCoBox core (see Figure 2).

3. Multidevice Architecture

To extend our system to multiple FPGAs, we will draw the boundaries between the static NetStage core (basic Ethernet and Internet protocol functions) and the dynamically exchangeable VEH slots (see the dotted line in Figure 2). One FPGA acting as Master node holds the network core and the network interfaces, and the remaining other FPGAs, called VEH nodes, contain the individual emulation blocks. The BEE3 platform supports a number of interdevice interconnection schemes. For future scalability independently of the BEE3 architecture, we decided to implement a ring structure. Such rings have already proven useful in multichip systems internally using NoCs [14].

For extending the NetStage-based MalCoBox to multiple devices, a unidirectional ring suffices (Figure 4). The unidirectional ring needs fewer I/O pins on the FPGAs and avoids the increased latency of the bus-turnaround cycles that would be required when running a bidirectional bus over the same pins. Also note that in contrast to the implementations in discussed in prior work, the one described here is using the external SDRAM instead of the internal BlockRAM to temporarily store bitstreams, conserving FPGA resources to allow more VEHs.

3.1. Ring Communication. For the communication on the ring, we use 66 of the 72 available inter-FPGA data lines on the BEE3 which are run in DDR mode, resulting in 132 bits of data per clock cycle [7]. Four bits are reserved for status bits, the remaining 128 form the data transmission word. As data words are sent continuously on the ring for synchronization purposes (even if not actual data needs to be transmitted), a valid flag is used to indicate words holding actual message data. For the separation of individual messages, we use two flags to signal the first and the last word of a message. As the actual byte size of the message is already stored within the internal control header (ICH, see Figure 8) used by NetStage (prefixed to the message body), no special-case processing is required for unused bytes in the last data word of a message. A final flag is used to denote special ring control messages used, for example, to control the DPR process (see next section). In a future extension, this could also be used, for example, to enumerate the ring nodes automatically during initialization. Currently, the destination addresses of the available FPGAs inside the ring are set during compile time.

The ICH-prefixed message is prefixed yet again with a 128 b ring control header (RCH) when it is transmitted between devices. The RCH carries the type information of the message and the destination FPGA. The remaining bits are reserved to implement further control functions in the future.

Since we want to maintain a high bandwidth and low latency even when distributing the architecture across multiple devices, we want to operate the interdevice links at maximum speed. To this end, we use the BEE3-provided global clock to all FPGAs as the ring clock. However, due to different trace lengths and other board-level signal integrity effects, reliable operation at our target frequency of 250 MHz requires a training of the individual FPGAs to the link characteristics. This is done using the technique proposed in [15]. A known training sequence is transmitted between adjacent FPGAs, and the receiver adjusts its delay until it receives a stable pattern from the transmitter.

Two additional signals are required to realize training procedure (see Figures 5 and 6). The Master starts the process by asserting a Sync signal, which is routed around the entire ring. It is used to both initiate training between neighboring FPGAs and to test whether the nodes did configure correctly on start-up (an error is indicated if the Sync sent out by the Master does not match the incoming Sync passed around the ring). Once the receiving FPGA of a synchronization pair has locked on to the training pattern, it asserts its internal Ready signal, which is ANDed with the Ready incoming from its transmitting partner before being output to its receiving partner (the next device on the ring). Once the Master has received an asserted Ready signal passed around the entire

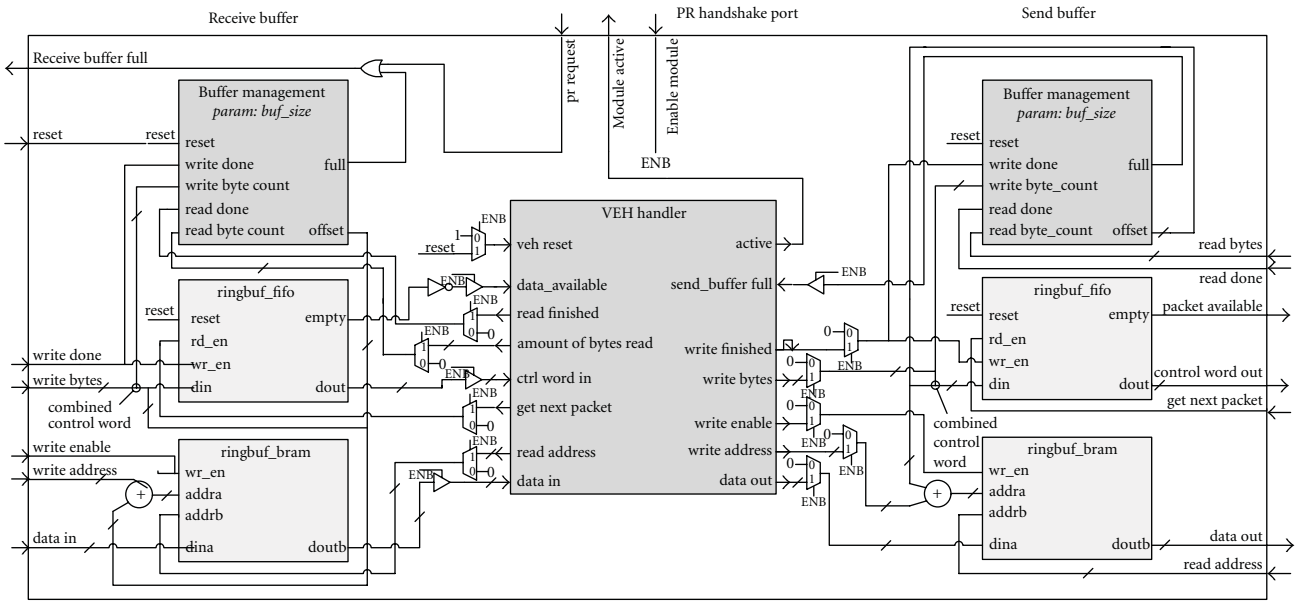

Figure 3: Wrapper encapsulating a vulnerability emulation handler slot.

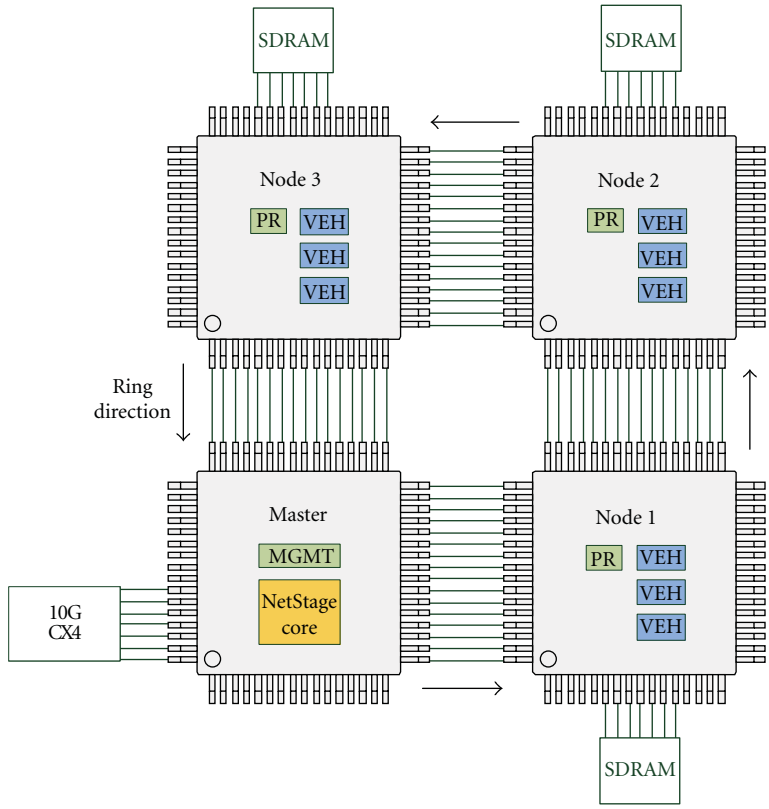

Figure 4: Multi-FPGA network processor in ring topology.

ring, it stops training and releases the ring into normal operation.

The ring thus achieves a transfer rate of 32 Gb/s between nodes, more than sufficient for our current 10 Gb/s network environment. For simplicity, and since we did not experience any data integrity issues in our practical experiments once training completed, we do not perform error detection/correction on the ring communications. However, for long-term production use, CRC/ECC facilities could be added here. As there are still data lines available on the BEE3, this could be easily implemented without affecting the base architecture.

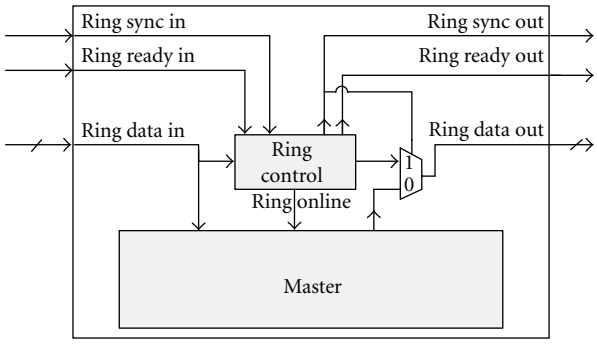

Figure 5: Schematic overview of the master.

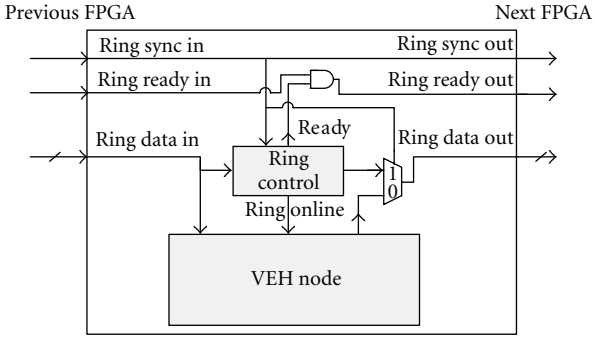

Figure 6: Schematic overview of the nodes.

3.2. Master Node. Beyond the network core and the management section that was already present in the single-chip NetStage implementation, the Master node (see Figure 7) now contains additional logic (Figure 7(a)) to handle the ring communication. In particular, this includes send and receive interface modules as well as the FPGA addressing logic. Note that we do not place any VEHs in the Master node and the currently unused space is intended to be used for future extensions of the NetStage core (e.g., to IPv6). Thus, the Master itself will not be dynamically reconfigured and does not require an internal ICAP controller. However, the Master is responsible for initiating the reconfiguration of the VEH nodes. Thus, the management section in the Master and the configuration controllers in the VEH nodes interact, which is achieved by specialized ring control messages.

The payload data traffic around the ring is organized on two levels. The 32B ICH (see Figure 8) replaces the original protocol headers for a packet with a more compact representation. It also carries the packet-to-handler routing information of a message on the ring in the form of a destination VEH node ID and the VEH slot on that node. Since the destination node ID is already specified in the RCH, this might be seen as redundant. However, the RCH is present only while a packet is transmitted between nodes and stripped from it for node-internal processing. Since we want to give VEHs the ability to transparently forward packets to other VEHs which might also reside on other nodes, they can read/write that destination data in the ICH instead (which, due to alignment reasons, is not efficient for performing ring-level routing).

As we now have multiple destination FPGAs, the Master routing table, which associated packets just with the responsible VEH slot in the single-chip version, needs be extended to hold the destination node ID as well (Figure 7(b)). This is used to build the RCH when the packet is sent out over the ring. To reduce the latency, the process to look up the destination address (Figure 7(c)) is pipelined between the core and the Ring Send module. This can be easily done as packets are not reordered between the two modules.

The Master will silently discard packets not matching any rule in its routing table to conserve bandwidth on the ring links. Furthermore, core IP protocols such as ARP and ICMP are usually handled with low-latency entirely inside the Master, and do not cause ring traffic, either.

3.3. VEH Node. The individual VEHs are housed in the VEH nodes (see Figure 9). For communication with the rest of the system, the VEH nodes need the same ring interface modules (Figure 9(a)) as the Master node. Furthermore, a node-local packet distributor and aggregator (Figure 9(b)) emulate the single-chip NetStage core interface so that VEHs can be attached directly connected to the network core of the single-chip implementation. VEHs are thus portable between the single- and multichip versions.

In contrast to the Master node, the VEH are actually dynamically reconfigured to exchange VEHs. Thus, they do need a PR controller and access to the ICAP. The details of this are described in greater detail in the next section.

The ring receive module in each VEH node checks the type of an incoming ring message and its destination address field and either forwards the packet to the local distributor module, a reconfiguration message to the local PR controller, or immediately inserts the message into the forwarding queue if is intended for another node. VEH response network packets are picked up by the packet aggregator and inserted into an output queue, passing it on around the ring until it reaches the external network connection at the Master node.

4. Partial Reconfiguration

Partial bitstream data is transferred from the management station (usually an external PC or server) to the MalCoBox via the management interface. As the MalCoBox should be able to run as an appliance under remote management, we implemented a stand-alone on-chip reconfiguration interface instead of using the JTAG port together with a software programmer on a host PC.

The underlying protocol used to transmit the bitstream to the MalCoBox consists of the raw bitstream prefixed by a reconfiguration header (see Figure 10). The header contains the bitstream size, the FPGA slot location information, and the rules for the Master node routing table to direct packets to the newly configure VEH.

In contrast to the single-chip implementation [6], in the multidevice scenario, the management interface housed in the Master does not have a direct connection to the partial reconfiguration controller (PRC). Instead, the bitstream is transferred over the ring to the destination VEH node

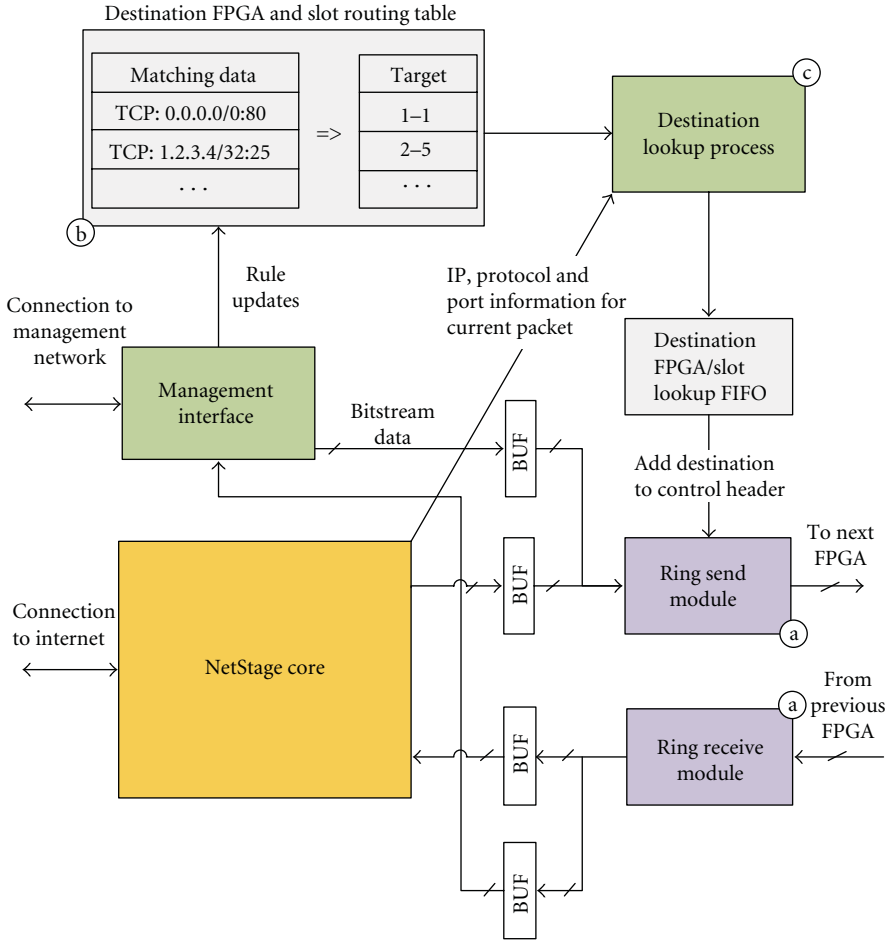

Figure 7: Master node architecture.

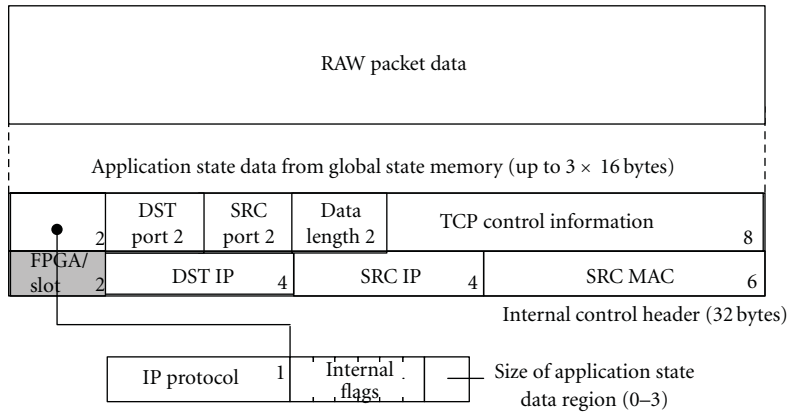

Figure 8: Structure of the internal control header.

FPGA, but the routing rules table still remains inside the Master node. The management interface, therefore, extracts the header information from incoming reconfiguration data requests and updates the routing table, while the raw bitstream data is forwarded to the FPGA specified in the reconfiguration header (see also Figure 7). As the partial reconfiguration process is now distributed across multiple devices, the time between the Master-local routing rule update and the activation of new VEH in a remote node is longer than on the single-chip system. Thus, to avoid misrouting of packets, the new routing rule is explicitly disabled until the VEH is actually ready to accept traffic.

In a VEH node, an incoming bitstream is stored in node-local external DDR-SDRAM memory. One an actual

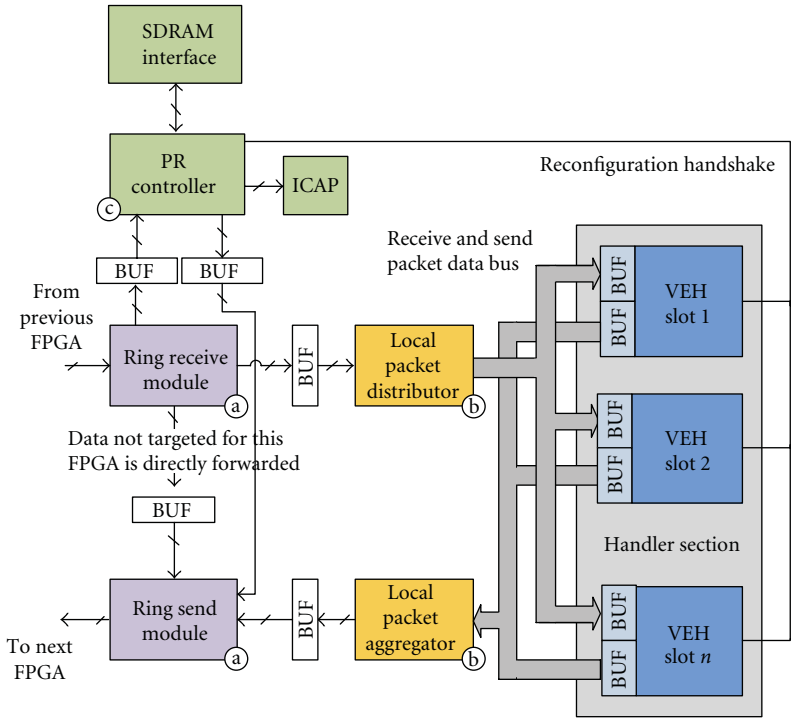

Figure 9: VEH node architecture.

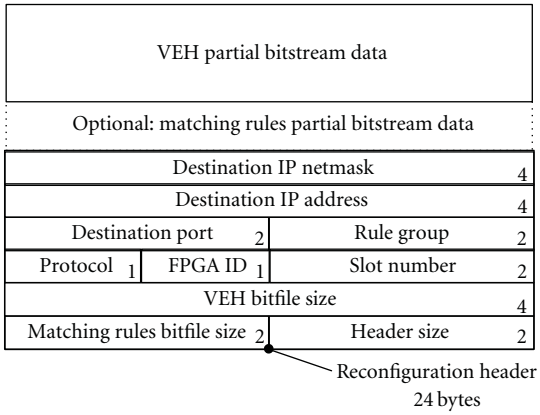

Figure 10: Custom PR header and bitstream data.

reconfiguration is requested, a fast DMA unit retrieves the bitstream data from memory and transfers it at maximum speed to the ICAP configuration interface. This two-step approach could also be used in a later extension to, for example, integrity-check the bitstream for communication errors, or to accept only signed bitstreams [16, 17]. Since the ring communication has proven reliable in our tests, and the management console is trusted, the current prototype does not implement these facilities.

4.1. Partial Reconfiguration Process. The distributed reconfiguration process is performed in the following order.

(1) The rule header of incoming bitstream data is extracted, and the rule table is updated with the new rule (eventually replacing an existing one), having the active flag set to zero.

(2) Incoming bitstream data is forwarded to the corresponding VEH node.

(3) After complete reception of bitstream data, the PRC in the VEH node starts the reconfiguration process.

(4) After completion of reconfiguration, the PRC sends a DONE status to the Master as a ring control message.

(5) The Master management interface receives the message and activates the routing rule so that packets will actually be forwarded.

Network packets and reconfiguration messages (including the bitstream data) share the ring. However, since reconfiguration management is crucial for the reliable operation of the system, these ring control messages receive priority over regular packet transmissions.

Internally, the reconfiguration process in the VEH nodes follows the approach implemented for the single-chip solution [6]. When the node-local PRC receives a reconfiguration request, it initially informs the wrapper of the target slot that the slot is about to be reconfigured. This will stop the receive buffer of the VEH from accepting new packets. The VEH is allowed to process all of the packets held in the buffer at this time, asserting a signal to the PRC on completion. The PRC then deactivates the VEH, and the now inactive VEH is disconnected from the slot wrapper. The actual bitstream data is then read from the DDR-SDRAM and fed into the ICAP. Once reconfiguration is completed, the PRC re-enables the VEH-wrapper connections and allows the new VEH to wake up in its reset state.

5. Implementation

The MalCoBox running on the multidevice NetStage architecture has been implemented on the BEEcube BEE3 FPGA-based reconfigurable computing platform, which is equipped with eight 10 Gb/s network interfaces and four Xilinx Virtex 5 FPGAs (2x LX155T, 2x LX95T). The Master node is realized as one of the smaller SX95Ts to have both of the larger LX155T devices available for VEHs.

Network connectivity is provided by the Xilinx XAUI and 10 G MAC IPs. The network core in the Master runs at the speed of the 156.25 MHz clock of the 10 G network interface. Together with the internal bus width of 128 bit, this leads to a maximum core throughput of 20 Gb/s. This overprovisioning allows us to react to brief stalls in the data flow. These could occur if complex VEHs needed extra time to process a packet (e.g., perform a DRAM lookup) and could not guarantee a steady 10 Gb/s throughput. The 20 Gb/s throughput supported by NetStage thus allows handlers to "catch up" with the normal 10 Gb/s traffic by burst-processing the data accumulated in the buffers at double speed.

To allow for greater design flexibility and allow the execution of more complex VEHs having longer combinational paths, the clock rates of the Master and VEH Nodes can be set independently. Currently, we run the VEH Nodes at 125 MHz, thus supporting burst processing at a rate of 16 Gb/s. This can be easily altered (sped up or slowed down) to match the complexity (delay) of the required VEHs, as long as the 10 Gb/s minimum throughput is always achieved.

The ICAP is operated at 32 b data width and driven by a separate clock to support variable reconfiguration speeds (and thus support experiments with overclocking the ICAP). Management access is implemented as dedicated network interface with a unique MAC address, directly connected to a standard desktop PC or server. The management interface receives bitstream data and control operations over the network using a custom protocol. Perl scripts are used to assemble the appropriate network packets. The DDR2-SDRAM interface in the VEH nodes is realized by a Xilinx MIG core and fully uses the DDR2-SDRAM bandwidth.

The size of all intermodule and slot buffers is set to 4 kB (to hold 2 packets with a maximum size of 1500 B), which is sufficient to assure stall-free operation as the modules generally consume the data at a minimum rate of 10 Gb/s). If many variable-latency VEHs requiring burst processing are employed, the buffer size will need to be increased correspondingly. This has already been done for the ring receive buffers, which are set to 16 kB to provide sufficient headroom to receive bursts of packets on the ring. The size of the global application state memory in the Master node is currently set to 1 Mb of BRAM, which is sufficient to manage the short running sessions (only a few seconds each) that we expect in the honeypot use case. By session, we mean the time interval the attacker needs to check whether the target is vulnerable until the reception of the attack code. For applications requiring more state storage, the data could also be stored in the per-node DDR2-SDRAM available on the BEE3. This would incur longer access latencies, though. These could be avoided on newer hardware platforms by using low-latency external memory technologies such as Bandwidth Engine [18] or Hybrid Memory Cube [19].

5.1. Destination Lookup. The rules that control the message routing to VEHs in different nodes are stored in the destination routing table (see Figure 11) inside the Master node. This table is implemented as BRAM (currently with a size of 1024 rules), to achieve high lookup speeds and flexible scaling. In addition to the routing information, each rule entry contains a rule ID that is used for management purposes and an active flag used during the reconfiguration process. The table supports multiple rules for the same destination VEH (e.g., to let it respond to different IP addresses).

As the destination routing decision is on the critical path with regard to latency, we use a hierarchical approach for lookups. Rules with the same destination port are represented as a linked list, and a CAM is used to retrieve the BRAM address of the list head for a given port (see Figure 12). Then, the individual rules for this port are searched in list order by following the rules' next pointers. Beyond quick lookups, this also ensure that rules with the longest IP address prefix will be matched first. The management process ensures that rules are inserted in the correct order.

For efficiency, we have restricted the CAM size to 256 entries, reasoning that 256 different active ports should be sufficient for most cases. Since the CAM has an 8b wide output, the heads of the per-port rule lists always start in the bottom 256 addresses of the routing table BRAM.

5.2. Example VEHs. To test the system, we have created a number of VEHs emulating different vulnerabilities and applications. In addition to controlling FSMs, the VEHs contain additional logic to perform tasks such as fast parallel pattern matching.

(1) *SIP*. The SIP VEH looks for packets exploiting a vulnerability of the software SIP SDK sipXtapi [20]. The exploit uses a buffer overflow occurring if a SIP INVITE packet contains a CSeq field value exceeding 24 bytes in length. This VEH is based on the UDP protocol.

(2) *MSSQL.* Another UDP-based VEH has a similar structure and is emulating a vulnerable MSSQL 2000 server looking for exploits targeting the resolution service [21]. This exploit was used in the past by, for example, the Slammer worm.

(3) *Web Server.* As a VEH for a further popular application, we implemented a simple web server emulation that contains a ROM with predefined HTML pages to be served to clients. The HTTP headers needed for response generation are also stored inside the ROM. An FSM checks the URL of incoming requests and fetches the corresponding output data to be sent from the ROM. This VEH can be flexibly used, for example, to emulate a login page for a company intranet and monitor

Target FPGA and slot selection table (BRAM)									
Addr.	Protocol	Target FPGA	Target Slot	Port	Netmask	IP addr.	Rule ID	Rule Act.	Next Rule
(10 b)	(8 b)	(8 b)	(8 b)	(16 b)	(32 b)	(32 b)	(16 b)	(1 b)	(10 b)
0	0x06	1	0	80	0x00000000	0x00000000	13	0	0
1	0x06	1	1	25	0xFFFFFFFF	0x53251021	25	1	256
...	...	...	...	...	...	...	...	...	...
256	0x11	2	3	25	0xFFFFFF00	0x10102500	47	1	257
257	0x06	1	1	25	0xFFFFFF00	0x32122500	69	1	0
...	...	...	...	...	...	...	...	...	...

⟶ Linked list to speed up lookups and to maintain IP/netmask prefix order for matching

Figure 11: Layout of the destination lookup table.

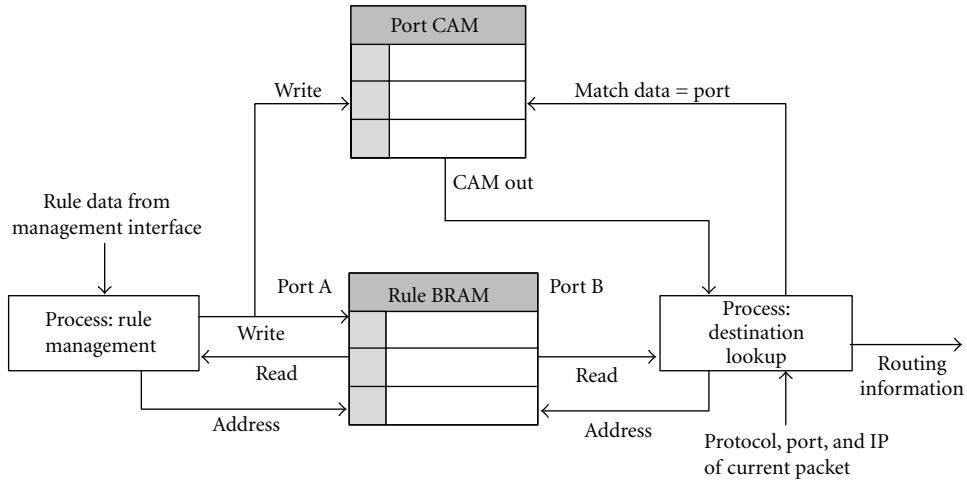

Figure 12: Implementation of the destination lookup process.

attack attempts (e.g., brute force logins) or attacks to the web server itself.

(4) *Mail Server.* As spam is amongst the widespread distribution techniques for malware, we implemented a mail server VEH that accepts incoming mails and pretends to be an open relay server. It contains a FSM that implements the basic SMTP dialog for the reception of mails.

6. Results

The design was synthesized and mapped using Xilinx ISE 12.4, targeting a SX95T for the Master node and both SX95T and the LX155T devices as VEH nodes. Partial reconfiguration was implemented using the latest partial reconfiguration flow available in PlanAhead 12.4 [22]. Each VEH node was configured to include 24 slots. The VEH module slots were placed manually on the FPGA and sized based on the resource usage trends shown by the sample VEH synthesis results. The resulting layout can be seen in Figure 13. To support VEHs with different resource needs (BRAM vs. LUTs), four kinds of slots, differing in the number and types of contained resources (see Table 5), are provided.

As techniques to dynamically relocate bitstreams on the FPGA matrix are not yet production ready, and even research versions have significant limitations (e.g., only support for outdated device families), we have to create separate bitstreams for the all of the different slots a VEH can be placed. In addition to requiring more storage, this also necessitates to run the place-and-route tools multiple times with different area constraints. Each run produces the partial bitfile for a specific VEH-slot combination. However, due to using partial reconfiguration, bitfile sets for different VEHs can be created and used independently, for example, exploiting a multicore server by executing many tool runs in parallel.

System tests were performed by simulation as well as on an actual BEE3 machine connected to a quad-XEON Linux server, sending data to the VEHs at 10 Gb/s. Partial reconfiguration was performed under operator control, loading in new bitstreams via network from the management station.

6.1. Synthesis Results. The synthesis results for all components are given in Tables 1 and 2. For the VEH nodes, we show results only for the LX155T, as the results for the SX95T are very similar (in terms of resource requirements).

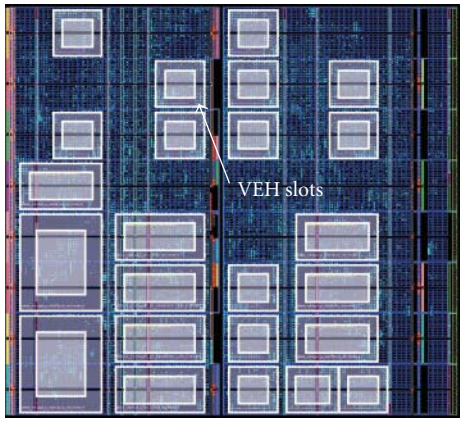

Figure 13: FPGA layout for 24 VEH slots.

The NetStage core on the SX95T Master node requires around 20% of the LUTs and 38% of the BRAMs. The high number of BRAMs is due to several buffers and the global application state memory. The mapped design including IP blocks occupies around 28% of the LUT and 47% of the BRAM resources distributed amongst 50% of the slices. This still leaves sufficient area unoccupied on the SX95T to allow for further extension of the Master node functionality.

The number of available BRAMs is crucial for our platform (due to the multiple buffers). As the SX95T and the LX155T have nearly the same number of BRAMs, these mapping results confirm our decision to put the Master node into the SX95T and leave the large number of LUTs inside the LX155T available for VEHs.

In the VEH node, the ring interface, the partial reconfiguration controller, and the VEH slot interfaces occupy around 20% of the FPGA. This leaves nearly 80% of the LUT resources available for the actual VEH implementations. In practice, the total number of slots per FPGA is limited by the number of BRAMs available to implement the slot buffers (five BRAMs are needed per slot). From these results, we conclude that we could theoretically support up to 36 VEH slots per FPGA on both the LX155T and SX95T.

In comparison to our single-chip implementation, which could hold 20 VEHs together with the NetStage core on a single LX155T device, the multi-FPGA approach is a significant improvement of the total processing power of our platform. When using all three VEH node FPGAs to their full extent, the system supports the parallel operation of 100 VEHs (depending on module size), which should suffice even for very complex honeypot use cases.

6.2. VEHs. Table 3 summarizes the area requirements for the various VEH modules. They are only showing little variation, which is advantageous for putting them into different slots on the FPGA. Amongst them, the SIP VEH requires the most LUTs, as it contains the most complex pattern matching algorithm. Overall, the VEHs are relatively small compared to the device capacity, thus we are confident that our slot numbers are realistic.

Table 1: Synthesis results for Master node components.

Module	LUT	Reg. bits	BRAM
Network core incl. management	12,297	8,884	93
Ring interface	788	1,489	16
Mapped incl. MAC, XAUI and clocks	16,532	13,526	117
In % of SX95T	28	22	47

Table 2: Synthesis results for VEH node components.

Module	LUT	Reg. bits	BRAM
Ring interface	976	2,048	20
PR controller	722	544	4
VEH section with 24 slots (w/o VEHs)	15,494	6,426	120
Total incl. MIG, without VEHs	19,540	12,428	150
In % of LX155T	20	12	70

Table 3: Synthesis results for the VEHs.

Module	LUT	Reg. bits
SIP VEH	1082	358
MSSQL VEH	875	562
Web server VEH	1026	586
Mail server VEH	741	362

6.3. Performance. The actual response time depends on the latency of the platform and the speed of the VEHs. As these numbers are, in turn, highly dependent on the implemented functionality, and the distribution of incoming network traffic, we show numbers for the upper and lower limits. For these experiments, we consider different fill levels of the buffers inside the NetStage core and the ring: all buffers empty (the best case), nearly half full (average case), and nearly full (worst case). For simplicity, we assume that all buffers in the system have the same fill level and that the VEHs are able to actually sustain a speed of 10 Gb/s (possible using the sample VEHs described above).

Table 4 lists the total round-trip times (RTTs) for a 1000 byte request packet that generates a 1000 byte response packet. As the packets have the same size, the time is independent of the ring location of the device holding the measured VEH.

Obviously, the fill level of the buffers inside the ring nodes has a severe impact on the latency, inducing a 10x increase in latency between empty and nearly full buffers. However, as we are currently feeding the system with only one 10 G interface, and the VEHs are all designed for high-speed operation, the buffers should not fill up in practice. We thus expect the average latency of the current system to be between 10–20 μs.

Table 4: Round-Trip Time for a 1000 B packet: overall and per system component.

Buffer Fill Level	Round-Trip Time	Core	Ring	VEH
Empty	5.5 μs	3.6 μs	1.4 μs	0.5 μs
Half	28.7 μs	9.8 μs	17.4 μs	1.5 μs
Full	51.9 μs	16 μs	33.4 μs	2.5 μs

6.4. Partial Reconfiguration Results. Table 5 lists the local reconfiguration time and the total time needed to update a VEH. The local reconfiguration time is measured from the beginning of the DMA transfer between node-local DDR2-SDRAM and ICAP and the end of the reconfiguration process. The remote update time is measured from the first reception of a bitstream packet request at the management interface until the DONE message sent by the node PRC has been received at the Master. This time includes all data transfers of bitstream data from the management station to the system using the dedicated management network interface, sending bitstream messages on the ring from Master to the VEH node and the actual device configuration time. The measurements were made using a 10G Ethernet link at 80% utilization as the management interface. The table lists the theoretical optimum time (assuming raw transmission speeds) as well as the actual measured time on the prototype machine (by using tcpdump).

We also distinguish two cases for when looking at the local reconfiguration times. On a clean shutdown (labeled "w/ SD"), an outgoing VEH is allowed to fully process the packets already present in its input queue. Without a clean shutdown ("w/o SD"), the enqueued packets are discarded when the slot is reconfigured. For the clean shutdown measurement, we assume that the receive buffer of the VEH to be replaced is half full and that the VEH is able to process data at 10 Gb/s (being conservative, since all of our current VEHs can actually handle more).

The time required for cleanly shutting down the outgoing VEH is negligible. Most of the reconfiguration time is actually taken by feeding the bitstream into the ICAP, which limits the overall reconfiguration speed. Thus, a small size of the VEHs is important for fast reconfiguration (see also Section 6.5) and justifies our approach of heterogeneously sized VEH slots (we can configure the 14 smaller VEH slots much faster than the 4 + 4 + 2 larger ones).

When looking at the total reconfiguration time including transfer of the bitstreams from the management station, the use of a 10 Gb/s management link theoretically adds only about 40% of overhead to the raw device reconfiguration time. Even in practice, the actually measured time for reconfiguring the bitstream for the largest VEH slot over the network was just 2.3 ms. Thus, the MalCoBox can be very quickly adapted to changing attack behavior even if the bitstreams are not already present in the node-local DDR2-SDRAM, but have to be fetched from the remote management station.

In [3], Baecher et al. presented results indicating that in their experiment, the honeypot received an average of 3 requests per second, with a peak of 1300 requests per second. Based on this data, MalCoBox should have sufficient headroom for very dynamic attack scenarios even if we assume that the system manages a larger attack surface (e.g., a million IP addresses). This capability will be studied in the upcoming live test of the MalCoBox system at a central Internet router.

Note that for the more common use-case of infrequent VEH updates, a 1 Gb/s Ethernet link to the management station is quite sufficient.

6.5. Impact of Data Path Width. To evaluate the impact of the 128 b data path on the VEH size, we created 64 b versions of the SIP and the Web Server VEHs and compared them to the original 128 b implementation (Table 6). Data path conversion between the NetStage core and the VEHs can be easily performed by the wrappers at the cost of a reduced throughput for the attached VEH.

The area overhead of the 128 b version is roughly 75% for the SIP VEH and 65% for the Web Server VEH. This was to be expected, since these VEHs mostly read data from the input buffer and write data to the output buffer. The area required is thus strongly related to the bus width. Together with the data path area, the BlockRAM usage is also reduced. With 64 b operation, we can now narrow the buffers and only require three BlockRAMs per wrapper instead of five for 128 b VEHs.

Given these results, the number of parallel VEHs in the system could be increased even further by using the smaller datapath width, but only at a loss of per-VEH throughput (8 Gb/s with 64 b width and 125 MHz VEH node clocks). Assuming a heterogeneous traffic distribution across all VEHs, this would not actually lead to a slow-down, since the NetStage core would keep its 20 Gb/s-capable 128 b data path width and *distribute* the traffic across multiple of the smaller-but-slower VEHs. The bottleneck would only become apparent if all traffic was to be directed at a *single* VEH, which then would not be able to keep up with the 10 Gb/s line rate.

7. Conclusion and Next Steps

With this refinement of our MalCoBox system, we have presented a scalable architecture to build a high-speed hardware-accelerated malware collection solution that offers great flexibility through partial reconfiguration and the distribution of VEHs over multiple FPGAs. In the multidevice scenario, the total amount of VEH processing power is significantly improved in contrast to the single-chip implementation, allowing us to implement even large-scale honeynets with a single appliance. A dedicated management interface allows quick updates or replacements of single vulnerability emulation handlers by loading new partial

Table 5: Slot size distribution and reconfiguration time.

Qty.	LUT/BRAM	Bitfile size	Reconfiguration time			
			Local reconfiguration		Remote reconfiguration	
			w/o SD	w/SD	Optimum	Measured
14	1440/0	59 KB	151 μs	154 μs	218 μs	574 μs
4	2304/0	119 KB	305 μs	308 μs	432 μs	1740 μs
4	2304/2	128 KB	328 μs	332 μs	465 μs	1886 μs
2	4864/0	237 KB	607 μs	610 μs	852 μs	2269 μs

Table 6: Synthesis results for 128 b and 64 b VEHs.

VEH	LUT	Reg. Bits
SIP 128 Bit	1082	358
SIP 64 Bit	619	278
Web Server 128 Bit	1026	586
Web Server 64 Bit	663	244

bitstreams, without interrupting the operation of the rest of the system.

Enabled by the high performance of the dedicated hardware, the VEHs actually performing the malware detection and extraction can contain a wide range of functionality. They can embed complex regular expression logic as well as simple request-response patterns, while still reaching the required throughput of 10 Gb/s. Furthermore, our hardware approach is resilient against compromising attacks and significantly reduces the risk of operating honeypots in a production environment.

The presented implementation of the multi-FPGA system on the BEEcube BEE3 quad-FPGA reconfigurable computing platform demonstrated the feasibility of the approach. Operators have a great flexibility to adapt the system to their needs: A tradeoff can easily be made between individual VEH complexity and total vulnerability coverage using many different VEHs just by altering the distribution of VEH slots sizes; throughput and area can be traded off by selecting between VEH implementations with 64 b and 128 b processing widths, and the overall system size can be scaled by selecting either the single-chip or the multi-FPGA approach.

We will continue our work in this area. MalCoBox is planned to be stress-tested in a real production environment connected to the Internet (e.g., university or ISP). From this, we expect to gain valuable insights on how to improve the architecture and its parameters in the future. Furthermore, we will combine the multi-FPGA system with our recent work on self-adapting by dynamic partial reconfiguration based on the observed traffic characteristics. We expect to achieve a platform that exploits many of today's cutting-edge technologies in reconfigurable computing to enable a system presenting maximal flexibility, performance, and security to the user.

Acknowledgments

This work was supported by CASED and Xilinx, Inc.

References

[1] "Internet Security Threat Report, Volume XV," Symantec, 2010, http://www.symantec.com/.

[2] "HoneyD," http://www.honeyd.org/.

[3] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The nepenthes platform: an efficient approach to collect malware," in *Recent Advances in Intrusion Detection*,, vol. 4219 of *Lecture Notes in Computer Science*, pp. 165–184, Springer, Berlin, Germany, 2006.

[4] S. Mühlbach, M. Brunner, C. Roblee, and A. Koch, "MalCoBox: designing a 10 Gb/s malware collection honeypot using reconfigurable technology," in *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 592–595, IEEE Computer Society, 2010.

[5] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable network packet processing on the field programmable port extender (FPX)," in *Proceedings of the ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays (FPGA '01)*, pp. 87–93, ACM, 2001.

[6] S. Mühlbach and A. Koch, "A dynamically reconfigured network platform for high-speed malware collection," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '10)*, pp. 79–84, IEEE Computer Society, 2010.

[7] S. Mühlbach and A. Koch, "A scalable multi-FPGA platform for complex networking applications," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, pp. 81–84, IEEE Computer Society, 2011.

[8] BEEcube, Inc., "BEE3 Hardware User Manual," 2008.

[9] V. Pejović, I. Kovačević, S. Bojanić, C. Leita, J. Popović, and O. Nieto-Taladriz, "Migrating a honeypot to hardware," in *Proceedings of the International Conference on Emerging Security Information, Systems, and Technologies (SECURWARE '07)*, pp. 151–156, 2007.

[10] J. W. Lockwood, N. McKeown, G. Watson et al., "NetFPGA—an open platform for gigabit-rate network switching and routing," in *Proceedingsof the IEEE International Conference on Microelectronic Systems Education: Educating Systems Designers for the Global Economy and a Secure World (MSE '07)*, pp. 160–161, IEEE Computer Society, 2007.

[11] C. Albrecht, R. Koch, and E. Maehle, "DynaCORE: a dynamically reconfigurable coprocessor architecture for network processors," in *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 101–108, IEEE Computer Society, 2006.

[12] C. Kachris and S. Vassiliadis, "Analysis of a reconfigurable network processor," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, p. 187, IEEE Computer Society, 2006.

[13] D. Yin, D. Unnikrishnan, Y. Liao, L. Gao, and R. Tessier, "Customizing virtual networks with partial FPGA reconfiguration," *ACM SIGCOMM—Computer Communication Review*, vol. 41, pp. 57–64, 2010.

[14] S. Bourduas and Z. Zilic, "A hybrid ring/mesh interconnectfor network-on-chip using hierarchical rings for global routing," in *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS '07)*, pp. 195–204, IEEE Computer Society, 2007.

[15] C. Thacker, "DDR2 SDRAM Controller for BEE3," Microsoft Research, 2008.

[16] K. v. d. Bok, R. Chaves, G. Kuzmanov, L. Sousa, and A. v. Genderen, "FPGA reconfigurations with run-time region delimitation," in *Proceedings of the 18th Annual Workshopon Circuits, Systems and Signal Processing (ProRISC '07)*, pp. 201–207, 2007.

[17] Y. Hori, A. Satoh, H. Sakane, and K. Toda, "Bitstream encryption and authentication using AES-GCM in dynamically reconfigurable systems," in *Proceedingsof the 3rd International Workshop on Security (IWSEC '08)*, pp. 261–278, Springer, 2008.

[18] M. Miller, "Bandwidth engine serial memory chip breaks 2 billion accesses/sec," in *Proceedings of the 23rd Hot Chips Symposium*, 2011.

[19] J. T. Pawlowski, "Hybrid memory cube: breakthrough DRAM performance with a fundamentally re-architected DRAM subsystem," in *Proceedings of the 23rd Hot Chips Symposium*, 2011.

[20] M. Thumann, "Buffer Overflow in SIP Foundry's SipXtapi," 2006, http://www.securityfocus.com/archive/1/439617.

[21] D. Litchfield, "Microsoft SQL Server 2000 Unauthenticated System Compromise," http://marc.info/?l=bugtraq&m=102760196931518&w=2.

[22] Xilinx, Partial Reconfiguration User Guide, 2010.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 127302, 10 pages
doi:10.1155/2012/127302

Research Article

Using Partial Reconfiguration and Message Passing to Enable FPGA-Based Generic Computing Platforms

Manuel Saldaña,[1] Arun Patel,[1] Hao Jun Liu,[2] and Paul Chow[2]

[1] *ArchES Computing Systems, 708-222 Spadina Avenue, Toronto, ON, Canada M5T 3A2*
[2] *The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, ON, Canada M5S 3G4*

Correspondence should be addressed to Manuel Saldaña, ms@archescomputing.com

Received 12 May 2011; Accepted 22 August 2011

Academic Editor: Marco D. Santambrogio

Partial reconfiguration (PR) is an FPGA feature that allows the modification of certain parts of an FPGA while the rest of the system continues to operate without disruption. This distinctive characteristic of FPGAs has many potential benefits but also challenges. The lack of good CAD tools and the deep hardware knowledge requirement result in a hard-to-use feature. In this paper, the new partition-based Xilinx PR flow is used to incorporate PR within our MPI-based message-passing framework to allow hardware designers to create *template bitstreams*, which are predesigned, prerouted, generic bitstreams that can be reused for multiple applications. As an example of the generality of this approach, four different applications that use the same template bitstream are run consecutively, with a PR operation performed at the beginning of each application to instantiate the desired application engine. We demonstrate a simplified, reusable, high-level, and portable PR interface for X86-FPGA hybrid machines. PR issues such as local resets of reconfigurable modules and context saving and restoring are addressed in this paper followed by some examples and preliminary PR overhead measurements.

1. Introduction

Partial reconfiguration (PR) is a feature of an FPGA that allows part of it to be reconfigured while the rest of it continues to operate normally. PR has been the focus of considerable research because of the many potential benefits of such a feature. For example, it allows the implementation of more power-efficient designs by using hardware on-demand, that is, only instantiate the logic that is necessary at a given time and remove unused logic. PR also allows the virtualization of FPGA resources by time sharing them among many concurrent applications. Alternatively, a single large application may be implemented even if it requires more logic resources than what a single FPGA can provide as long as the logic resources are not required simultaneously. Fault-tolerant systems and dynamic load-balancing are also potential benefits of PR. All these features make PR attractive for applications in the fields of automotive and aerospace design, software radio, video, and image processing, among other markets.

However, there are many challenges for PR to be more widely accepted, for example, dynamic changing of logic adds an extra level of difficulty to verification performance overheads in terms of designed target frequency and higher resource utilization complex design entry tools and PR CAD flows. A PR design requires the partitioning and floorplanning of the entire FPGA into static regions (SRs) and reconfigurable regions (RRs). The SR does not change during the execution of an application, and RRs may be dynamically reconfigured during the execution of the application. This partitioning has added another layer of complexity to FPGA design. To cope with this increased complexity, Xilinx has released a new partition-based ISE 12 solution that simplifies PR design [1].

By looking at Xilinx documentation and comparing the resources in Virtex-E and Virtex 7 devices, we can see that

in about ten years, FPGAs have increased their resources roughly over 18-fold in LUTs, 14-fold in Flip-Flops, and 31-fold in BRAMs. Furthermore, the number of configuration bits for the XC7V2000T FPGA is just under 54 MB. At this rate, handling partial bitstreams may become more practical than handling entire bitstreams. As FPGA sizes continue to increase, new use case models that were not possible before will now start to emerge, and PR can be an enabling technology of such models.

In this paper, we extend previous work [2] on partial reconfiguration to explore new use cases for FPGAs by using PR within a message-passing system to provide generic, predesigned and prerouted computing platforms based on template designs, where the user can dynamically populate the RRs with application cores. These generic templates can then be modified as needed for a particular application and still be released as application-specific templates. The goal is to relieve the user from the burden of having to design the communications infrastructure for an application and focus solely on application cores. Equally important is to do this in a portable manner across platforms. To this end, we add PR capabilities to the ArchES-MPI framework [3, 4], which provides a communication abstraction layer that enables point-to-point communications between high-end X86 and embedded processors, and hardware accelerators.

The rest of the paper is organized as follows: Section 2 mentions some related work in PR and Section 3 introduces the concept of PR within our message-passing framework. Section 4 describes the synchronization process suggested to perform PR, how to store and restore the current status, and how to generate necessary resets. Section 5 shows the hardware platform used to run the tests. Section 6 explains the software flow to create and handle partial bitstreams. Experimental results are shown in Section 7. Section 8 describes an example where four applications reuse the same template bitstream. Finally, Section 9 presents some concluding remarks.

2. Related Work

There has been abundant research on PR in the last decade with much of it focusing on specific aspects of PR such as CAD flows [5], scheduling [6], communications [7], configuration time evaluation frameworks [8], and core relocation [9]. Our long-term goal is to use PR to implement an entire, practical and generic framework that allows hardware designers to create generic and application-specific template platforms that follows a higher-level parallel programming model that is easier to understand by software developers. This paper presents a step in that direction, providing a working framework that includes the software interface, CAD flow, network-on-chip (NoC), and message-passing layer. Future work will focus on adding and optimizing different aspects of the PR framework.

One of the PR aspects is the state store and restore capability before and after PR takes place. An approach for doing this is by reading back parts of the FPGA configuration memory as described in [10]. However, this approach assumes deep knowledge of the bitstream format and it is strongly dependent upon the FPGA architecture, which is too low level for most developers. Additionally, unnecessary data is read back, which increases the storage overhead and the time to perform the context switch. Our approach is to let the user decide what to store and restore using the message-passing infrastructure making the solution more portable and higher-level at the expense of some additional design effort. Research has been done to provide platform-independent PR flows [11]. Similarly, our goal is to achieve more portable PR systems by using our existing MPI infrastructure.

The BORPH research project [12] follows similar goals with our work in the sense that it aims at simplifying the use of FPGAs by inserting high-level and well-known abstractions. PR has been added to BORPH as a way to dynamically configure a hardware engine at runtime and treat it as an operating system process. Communication with the engine is achieved via file system calls, such as open, read, write, and close. In our work, we treat the hardware engines as MPI processes (not operating system processes) and communication is achieved via MPI function calls, such as MPI_Send and MPI_Recv. Our approach is more suitable for parallel programming as it is based on a standard designed specifically for such a purpose.

In most prior art in this field, the configuration controller is an embedded processor (MicroBlaze or PowerPC) using a fixed configuration interface (ICAP or SelectMAP) [13] and the controller also generates the local reset pulse after PR. In our framework, the X86 processor controls the configuration and the actual configuration interface is abstracted away from the user, who does not need to know which interface is used. Also, in our approach, an embedded processor is not necessary to issue the local reset, rather it is generated by using the underlying messaging system already available.

3. Message Passing and PR in FPGAs

The MPI standard [14] is a widely used parallel programming API within the high-performance computing world to program supercomputers, clusters, and even grid applications. Previous work presented TMD-MPI [4] and proved that a subset implementation of the MPI standard can be developed targeting embedded systems implemented in FPGAs. TMD-MPI was initially developed at the University of Toronto and now it is supported as a commercial tool known as ArchES-MPI. It allows X86, MicroBlaze, and PowerPC processors as well as hardware accelerators to all exchange point-to-point messages and work concurrently to solve an application. By using MPI as a communications middleware layer, portability across multiple platforms can be achieved easily. In particular, platforms that include FPGAs have extra benefit from a portability layer as hardware can change all the time, either because of the nature of the FPGA itself or due to changes in the boards that FPGAs are placed on. But most importantly, ArchES-MPI provides a unified programming model and proposes a programming

paradigm to the developer that facilitates the implementation of heterogeneous, multicore, multiaccelerator systems.

The use of PR within our message-passing framework is to allow the creation of predesigned and prerouted platforms that can be distributed as templates to the end users with the purpose of simplifying the system-level design process. By using PR it is possible to create an array of RRs that can be populated with user application cores, known as reconfigurable modules (RMs) at run time. Figure 1 shows some examples of these generic designs for one, four, and eight RRs. Application cores can also be placed in the SR and the ArchES-MPI infrastructure transparently handles the communication between the cores regardless of the region they are placed in.

A typical MPI program has multiple software processes, each with a unique ID number known as the rank. In ArchES-MPI there are software ranks running on processors and hardware ranks running as hardware engines. One MPI hardware rank can have multiple RMs. This means that the rank number assigned to it does not change during the execution of the entire application, just its functionality depending on the RM currently configured. There is another situation (not covered in this paper) where a new RM requires a change in the assigned rank number, which requires the NoC routing tables to be updated dynamically.

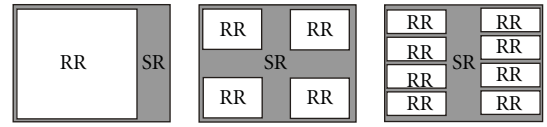

Figure 1: Layout of template-based, generic bitstreams for one, four, and eight reconfigurable regions (RRs) and the static region (SR).

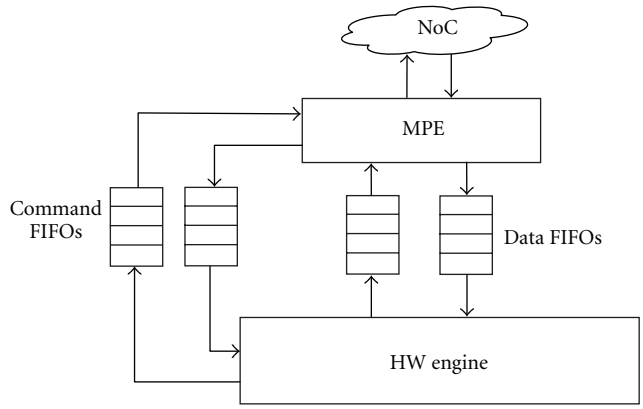

Figure 2: Connection of the application hardware engine and the MPE.

3.1. Message-Passing Engine and Reconfigurable Regions. To processors, ArchES-MPI appears as a software library that provides message-passing capabilities. Hardware engines must use a hardware block called the MPE (message-passing engine), which encapsulates in hardware some of the MPI functionality. The MPE provides the hardware equivalent to MPI_Send and MPI_Recv to a hardware engine in the FPGA. It handles unexpected messages, processes the communication protocol, and divides large messages into smaller size packets to be sent through the NoC. As shown in Figure 2, the MPE is connected between the hardware engine and the NoC. The MPE receives the message parameters and data from the hardware engine, such as the operation (whether it is sending or receiving a message), the destination node id (rank of the process in the MPI environment), the length of the message, and an identification number for the message (the Tag parameter in a normal MPI send/receive operation). After this, the MPE will handle the communications through the NoC with the destination rank. The hardware engine and the MPE form a single endpoint in the message-passing network. The interfaces between the MPE and the hardware engine are four FIFOs (two for commands and two for data), making integration easy. These FIFOs can be asynchronous allowing the hardware engine to operate at different frequencies than its associated MPE.

Based on where the MPE is placed relative to the RR, we can have three possible scenarios. Figure 3(a) shows the MPE placed outside the RR, and it is connected to a user wrapper that controls the MPE and the actual RM through control signals. The RM is a block instantiated within the wrapper. The RM block can be thought of as the main computational pipeline and the wrapper as a higher-level, application-specific controller. One of the wrapper's duties is to control the PR synchronization process (described in the next section). In Figure 3(b), the MPE is also placed outside the RR but with no wrapper (or a very small and generic wrapper) and it is connected directly to the RR. This means that the entire RR is reserved for a single and self-contained RM that must directly control the MPE and handle its own PR synchronization process. This self-contained scenario is more generic than the wrapper-based scenario because it does not contain an application-specific wrapper that may not work with a different application. From the implementation point of view, the user can still design a specific wrapper for a particular application but it would have to be placed in the RR and not in the SR. Finally, Figures 3(c) and 3(d) show the scenario where there is no MPE in the SR. This scenario gives the user the opportunity to implement a sub-NoC that may contain many more ranks or user-defined bridges for off-chip communications such as an Ethernet-based bridge. For brevity, in this paper we only focus on the first two scenarios.

4. Partial Reconfiguration Synchronization

Certain applications may need a way to store the current status of registers, state machines, and memory contents before PR takes place and restore them once the RM is configured back again. This is analogous to pushing and popping variables to a stack before and after a function call. A local reset pulse may also be required to initialize the newly partially reconfigured module to drive registers or state machines to their quiescent state; the global reset cannot be used for this.

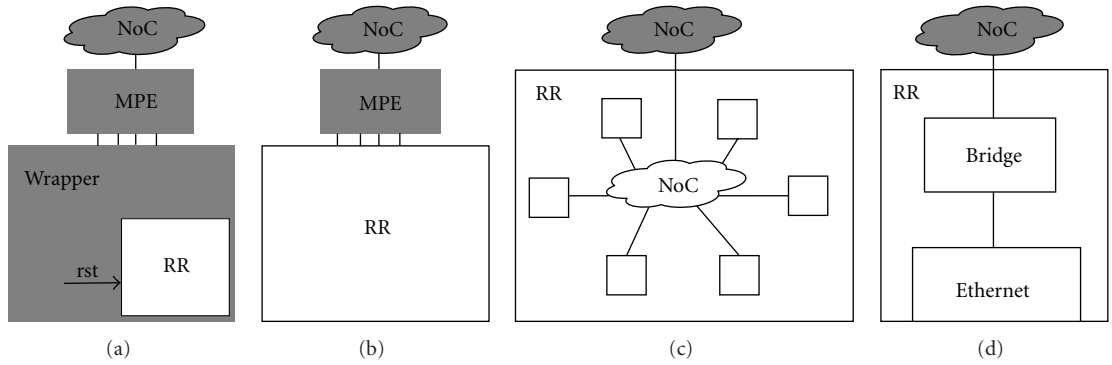

Figure 3: (a) Wrapper-contained RR, (b) Self-contained RR, (c) and (d) no-MPE RRs. (Gray means static and white means dynamic).

4.1. Status Store and Restore. The requirement to store and restore the current status of an RM is solved by sending explicit messages between the configuration controller (a rank within the MPI world) and the rank that is going to be partially reconfigured. Since any rank in MPI can initiate a message, there can be two options to start the PR process based on the initiating entity: the *processor-initiated PR* and the *RM-initiated PR*. This is shown in Figure 4. In the processor-initiated PR, the X86 processor (Rank 0) acts as a master and initiates the reconfiguration synchronization by sending an explicit message ("cfg") to RM_A (Rank 1), which acts as a slave waiting for this message and reacting to it. RM_A then sends an "OK-to-configure" message back to Rank 0 when it is safe to perform the configuration process. The payload of this "OK-to-configure" message can be used to save the current status of RM_A. When Rank 0 receives the "OK-to-configure" message it stores the status (if any) in memory and proceeds with the partial configuration process of RM_B (also Rank 1). Once this is done, a "configuration-done" message is sent to the newly configured module RM_B with the previously stored status data (if any), which is received and used by RM_B to restore its state prior to PR.

The second synchronization option is when the RM is the master and the X86 is the slave. In this case, when a certain condition is met, RM_A sends a "request-to-configure" message along with the status data to Rank 0, which stores the status data and proceeds with the reconfiguration of RM_B. After this is done, Rank 0 sends the "configuration-done" message to the newly configured RM_B along with the status data to restore its state. The user-defined condition that triggers the PR entirely depends on the application.

4.2. Local Reset. The reset issue can be handled easily in the wrapper-contained scenario mentioned in Section 3.1. Since the wrapper controls the MPE it knows when the PR synchronization messages are sent and received and it can assert the reset signal of the RR while PR takes place and release it after the "configuration-done" message is received. In the wrapper-contained scenario the wrapper is placed in the SR and it is aware of when PR has occurred. In contrast, in the self-contained scenario all the application logic is in the RM and it is all being configured, therefore, a different reset mechanism must be implemented. In multi-FPGA systems this is an inherently distributed situation where the RR might be in a different FPGA than where the configuration controller (e.g., embedded processor) is located. Everything has to be done at the endpoints using messagepassing otherwise a reset signal might need an off-chip wire between FPGAs, and as many wires as there are RRs per FPGA, which is not practical.

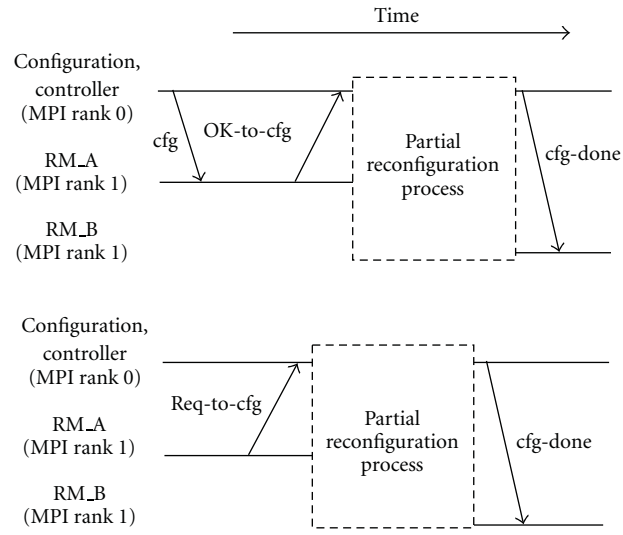

Figure 4: Explicit message synchronization between PR controller and RMs.

Previous work [2] suggested that a possible way to autogenerate a local reset is by using an LUT configured as shift register (SRL) in each RM with all the bits initialized to the reset value (1's if reset is active high) and its input tied to the global system reset signal. The output of the SRL is used to reset the RM. This way, as soon as the RM is configured the RM will be in the reset state and the clock will start shifting the current value of the global reset signal, which should be deasserted assuming the rest of the system is currently running. Eventually, after a given number of cycles (length of the shift register, e.g., 16 cycles) the SRL output will be deasserted and the RM will come out of reset. This process is repeated every time the RM is configured.

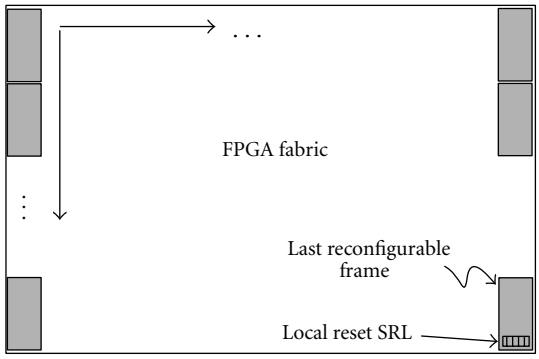

Figure 5: Placement of the reset SRL within the RR.

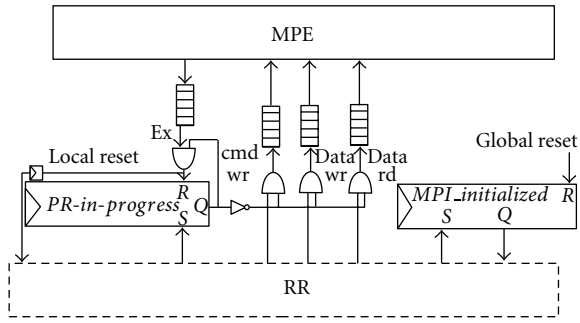

Figure 6: Static registers and decoupling logic between SR and RR.

However, a problem with the approach is that it will not work if the SRL placement is not properly constrained. We believe that partial reconfiguration of an entire RR does not happen atomically, which means that not all the logic of an RM is configured at once, but frame by frame. A PR frame is the smallest reconfigurable building block. A PR frame is 1 CLB wide by 16, 20, or 40 CLBs high for Virtex 4, 5, and 6, respectively. If the reset SRL is configured before the rest of the RM logic is configured then the local reset will be applied to old logic from the previous RM. The actual time it takes to complete PR depends on how fast the partial bitstream is transferred from host memory to the ICAP or SelectMAP interfaces, which in turn depends on what communication mechanism is being used (PCIe, FSB, Ethernet, etc.). In any case, a good solution should not rely on configuration time to be correct.

A possible solution to this problem is to force the CAD tools to place the SRL in the last LUT of the last frame within the RR to be configured as shown in Figure 5. This way we ensure that all the logic from the RM has been configured before the reset SRL and that the local reset will apply to the new RM logic. However, the caveat of this approach is, that to the best of our knowledge, there is no official documentation from Xilinx regarding how PR is done over the RR. For example, Figure 5 assumes that PR follows a top-down and left-right approach, but it could be otherwise.

To avoid the previous problems, the approach used to generate a local reset is to take advantage of the existing message-passing capability and the MPE to produce an external signal to the RM. This approach has the added benefit being platform independent. After an RM receives the configuration message ("cfg") and before PR takes place, the current RM can instruct the MPE to be ready to receive the "configuration-done" message, which will come eventually. The RM's FSM then goes into an idle state waiting for PR to occur. After PR is done and the new RM has been completely configured, the X86 processor sends the "configuration-done" message, which the MPE is already expecting. This causes the MPE to inform the RM of the newly arrived message by writing a control word to the command FIFO between the MPE and the RM. The exists signal (Ex in Figure 6) of the command FIFO is then used to generate the reset pulse for the new RM. After coming out of reset the RM can dequeue the control word and start running, or resume its operation by restoring its previous status (if any) as mentioned in Section 4.1.

There is an additional requirement for RMs that use ArchES-MPI when coming out of reset. The MPI_Init function must be called only once by all ranks at the beginning of the application. This function initializes the MPI environment and in the case of a hardware rank the function initializes the MPEs. This means that after coming out of reset an RM must know whether it is the first time the application is running, and therefore the MPE needs to be initialized, or if the MPE has been initialized already by a previous RM. In the later case the RM must not go through the initialization process again. Instead, it must continue its execution and restore its status (if any) using the "configuration-done" message.

A simple solution for this is to add a set-reset flip-flop to the SR that indicates that the MPE has been initialized. This flag, shown as *MPI_Initialized* in Figure 6, must be set by the first RM to run in the FPGA during the initialization process after global reset. Successive RMs should check this flag to know whether or not to do the initialization process. This flag is analogous to a static variable within a software function that is initialized only once during the first time the function is called.

4.3. Decoupling Logic. Another consideration when using PR is the decoupling between SR and RR logic because the outputs of the RR may be undefined during PR. This may cause undesired reads or writes to FIFOs with invalid data. To this end, the synchronization messages can be used to set and clear a flag to gate the outputs of the RM as shown in Figure 6. When the RM receives the PR message ("cfg" in Figure 4) it sets the *PR-in-progress* flag, which is a set-reset flip-flop placed in SR that will disable the RR outputs during PR. After PR is done, the flag is cleared when the "configuration-done" message is received. This is the same message used to generate the local reset as previously discussed. The decoupling logic occupies very little resources: 158 flip-flops and 186 LUTs per RR. With one RR this means less than 1% of the resources available in the XC5VLX330 FPGA.

5. Development Platform

Portability is one of the benefits of using a communications abstraction layer such as ArchES-MPI, hence adding partial reconfiguration must be done in a portable way as well. ArchES-MPI has been used in a variety of platforms, but the focus here will be on the one shown in Figure 7 as it provides a heterogeneous platform. It is based on a quad-socket motherboard with one Intel Quad-core Xeon processor and the other sockets have Nallatech FSB-socket accelerator modules [15]. The accelerator module consists of a stack of up to three PCB layers. The first layer (bottom-up) contains the base FPGA (XC5VLX110) and it is reserved as a connection interface between the front side bus (FSB) and the upper layers. The second and third layers contain two XC5VLX330 FPGAs each. The FPGAs are directly connected by LVDS buses. Figure 7 also shows a Xilinx evaluation board XUPV5LX110T with a PCIe-x1 link.

The FPGA on the FSB base module drives the SelectMAP interface of the FPGAs in the upper layers to perform PR. In contrast, the PCIe board uses the ICAP port as there is only one FPGA. The FSB_Bridge and PCIe_Bridge are hardware components that are used by the NoC to handle the communications over their respective interfaces and move data packets from the FPGA to shared memory and vice-versa. The bridges are also in charge of receiving the partial bitstreams and writing them to the appropriate configuration interface: ICAP or SelectMAP.

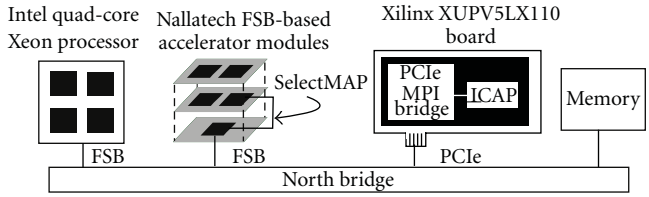

Figure 7: Development platform with FSB and PCIe-based FPGA boards.

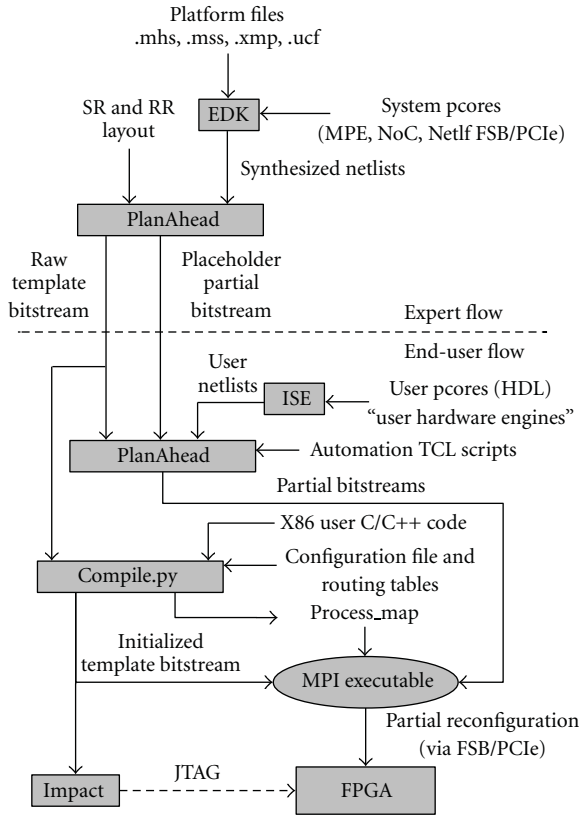

Figure 8: Expert and end-user software flows to generate template bitstreams, partial bitstreams, and the message-passing executables.

6. Software Flow

For the purposes of this paper, which is to incorporate PR into our existing MPI infrastructure, a standard Xilinx flow is followed with some extensions to it. However, any other PR flow that can generate and handle full and partial bitstreams could potentially be used as well.

There are two parts of the CAD flow shown in Figure 8: the first flow is used to create the template bitstreams by partitioning the design into static and reconfigurable regions. The second part of the flow allows the end user to reuse the prebuilt, prerouted template design. The Xilinx EDK/ISE 12.1 suite is used to implement and synthesize the system-level design including the NoC and message-passing infrastructure (system components). The netlists are then used in PlanAhead to layout the RRs, which requires certain expertise. With the new Xilinx Partition-based flow there is no need to explicitly implement bus macros. PlanAhead will automatically insert LUT1 elements where they are required. Finally, the full template bitstream is created and can be distributed to users. Additionally, the partial bitstream of a placeholder RM can also be created. The placeholder (or bootloader RM) is discussed in Section 8.1.

The end user can implement hardware engines as Xilinx pcores in an EDK project. The engines can be coded as HDL or via C-to-Gates tools and synthesize them. The resulting netlist can then be integrated into the PlanAhead project where the predesigned SR is reused and it is only necessary to place and route the RMs and generate user partial bitstreams. The end user does not have to do any layout, but PlanAhead is still necessary to generate the partial bitstreams. However, additional automation TCL scripts can help in simplifying the task of using PlanAhead. The last step is to run an ArchES-MPI Python script (compile.py in Figure 8) to compile the MPI code for all the processors (X86, MicroBlaze, PowerPC) and initialize the BRAMs for those embedded processors in the FPGA. The script also initializes the routing tables for the NoC and generates a hosts file required to run the application.

This host file (process_map in Figure 8) in a typical MPI implementation is known as a machine host file and it is used to specify the host, the type of processor and the executable binary assigned to each MPI rank. The ArchES-MPI library uses a similar concept but it is extended to use bitstreams. This file has two sections, the first section assigns each rank to a host, a processing element, and a binary to execute. If it is a processor, then the binary to execute is an

```
main()
MPI_Init(···);//implicit config. of SRs and initial RMs
...
MPI_Send(···, destRank, CFG_MSG_TAG···);
MPI_Recv(statusData_RM_A,..., destRank,OK_TO_CFG_TAG,···);
ARCHES_MPI_Reconfig(RM_B.bit,boardNum,fpgaNum);
MPI_Send(statusData_RM_B,..., destRank, CFG_DONE_TAG,···);
...
```

Figure 9: Partial reconfiguration code snippet.

ELF file generated by the compiler (e.g., gcc) for X86, or cross-compiler for embedded processors (e.g., mb-gcc). If it is a hardware engine, the binary to execute is the partial bitstream file generated by PlanAhead. The second section assigns the full template bitstreams (as opposed to partial bitstreams) to the FPGAs available in the platform.

The template bitstreams must be downloaded first. For multi-FPGA systems such as the Nallatech FSB-based accelerator modules, there can be many template bitstreams and they all can be downloaded at runtime via FSB. For the PCIe board, JTAG or Flash memory are the only options to configure the FPGA at boot time. Once the template bitstreams are in place, the partial bitstreams can be downloaded as many times as necessary using the FSB or PCIe link without the need to reboot the host machine.

The ArchES-MPI library parses the configuration file at runtime and during the execution of the `MPI_Init()` function the template bitstreams (if there are more than one) are downloaded to the FPGAs. This is completely transparent to the user. After this, the user can explicitly download partial bitstreams by calling the `ARCHES_MPI_Reconfig()` function (as shown in Figure 9) within the source code. This function uses three parameters: the partial bitstream filename, the FPGA board number (if there are multiple FPGA boards), and the FPGA number (in case a given board has many FPGAs). The partial bitstream file includes information that determines which RR to use. In addition, the user can send and receive the synchronization messages described in Section 4. The code uses MPI message tags (user defined) to distinguish the configuration messages from the application messages. Note that the same code can be used regardless of the type of communication interface (PCIe or FSB) or the type of configuration interface (ICAP or SelectMAP); it is all abstracted away from the user.

7. The Vector Engine Example

The overall performance of a parallel application programmed using message passing depends on many different factors, such as communication-to-computation ratio, memory access times, and data dependencies, among others. When using PR, additional factors must be included, such as the partial bitstream size, the number of RRs, and the PR granularity of the application, which is the ratio between the amount of computation per PR event. It is beyond the scope of this paper to provide a complete evaluation of PR overhead by exploring all the possible parameters. Instead,

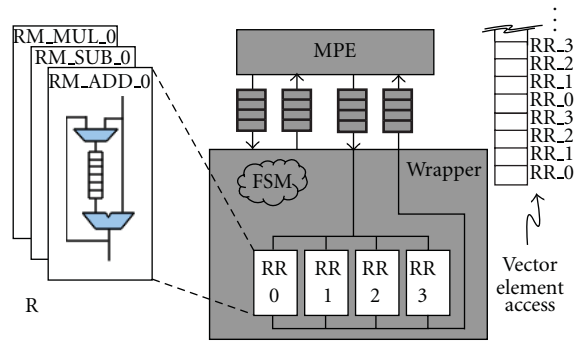

Figure 10: Vector Engine with 4 functional units (RRs) and three RMs.

we focus on measuring the PR overhead relative to the PR granularity. A simple vector engine example is used to verify that PR works in real hardware and to provide initial overhead measurements; it is not intended to be a fast vector operation accelerator.

This is an example of a wrapper-contained scenario (see Figure 3) because application logic is placed in the static region. The vector engine core is shown in Figure 10. The vector elements are distributed across the four functional pipelines of the engine (i.e., stride access of four). Each pipeline is an RM that can dynamically change the operation to perform (add, subtract, or multiply). Also, each pipeline includes an FIFO to store the vector elements.

The wrapper issues Send and Receive commands to the MPE and the data goes directly in and out of the pipelines. The wrapper decodes commands sent as messages from the master rank (Rank 0), which is the X86 processor; the vector engine is Rank 1. The vector commands can be LOAD (load a vector into the accumulator), ACC (perform the operation and store the results back into the accumulator), and STORE (flush the accumulator and send it back to the master rank). The wrapper controls the pipeline via control signals (e.g., enable, done, command, and vector_size).

There are four RRs and only one Vector Engine (i.e., one slave rank). The partial bitstream size is 130872 bytes per RR, and the vector size is set to 4000 floating point numbers. The experiment consists of performing 100 LOAD-ACC-STORE (LAS) cycles and performing PR every certain number of cycles. This setup allows us to control the amount of communication and computation per PR event. Note that the point of the experiment is to measure the overhead for a given setup and not to compare FSB and PCIe performance. The results are shown in Table 1. The worst-case scenario is when PR is performed every LAS cycle (i.e., fine PR granularity) with a degradation of 425-fold for PCIe and 308-fold for FSB compared to the case where no PR is performed at all. The best-case scenario is when PR is performed only once (i.e., coarse PR granularity) for the entire run with an overhead factor of 5 and 4 for PCIe and FSB, respectively. From those numbers, only 0.5% of the time is spent in the synchronization messages, the rest is spent in

Table 1: PR overhead at different PR event granularities.

Num of PR events	PCIe		FSB	
	Exec. time (s)	Times slower	Exec. Time (s)	Times slower
100	18.695	425	6.466	308
10	1.912	43	0.666	32
1	0.231	5	0.085	4
no PR	0.044	—	0.021	—

transferring the partial bitstreams. Note that there are four of them, one for each pipeline (RR).

These numbers are important to measure further improvements to the PR infrastructure, but they are not representative of the performance degradation expected of any application that uses this PR framework. Even for the same application, the overhead is less for a smaller partial bitstream, which is proportional to the size of the RR. The RR was drawn to an arbitrary size and the pipeline only uses 13% of LUTs and FFs, and 25% of BRAMs and DSPs of the RR. There was no attempt to minimize the RR size or to compress the partial bitstream. The MPE, NoC, and other infrastructure use 11044 LUTs (15%), 15950 FFs (23%), 80 BRAMs (54%), and 3 DSPs (4%) of the static region on the XC5VLX110.

The FSB-based board has less runtime overhead than the PCIe-based board because the FSB has less latency and more bandwidth than the PCIe-x1 link. Also, the FPGA in the FSB board runs at 133 MHz while the FPGA on the PCIe board runs at 125 MHz.

8. Generic Computing Platform Example

The vector accumulator is an application-specific template example because the wrapper implemented in the SR is part of the application. This section presents an example of a generic template bitstream that can be reused by four different applications, therefore, no application-related logic can be placed in the SR. In any case, the objective is not to show hardware acceleration but to prove functionality and that the same template bitstream is reused by four different applications. All of this is done without the user having to design the communications infrastructure or dealing with low-level hardware details.

Figure 11 shows the analogy between software and hardware binaries. The software binaries (.elf files) are loaded by the operating system and executed by X86 processor cores, and the partial bitstreams (.bit files) are loaded by the ArchES-MPI library and *executed* by the FPGA fabric within the RRs. In this case, PR is used to download the corresponding hardware engine to the FPGA at the beginning of each application. Figure 11 shows four RRs, this means that we could instantiate up to four hardware engines of the same application. Technically, it is also possible to have hardware engines of different applications in different RRs. However, our goal is to test that the same RR can be reused by different hardware engines.

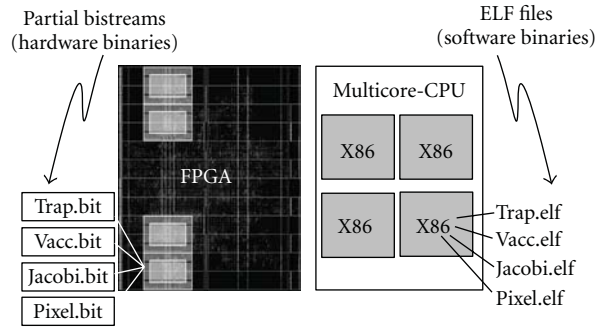

Figure 11: Mapping binaries to a generic template bitstream and X86 multicore CPU.

The first application is the same vector accumulator discussed in Section 7 but now in a self-contained form. This means the wrapper is now also in the RR, not in the SR as before. This implies some minor changes to the wrapper's FSM, basically it was necessary to remove the functionality to generate the local reset to the floating point pipelines. Now the reset is generated as described in Section 4.2, external to the wrapper using the MPE and messages. Now the wrapper itself needs to be reset as well. The same reset mechanism is used for all four different application engines. The vector accumulator is hand coded in VHDL.

The second application computes the area under the curve for a given function using the trapezoidal method. This application is based on the software MPI parallel implementation presented in Pacheco [16]. Impulse-C [17], a C-to-Gates compiler, is used to generate the hardware engine. With the addition of some macros introduced in previous work [18], it is possible to translate MPI calls in the C code to MPE commands in VHDL. This application requires at least two ranks but is designed to use more. In this case, one software rank and one hardware rank; both ranks perform the same computation on different data.

The third application is a simple Jacobi iteration method for approximating the solution to a linear system of equations to solve the Laplace equation with finite differences. The implementation of this application is based on previous work [19], and it was written in C and MPI for embedded processors. This application uses three ranks, one master and two slaves. The master and one slave are software ranks, and the other slave is a hardware rank. Again, Impulse-C is used to generate the hardware engine with commands to the MPE.

Finally, the fourth application is another simple MPI program written in C to invert pixels from image frames coming from a webcam or a file stored in the host's hard drive. This application uses OpenCV [20], an open source library for computer vision that provides an easy way to access the Linux video device and to open, read, and write video and image files. This application uses three ranks, Rank 0 (X86) captures the frames and send them to Rank 1 (hardware engine) where the pixels are inverted and then it sends the processed frame to Rank 2 (X86) where the frame is stored in a file or displayed on the screen. The hardware engine is hand coded in VHDL.

Although there has not been any attempt to optimize the size and placement of the RRs, the FPGA in Figure 11 has been partitioned following an educated guess. This guess comes from the experience of having to design the low-level LVDS communication interface for the FSB-based platform (See Figure 7), which is the one used in this example. The FPGA in this platform has abundant LVDS I/O logic to communicate with other FPGAs above, sideways, and below. This logic is placed and concentrated towards the middle of the chip closer to specific I/O banks. Therefore, placing the RR around the middle of the chip would complicate timing closure because the RR would displace the LVDS logic creating longer paths and delays. Additionally, the XC5VLX330 FPGA only has two columns of DSP blocks on the left side of the chip—it is not symmetrical, therefore, there are no RRs on the right side for this particular template. A different template may include DSP-enabled and non-DSP-enabled RRs. In any case, these are exactly the kind of decisions that the end user, that is, the application developer, should not need to worry about. The place where the I/O banks or DSPs are located is completely irrelevant to the trapezoidal method or the Jacobi algorithm. Instead, the template bitstream has already been created by a hardware engineer familiar with the platform who can layout the RRs for the user to work with.

The four applications were compiled, implemented, and run on the FSB-based platform and all the output results were correct for each application. The RR itself occupies 6.8% of LUTs and FFs, 4.9% of BRAMs, and 16.6% of DSPs available in the XC5VLX330 FPGA. The partial bitstreams for the four application RMs have the same size (477666 bytes), although they do not require the same resources. Table 2 presents the utilization percentage with respect to the RR not the entire FPGA. In this case, all the engines fit within that RR. If this would not have been the case then another template with a larger RR would have been selected. The decision of what template bitstream to use from a collection of them can be done by a higher-level tool that matches the resources required by an application and resources available in a template. This is an optimization problem that can be addressed in future research.

8.1. Boot Loader Reconfigurable Module. When a template bitstream is downloaded to an FPGA each RR must have an initial RM configured. The question is what RM should be placed at the beginning. In our generic platform a simple placeholder RM has been created to be the initial RM. This placeholder is in fact a small bootloader for partial bitstreams that merely follows the synchronization steps described in Section 4. It instructs the MPE to receive a PR message ("cfg" message in Figure 4) indicating that a new RM is going to be configured. The bootloader RM will then issue another receive command to the MPE for the eventual "configuration-done" message, then it disables all the outputs from the RR by setting the *PR-in-progress* flag and finally it will go to an idle state waiting for PR to occur. Once the application RM is configured and out of reset the application begins. Just before the application is about to finish, during the call to the MPI_Finalize function (all ranks must call it) the bootloader RM is restored in all the RRs to set the FPGA ready for the next application. This last step is analogous to a software process that finishes and returns the control of the processor to the operating system. Similarly, when the application RM finishes it returns the control of the MPE to the bootloader RM.

Table 2: Resource utilization of four application hardware engines (Percentages are relative to the RR).

	RR	Pixel	Trapezoidal	Vector acc	Jacobi
LUTs	9920	154 (2%)	6397 (65%)	1476 (20%)	5271 (54%)
FFs	9920	112 (2%)	3993 (41%)	1553 (16%)	4460 (45%)
DSPs	32	0	32 (100%)	8 (25%)	13 (41%)
BRAM	16	0	0	4 (25%)	2 (13%)

9. Conclusions

The main contribution of this paper is the concept of embedding partial reconfiguration into the MPI programming model. We have presented an extension to the ArchES-MPI framework that simplifies the use of PR by abstracting away the hardware details from the user while providing a layer of portability. Such layer allows PR to be performed regardless of the configuration interface (ICAP or SelectMAP) and independent of the communication channel used (PCIe or FSB) to communicate with an X86 processor, which is used as a configuration controller. Also, we demonstrated a method to generate local resets without the need of an embedded processor, and provided an alternative to actively store and restore the status of reconfigurable modules via explicit messages. Very little additional logic and code were required on top of the existing message-passing infrastructure to enable PR. In addition, the concept of template-based bitstreams was introduced as a way to use this PR framework to create reusable and generic computing platforms. Four different applications were actually implemented and executed consecutively reusing the same template bitstream. An approach like this simplifies and speeds up the design process of creating multiaccelerator-based computing systems by allowing third-party hardware experts to provide prebuilt accelerators, I/O interfaces, NoC, and memory controllers, letting the designer focus on the actual value-added application logic. The initial performance overhead numbers obtained set a reference point to measure future improvements to the PR framework.

Acknowledgments

The authors acknowledge the CMC/SOCRN, NSERC, Impulse Accelerated Technologies, and Xilinx for the hardware, tools, and funding provided for this paper.

References

[1] Xilinx, Inc., "Partial Reconfiguration User Guide," http://www.xilinx.com/support/documentation/sw_manuals//xilinx12_2/ug702.pdf.

[2] M. Saldaña, A. Patel, H. J. Liu, and P. Chow, "Using partial reconfiguration in an embedded message-passing system," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 418–423, December 2010.

[3] ArchES Computing, Inc., http://www.archescomputing.com.

[4] M. Saldaña, A. Patel, C. Madill et al., "MPI as an abstraction for software-hardware interaction for HPRCs," in *Proceedings of the 2nd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '08)*, pp. 1–10, November 2008.

[5] 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 Applications (FPL '06)*, pp. 1–6, August 2006.

[6] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic scheduling of tasks on partially reconfigurable FPGAs," *IEE Proceedings: Computers and Digital Techniques*, vol. 147, no. 3, pp. 181–188, 2000.

[7] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "DyNoC: a dynamic infrastructure for communication in dynamically reconfigurable devices," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 153–158, August 2005.

[8] K. Papadimitriou, A. Anyfantis, and A. Dollas, "An effective framework to evaluate dynamic partial reconfiguration in FPGA systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 59, no. 6, pp. 1642–1651, 2010.

[9] C. Rossmeissl, A. Sreeramareddy, and A. Akoglu, "Partial bitstream 2-D core relocation for reconfigurable architectures," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 98–105, August 2009.

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

[11] D. Koch and J. Teich, "Platform-independent methodology for partial reconfiguration," in *Proceedings of the 1st Conference on Computing Frontiers (CF '04)*, pp. 398–403, ACM, New York, NY, USA, 2004.

[12] H.-H. So, A. Tkachenko, and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pp. 259–264, October 2006.

[13] L. Möller, R. Soares, E. Carvalho, I. Grehs, N. Calazans, and F. Moraes, "Infrastructure for dynamic reconfigurable systems: choices and trade-offs," in *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design (SBCCI '06)*, pp. 44–49, ACM, New York, NY, USA, 2006.

[14] The MPI Forum, "MPI: a message passing interface," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 878–883, ACM, New York, NY, USA, November 1993.

[15] Nallatech, http://www.nallatech.com/.

[16] P. S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997.

[17] Impulse Accelerated Technologies, http://www.impulseaccelerated.com/.

[18] A. W. House, M. Saldaña, and P. Chow, "Integrating high-level synthesis into MPI," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, pp. 175–178, May 2010.

[19] M. Saldaña, D. Nunes, E. Ramalho, and P. Chow, "Configuration and programming of heterogeneous multiprocessors on a multi-FPGA system using TMD-MPI," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA's, (ReConFig '06)*, pp. 1–10, September 2006.

[20] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 439141, 15 pages
doi:10.1155/2012/439141

Research Article

Exploring Many-Core Design Templates for FPGAs and ASICs

Ilia Lebedev,[1] Christopher Fletcher,[1] Shaoyi Cheng,[2] James Martin,[2] Austin Doupnik,[2] Daniel Burke,[2] Mingjie Lin,[2] and John Wawrzynek[2]

[1] *CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*
[2] *Department of EECS, University of California at Berkeley, CA 94704, USA*

Correspondence should be addressed to Ilia Lebedev, ilebedev@csail.mit.edu

Received 2 May 2011; Accepted 15 July 2011

Academic Editor: Claudia Feregrino

We present a highly productive approach to hardware design based on a many-core microarchitectural template used to implement compute-bound applications expressed in a high-level data-parallel language such as OpenCL. The template is customized on a per-application basis via a range of high-level parameters such as the interconnect topology or processing element architecture. The key benefits of this approach are that it (i) allows programmers to express parallelism through an API defined in a high-level programming language, (ii) supports coarse-grained multithreading and fine-grained threading while permitting bit-level resource control, and (iii) reduces the effort required to repurpose the system for different algorithms or different applications. We compare template-driven design to both full-custom and programmable approaches by studying implementations of a compute-bound data-parallel Bayesian graph inference algorithm across several candidate platforms. Specifically, we examine a range of template-based implementations on both FPGA and ASIC platforms and compare each against full custom designs. Throughout this study, we use a general-purpose graphics processing unit (GPGPU) implementation as a performance and area baseline. We show that our approach, similar in productivity to programmable approaches such as GPGPU applications, yields implementations with performance approaching that of full-custom designs on both FPGA and ASIC platforms.

1. Introduction

Direct hardware implementations, using platforms such as FPGAs and ASICs, possess a huge potential for exploiting application-specific parallelism and performing efficient computation. As a result, the overall performance of custom hardware-based implementations is often higher than that of software-based ones [1, 2]. To attain bare metal performance, however, programmers must employ hardware design principles such as clock management, state machines, pipelining, and device specific memory management—all concepts well outside the expertise of application-oriented software developers.

These observations raise a natural question: *does there exist a more productive abstraction for high-performance hardware design*? Based on modern programming disciplines, one viable approach would (1) allow programmers to express parallelism through some API defined in a high-level programming language, (2) support coarse-grain multithreading and fine-grain threading while permitting bit-level resource control, and (3) reduce the effort required to repurpose the implemented hardware platform for different algorithms or different applications. This paper proposes an abstraction that constrains the design to a *microarchitectural template*, accompanied by an API, that meets these programmer requirements.

Intuitively, constraining the design to a template would likely result in performance degradation compared to fully-customized solutions. Consider the high-level chart plotting designer effort versus performance, shown in Figure 1. We argue that the shaded region in the figure is attainable by template-based designs and warrants a systematic exploration. To that end, this work attempts to quantify the performance/area tradeoff, with respect to designer effort, across template-based, hand-optimized, and programmable approaches on both FPGA and ASIC platforms. From our analysis, we show how a disciplined approach with architectural constraints, without resorting to manual hardware design,

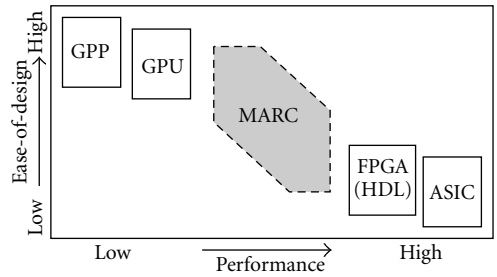

Figure 1: Landscape of modern computing platforms. Ease of application design and implementation versus performance (GPP stands for *general purpose processor*).

may reduce design time and effort while maintaining acceptable performance.

In this paper; we study microarchitectural templates in the context of a compute-intensive data-parallel Bayesian inference application. Our thesis, therefore, is that we can efficiently map our application to hardware while being constrained to a *many-core* template and parallel programming API. We call this project MARC, for *Many-core Approach to Reconfigurable Computing*, although template-based architectures can be applied outside the many-core paradigm.

We think of a *template* as an architectural model with a set of parameters to be chosen based on characteristics of the target application. Our understanding of which aspects of the architecture to parameterize continues to evolve as we investigate different application mappings. However, obvious parameters in the many-core context are the number of processing cores, core arithmetic-width, core pipeline depth, richness and topology of an interconnection network, and customization of cores—from addition of specialized instructions to fixed function datapaths as well as details of the cache and local store hierarchies. In this study we explore a part of this space and compare the performance/area between MARC and hand-optimized designs in the context of a baseline GPGPU implementation.

The rest of the paper is organized as follows: Section 2 introduces the Bayesian network inference application, the case study examined in this paper. Section 3 describes the execution model used by OpenCL, the high-level language used to describe our MARC and GPGPU implementations. The application mapping for the GPGPU platform is detailed in Section 4. We discuss the hand-optimized and MARC implementations in Section 5. Section 6 covers hardware mappings for both the hand-optimized and MARC designs as well as a comparison between FPGA and ASIC technology. Finally, in Section 7, we compare the performance/area between the MARC hand-optimized and GPGPU implementations.

1.1. Related Work. Numerous projects and products have offered ways to ease FPGA programming by giving developers a familiar C-style language in place of HDLs [3]. Early research efforts including [4–6] formed the basis for recent commercial offerings: Catapult C from Mentor Graphics, ImpulseC, Synfora Pico from Synopsys, and AutoESL from Xilinx, among others. Each of these solutions requires developers to understand hardware-specific concepts and to program using a model that differs greatly from standard C—in a sense, using an HDL with a C syntax. Unlike these approaches, the goal of MARC is to expose software programming models applicable to design of efficient hardware.

There has been a long history of mapping conventional CPU architectures to FPGAs. Traditionally, soft processors have been used either as a controller for a dedicated computing engine, or as an emulation or prototyping plat-form for design verification. These efforts have primarily employed a single or small number of processor cores. A few FPGA systems with a large number of cores have been implemented, such as the RAMP project [7]. However, the primary design and use of these machines have been as emulation platforms for custom silicon designs.

There have been many efforts both commercially and academically on customization of parameterized processor cores on an application-specific basis. The most widely used is Xtensa from Tensilica, where custom instructions are added to a conventional processor architecture. We take processor customization a step further by allowing the instruction processors to be replaced with an application-specific datapath that can be generated automatically via our C-to-gates tool for added efficiency and performance. We leverage standard techniques from C-to-gates compilation to accomplish the generation of these custom datapaths.

More recently, there have been several other efforts in integrating the C-to-gates flow with parallel programming models, [8, 9]. These projects share with MARC the goal of exploiting progress in parallel programming languages and automatically mapping to hardware.

2. Application: Bayesian Network Inference

This work's target application is a system that learns Bayesian network structure from observation data. Bayesian networks (BNs) and graphical models have numerous applications in bioinformatics, finance, signal processing, and computer vision. Recently they have been applied to problems in systems biology and personalized medicine, providing tools for processing everincreasing amounts of data provided by high-throughput biological experiments. BNs' probabilistic nature allows them to model uncertainty in real life systems as well as the noise that is inherent in many sources of data. Unlike Markov Random Fields and undirected graphs, BNs can easily learn sparse and causal structures that are interpretable by scientists [10–12].

We chose to compare MARC in the context of Bayesian inference for two primary reasons. First, Bayesian inference is a computationally intensive application believed to be particularly well suited for FPGA acceleration as illustrated by [13]. Second, our group, in collaboration with Stanford University, has expended significant effort over the previous two years developing several generations of a hand-optimized FPGA implementation tailored for Bayesian inference [13, 14]. Therefore, we have not only a concrete reference design but also well-corroborated performance results for fair comparisons with hand-optimized FPGA implementations.

2.1. Statistics Perspective. BNs are statistical models that capture conditional independence between variables via the local Markov property: *that a node is conditionally independent of its nondescendants, given its parents*. Bayesian inference is the process by which a BN's graph structure is learned from the quantitative observation, or *evidence*, that the BN seeks to model. Once a BN's structure (a set of nodes $\{V_1, \ldots, V_{\mathcal{N}}\}$) is determined, and the conditional dependence of each node V_i on its parent set Π_i is tabulated, the joint distribution over the nodes can be expressed as

$$P(V_1, \ldots, V_{\mathcal{N}}) = \prod_{i=1}^{\mathcal{N}} P(V_i \mid \Pi_i). \tag{1}$$

Despite significant recent progress in algorithm development, computational inference of a BN's graph structure from evidence is still NP-hard [15] and remains infeasible except for cases with only a small number of variables [16].

The algorithm surveyed in this paper is the union of two BN inference kernels, the order and graph samplers (jointly called the "order-graph sampler"). An *order* is given by a topological sort of a graph's nodes, where each node is placed after its parents. Each order can include or be *compatible* with many different graphs. The order sampler takes a BN's observation data and produces a set of "high-scoring" BN orders (orders that best explain the evidence). The graph sampler takes this set of high-scoring orders and produces a single highest-scoring graph for each order. The observation data is generated in a preprocessing steps and consists of $\mathcal{N}$ (*for each node*) sets of $\mathcal{P}$ local-score/parent-set pairs (which we will refer to as "data" for short). A local score describes the likelihood that a given parent set is a node's true parent set, given an order. A postprocessing step is performed after the order-graph sampler to normalize scores and otherwise clean up the result before it is presented to the user.

In this work, we only study the order and graph sampler steps for two reasons. First, the order-graph sampler is responsible for most of the algorithm's computational complexity. Second, the pre- and postprocessing phases are currently performed on a GPP (general purpose processor) platform regardless of the platform chosen to implement the order-graph sampler.

Following [14, 16, 17], the order-graph sampler uses Markov chain Monte Carlo (MCMC) sampling to perform an iterative random walk in the space of BN orders. First, the algorithm picks a random initial order. The application then iterates as follows (1) the current order is modified by swapping two nodes to form a "proposed order," which is (2) scored, and (3) either accepted or rejected according to the Metropolis-Hastings rule. The scoring process, itself, (1) breaks the proposed order into $\mathcal{N}$ disjoint "local orders," and (2) iterates over each node's parent sets, accumulating each local score whose parent set is *compatible* with the node's local order. For a network of $\mathcal{N}$ nodes, the proposed order's score can be efficiently calculated by [18]

$$\text{Score}\left(\mathcal{O}_p \mid \mathcal{D}\right) = \prod_{i=1}^{\mathcal{N}} \sum_{\Pi_i \in \Pi_{\mathcal{P}}} \text{LocalScore}\left(V_i, \Pi_i; \mathcal{D}, \mathcal{O}_p\right), \tag{2}$$

where $\mathcal{D}$ is the set of raw observations that are used by the preprocessor to generate local scores, and $\mathcal{O}_p$ is the proposed order. This iterative operation continues until the score has converged.

To decrease time to convergence, $\mathcal{C}$ orders (together called a *chain*) can be dispatched over a temperature ladder and exchanged per iteration in a technique known as parallel tempering. Additionally, $\mathcal{R}$ independent chains (called multiple *restarts*) can be dispatched to increase confidence that the optimum score is a global optimum.

2.2. Compute Perspective. The order-graph sampler is a compute intensive set of nested loops, shown in Algorithm 1, while the *score*() function arithmetic is shown in Algorithm 2. To put the number of loop iterations into perspec-tive, typical parameter values for $\{\mathcal{I}, \mathcal{R} * \mathcal{C}, \mathcal{N}\}$ are $\{10000, 512, 32 \text{ or } 37\}$, where $\mathcal{I}$ is the number of MCMC iterations. Furthermore, $\mathcal{P} = \sum_{i=0}^{4} \binom{\mathcal{N}-1}{i}$, which equates to 36457 and 66712 ($\mathcal{N} = 32$ and $\mathcal{N} = 37$, resp.) for the design points that we study (see Section 7).

We classify the reformulated loop nest as compute intensive for two reasons. First, a relatively small amount of input (a local order) is needed for the *score*() function to compute per-node results over the $\mathcal{R} * \mathcal{C}$ orders. Second, $\mathcal{D}[n]$ (shown in Algorithm 1) depends on $\mathcal{N}$ and not $\mathcal{R} * \mathcal{C}$. Since $\mathcal{N} \approx 37$ and $\mathcal{R} * \mathcal{C} = 512$ (i.e., $\mathcal{R} * \mathcal{C} \gg \mathcal{N}$), in practice there is a large amount of compute time between when n in $\mathcal{D}[n]$ changes.

3. The OpenCL Execution Model

To enable highly productive hardware design, we employ a high-level language and execution model well suited for the paradigm of applications we are interested in studying: data-parallel, compute-bound algorithms. Due to its popularity, flexibility, and good match with our goals, we employ OpenCL (Open Computing Language) as the programming model used to describe applications.

OpenCL [19] is a programming and execution model for heterogeneous systems containing GPPs, GPGPUs, FPGAs [20], and other accelerators designed to explicitly capture data and task parallelism in an application. The OpenCL model distinguishes control thread(s) (to be executed on a GPP host) from kernel threads (data parallel loops to be executed on a GPGPU, or similar, device). The user specifies how the kernels map to an n-dimensional dataset, given a set of arguments (such as constants or pointers to device or host memory). The runtime then distributes the resulting workload across available compute resources on the device. Communication between control and kernel threads is provided by shared memory and OpenCL system calls such as barriers and bulk memory copy operations.

A key property of OpenCL is its memory model. Each kernel has access to three disjoint memory regions: private, local, and global. Global memory is shared by all kernel threads, local memory is shared by threads belonging to the same group, while private memory is owned by one kernel thread. This alleviates the need for a compiler to perform

```
    for {r, c} in ℛ × 𝒞 do
      initialize(𝒪[r][c])
    end for
    for i in ℐ do
5:    for {r, c} in ℛ × 𝒞 do
        𝒪_p[r][c] ← swap(𝒪[r][c])
        Variable initialization:
        𝒪_p[r][c] · 𝒮_o, 𝒪_p[r][c] · 𝒮_g ← 0
        𝒪_p[r][c] · 𝒢 ← []
10:   end for
      for n in 𝒩 do
        for {r, c} in ℛ × 𝒞 do
          (s_o, s_g, g) = score(𝒟[n], 𝒪_p[r][c][n])
          𝒪_p[r][c] · 𝒮_o ← 𝒪_p[r][c] · 𝒮_o + s_o
15:       𝒪_p[r][c] · 𝒮_g ← 𝒪_p[r, c] · 𝒮_g + s_g
          𝒪_p[r][c] · 𝒢 · append(g)
        end for
      end for
      for r in ℛ do
        Metropolis-hastings:
20:     for c in 𝒞 do
          if 1/T_c × (𝒪_p[r][c] · 𝒮_o − 𝒪[r][c] · 𝒮_o) >
          log(rand(0, 1)) then
            𝒪[r, c] · 𝒮_o ← 𝒪_p[r][c] · 𝒮_o
            save({𝒪_p[r][c] · 𝒮_g, 𝒪_p[r, c] · 𝒢})
          end if
25:     end for
        Parallel tempering:
        for c in 𝒞 do
          d ← 𝒪[r][c] · 𝒮_o − 𝒪[r][c + 1] · 𝒮_o
          if log(rand(0, 1)) < d × (1/T_c − 1/T_{c+1}) then
30:         exchange(𝒪[r][c], 𝒪[r][c + 1])
          end if
        end for
      end for
    end for
```

Algorithm 1: The reformulated order-graph sampler loop nest. $\{\mathcal{S}_o, \mathcal{S}_g\}$ and $\mathcal{G}$ are the current {order, graph} scores and graph associated with an order. *initialize*() generates a random order, *swap*() exchanges nodes in an order, and *save*() saves a result for the postprocessing step.

costly memory access analysis to recognize dependencies before the application can be parallelized. Instead, the user specifies how the application is partitioned into data-parallel kernels. With underlying SIMD principles, OpenCL is well suited for data-parallel problems and maps well to the parallel thread dispatch architecture found in GPGPUs.

4. Baseline GPGPU Implementation

To implement the order-graph sampler on the GPGPU, the application is first divided to different parts according to their characteristics. The scoring portion of the algorithm, which exhibits abundant data parallelism, is partitioned into a kernel and executed on the GPGPU, while the less parallelizable score accumulation is executed on a GPP. This ensures that the kernel executed on the GPGPU is maximally parallel and exhibits no interthread communication—an approach we experimentally determined to be optimal. Under this scheme, the latency of the control thread and score accumulation phases of the application, running on a GPP, are dominated by the latency of the scoring function running on the GPGPU. Moreover, the *score*() kernel (detailed in the following section) has a relatively low bandwidth requirement, allowing us to offload accumulation to the GPP, lowering total latency. The GPP-GPGPU implementation is algorithmically identical to the hardware implementations, aside from minor differences in the precision of the $log1p(\exp(d))$ operation, and yields identical results up to the random sequence used for Monte Carlo integration.

4.1. Optimization of Scoring Kernel. We followed four main strategies in optimizing the scoring unit kernel: (1) minimizing data transfer overhead between the control thread and the scoring function, (2) aligning data in device memory, (3)

```
    s_o, s_g ← −∞
    g ← NULL
    for p in 𝒫 do
      if compatible(𝒟[p] · ps, 𝒪_l) then
5:      Order sampler:
        d ← 𝒟[p] · ls − s_o
        If d ≥ HIGH_THRESHOLD then
          s_o ← 𝒟[p] · ls
        else if d > LOW_THRESHOLD then
10:       s_o ← s_o + log(1 + exp(d))
        end if
        Graph sampler:
        if 𝒟[p] · ls > s_g then
          s_g ← 𝒟[p] · ls
15:       g ← 𝒟[p] · ps
        end if
      end if
    end for
    Return:(s_o, s_g, g)
```

Algorithm 2: The $score(\mathcal{D}, \mathcal{O}_l)$ function takes the data $\mathcal{D}$ (made of parent set (ps) and local score (ls) pairs) and a local order ($\mathcal{O}_l$) as input. The scoring function produces an order score (s_o), graph score (s_g), and graph fragment (g).

allocating kernel threads to compute units on the GPGPU, and (4) minimizing latency of a single kernel thread.

First, we minimize data transfers between the GPP and GPGPU by only communicating changing portions of the data set throughout the computation. At application startup, we statically allocate memory for all arrays used on the GPGPU, statically set these arrays' pointers as kernel arguments, and copy all parent sets and local scores into off-chip GPGPU memory to avoid copying static data each iteration. Each iteration, the GPP copies $\mathcal{R} * \mathcal{C}$ proposed orders to the GPGPU and collects $\mathcal{R} * \mathcal{C} * \mathcal{N}$ proposed order/graph scores, as well as $\mathcal{R} * \mathcal{C}$ graphs from the GPGPU. Each order and graph is an $\mathcal{N} \times \mathcal{N}$ matrix, represented as $\mathcal{N}$ 64 bit integers, while partial order and graph scores are each 32 bit integers (additional range is introduced when the partial scores are accumulated). The resulting bandwidth requirement per iteration is $8 * \mathcal{R} * \mathcal{C} * \mathcal{N}$ bytes from the GPP to the GPGPU and $16 * \mathcal{R} * \mathcal{C} * \mathcal{N}$ bytes from the GPGPU back to the GPP. In the BNs surveyed in this paper, this bandwidth requirement ranges from 128 to 256 KB (GPP to GPGPU) and from 256 to 512 KB (GPGPU to GPP). Given these relatively small quantities and the GPGPU platform's relatively high transfer bandwidth over PCIe, the transfer latency approaches a minimal value. We use this to our advantage and offload score accumulation to the GPP, trading significant accumulation latency for a small increase in GPP-GPGPU transfer latency. This modification gives us an added advantage via avoiding intra-kernel communication altogether (which is costly on the GPGPU because it does not offer hardware support for producer-consumer parallelism).

Second, we align and organize data in memory to maximize access locality for each kernel thread. GPGPU memories are seldom cached, while DRAM accesses are several words wide—comparable to GPP cache lines. We therefore coalesce memory accesses to reduce the memory access range of a single kernel and of multiple kernels executing on a given compute unit. No thread accesses (local scores and parent sets) are shared across multiple nodes, so we organize local scores and parent sets by $[\mathcal{N}][\mathcal{P}]$. When organizing data related to the $\mathcal{R} * \mathcal{C}$ orders (the proposed orders, graph/order scores, and graphs), we choose to maximally compact data for restarts, then chains, and finally nodes ($[\mathcal{N}][\mathcal{C}][\mathcal{R}]$). This order is based on the observation that a typical application instance will work with a large number of restarts relative to chains. When possible, we align data in memory—rounding both $\mathcal{R}$, $\mathcal{C}$ and $\mathcal{P}$ to next powers of two to avoid false sharing in wide word memory operations and to improve alignment of data in memory.

Third, allocating kernel threads to device memory is straightforward given the way we organize data in device memory; we allocate multiple threads with localized memory access patterns. Given our memory layout, we first try dispatching multiple restarts onto the same compute unit. If more threads are needed than restarts available, we dispatch multiple chains as well. We continue increasing the number of threads per compute unit in this way until we reach an optimum—the point where overhead due to multithreading overtakes the benefit of additional threads. Many of the strategies guiding our optimization effort are outlined in [21].

Finally, we minimize the scoring operation latency over a single kernel instance. We allow the compiler to predicate conditional to avoid thread divergence. Outside the inner loop, we explicitly precompute offsets to access the $[\mathcal{N}][\mathcal{P}]$ and $[\mathcal{N}][\mathcal{C}][\mathcal{R}]$ arrays to avoid redundant computation. We experimentally determined that loop unrolling the $score()$ loop has minimal impact on kernel performance, so we allow the compiler to unroll freely. We also evaluated a direct implementation of the $log1p(\exp(d))$ operation versus the use of a lookup table in shared memory (which mirrors

the hand-optimized design's approach). Due to the low utilization of the floating point units by this algorithm, the direct implementation tends to perform better than a lookup table given the precision required by the algorithm.

4.2. Benchmarking the GPGPU Implementation. To obtain GPGPU measurements, We mapped the data parallel component to the GPGPU via OpenCL, and optimized the resulting kernel as detailed in Section 4.1. We measured the relative latency of each phase of the algorithm by introducing a number of GPP and GPGPU timers throughout the iteration loop. We then computed the latency of each phase of computation (scoring, accumulation, MCMC, etc.) and normalized to the measured latency of a single iteration with no profiling syscalls. To measure the iteration time, we ran the application for 1000 iterations with no profiling code in the loop and then measured the total time elapsed using the system clock. We then computed the aggregate iteration latency.

5. Architecture on Hardware Platforms

As with the GPGPU implementation, when the Bayesian inference algorithm is mapped to hardware platforms (FPGA/ASIC), it is partitioned into two communicating entities: a data-parallel scoring unit (a collection of Algorithmic-cores or A-cores) and a control unit (the Control-core, or C-core). The A-cores are responsible for all iterations of the *score*() function from Algorithm 1 while the C-core implements the serial control logic around the *score*() calls. This scheme is applied to both the hand-optimized design and the automatically generated MARC design, though each of them has different interconnect networks, memory subsystems, and methodologies for creating the cores.

5.1. Hand-Optimized Design. The hand-optimized design mapping integrates the jobs of the C-core and A-cores on the same die and uses a front-end GPP for system initialization and result collection. At the start of a run, network data and a set of $\mathcal{R} * \mathcal{C}$ initial orders are copied to a DRAM accessible by the A-cores, and the C-core is given a "Start" signal. At the start of each iteration, the C-core forms $\mathcal{R} * \mathcal{C}$ proposed orders, partitions each by node, and dispatches the resulting $\mathcal{N} * \mathcal{R} * \mathcal{C}$ local orders as threads to the A-cores. As each iteration completes, the C-core streams results back to the front-end GPP while proceeding with the next MCMC iteration.

The hand-optimized design is partitioned into four clock domains. First, we clock A-cores at the highest frequency possible (between 250 and 300 MHz) as these have a direct impact on system performance. Second, we clock the logic and interconnect around each A-core at a relatively low frequency (25–50 MHz) as the application is compute bound in the cores. Third, we set the memory subsystem to the frequency specified by the memory (~200 MHz, using a DRAM DIMM in our case). Finally, the C-core logic is clocked at 100 MHz, which we found to be ideal for timing closure and tool run time given a performance requirement (the latency of the C-core is negligible compared to the A-cores).

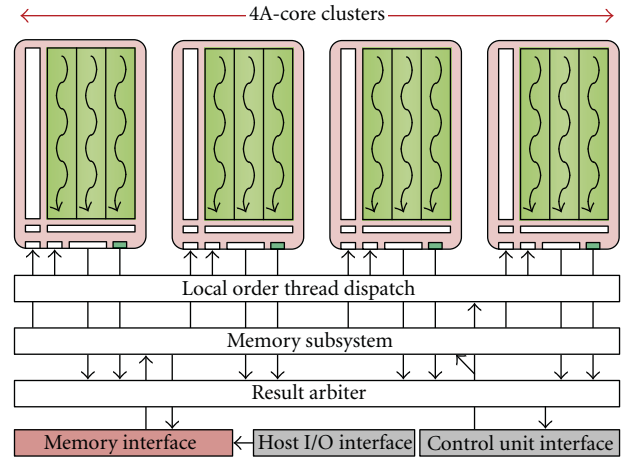

Figure 2: The hand-optimized scoring unit (with 4 A-core clusters).

5.1.1. Scoring Unit. The scoring unit (shown in Figure 2) consists of a collection of clustered A-cores, a point-to-point interface with the control unit, and an interface to off-chip memory.

A scoring unit cluster caches some or all of a node's data, taking a stream of local orders as input and outputs order/graph scores as well as partial graphs. A cluster is made up of {A-cores, RAM} pairs, where each RAM *streams* data to its A-core. When a cluster receives a local order, it (a) pages in data from DRAM as needed by the local orders, strip-mines that data evenly across the RAMs and (b) dispatches the local order to each core. Following Algorithm 1, $\mathcal{R} * \mathcal{C}$ local orders can be assigned to a cluster per DRAM request. Once data is paged in, each A-core runs $\mathcal{P}_f/U_c$ iterations of the *score*() inner loop (from Algorithm 2), where $\mathcal{P}_f$ is the subset of $\mathcal{P}$ that was paged into the cluster, and U_c is the number of A-cores in the cluster.

A-core clusters are designed to maximize local order throughput. A-cores are replicated to the highest extent possible to maximize read bandwidth achievable by the RAMs. Each A-core is fine-grained multithreaded across multiple iterations of the *score*() function and uses predicated execution to avoid thread divergence in case of non-compatible (!*compatible*()) parent sets. To avoid structural hazards in the scoring pipeline, all scoring arithmetic is built directly into the hardware.

Mapping a single node's scoring operation onto multiple A-cores requires increased complexity in accumulating partial node scores at the end of the *score*() loop. To maximally hide this delay, we first interleave cross-thread accumulation with the next local order's main scoring operation (shown in Figure 3). Next, we chain A-cores together using a dedicated interconnect, allowing cross-core partial results to be interleaved into the next local order in the same way as threads. Per core, this accumulation scheme adds T cycles

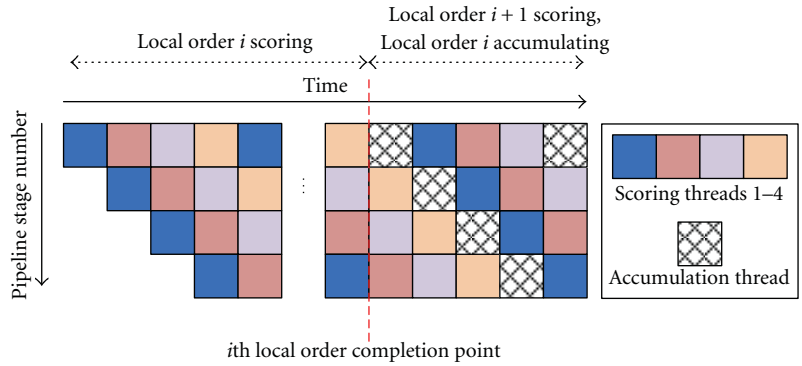

Figure 3: Thread accumulation over a 4-thread/stage core for two adjacent local orders.

of accumulation overhead to the scoring process, for a T-thread datapath and a single additional cycle for cross-core accumulation. To simplify the accumulation logic, we linearly reduce all threads across an A-core and then accumulate linearly across A-cores. The tradeoff here is that the last local order's accumulation is not hidden by another local order being scored and takes $T^2 + T * U_c$ cycles to finish, where U_c is the number of A-cores in the cluster.

Given sufficient hardware resources, more advanced A-core clusters can be built to further increase system throughput. First, the number of A-cores per RAM can be increased to the number of read ports each RAM has. Second, since a given node's data ($\mathcal{D}[n]$) does not change over the $\mathcal{R} * \mathcal{C}$ local orders, A-core chains can be replicated entirely. In this case, we say that the cluster has been split into two or more *lanes*, where A-cores from each lane are responsible for a different local order. In this setup, the cluster's control strip-mines local orders across lanes to initiate scoring. While scoring, corresponding A-cores (the first A-core in each of several lanes, e.g.) across the lanes (called *tiles*) read and process the same data from the same RAM data stream. An example of an advanced A-core cluster is shown in Figure 4.

The following analytic model can be used to estimate the parallel completion time to score $\mathcal{O}_l$ local orders over the $\mathcal{P}_f$ subset of the data (for a single cluster):

$$\text{Cycles}_{\text{DRAM}} + \frac{\mathcal{O}_l}{U_l} * \left(\frac{\mathcal{P}_f}{U_c} + (T+1)\right) + (T^2 + T * U_c), \tag{3}$$

where $\text{Cycles}_{\text{DRAM}}$ is the number of cycles (normalized to the core clock) required to initialize the cluster from DRAM, U_c is the number of A-cores per lane (doubles when two SRAM ports are used, etc.), U_l is the number of lanes per cluster, and T is the number of hardware threads per A-core.

5.1.2. Memory Subsystem. The scoring unit controls DRAM requests when an A-core cluster requires a different subset of the data. Regardless of problem parameters, data is always laid out contiguously in memory. As DRAM data is streamed to a finite number of RAMs, there must be enough RAM write bandwidth to consume the DRAM stream. In cases where the RAM write capability does not align to the DRAM read capacity, dedicated alignment circuitry built into the scoring unit dynamically realigns the data stream.

5.1.3. Control Unit. We implemented the MCMC control unit directly in hardware, according to Figure 5. The MCMC state machine, node swapping logic, parallel tempering logic, and Metropolis-Hasting logic is mapped as hardware state machines. Furthermore, a DSP block is used for multiplicative factors, while $log(rand(0,1))$ is implemented as a table lookup. The random generators for row/column swaps, as well as Metropolis-Hastings and parallel tempering, are built using free-running LFSRs.

At the start of each iteration, the control unit performs node swaps for each of the $\mathcal{R} * \mathcal{C}$ orders and schedules the proposed orders onto available compute units. To minimize control unit time when $\mathcal{R} * \mathcal{C}$ is small, orders are stored in row order in RAM, making the swap operation a single cycle row swap, followed by an $\mathcal{N}$ cycle column swap. Although the control unit theoretically has cycle accurate visibility of the entire system and can therefore derive optimal schedules, we found that using a trivial greedy scheduling policy (first come first serve) negligibly degrades performance with the benefit of significantly reducing hardware complexity. To minimize A-core cluster memory requirements, all $\mathcal{R} * \mathcal{C}$ local orders are scheduled to compute units in bulk over a single node.

When each iteration is underway, partial scores received from the scoring unit are accumulated as soon as they are received, using a dedicated A-core attached to a buffer that stores partial results. In practice, each A-core cluster can only store data for a part of a given node at a time. This means that the A-core, processing partial results, must perform both the slower *score*() operation and the simpler cross-node "+" accumulations. We determined that a single core dedicated to this purpose can rate match the results coming back from the compute-bound compute units.

At the end of each iteration, Metropolis-Hastings checks proceed in $[\mathcal{R}][\mathcal{C}]$ order. This allows the parallel tempering exchange operation for restart r to be interleaved with the Metropolis-Hastings check for restart $r+1$.

5.2. The MARC Architecture

5.2.1. Many-Core Template. The overall architecture of a MARC system, as illustrated in Figure 6, resembles a scalable, many-core-style processor architecture, comprising one

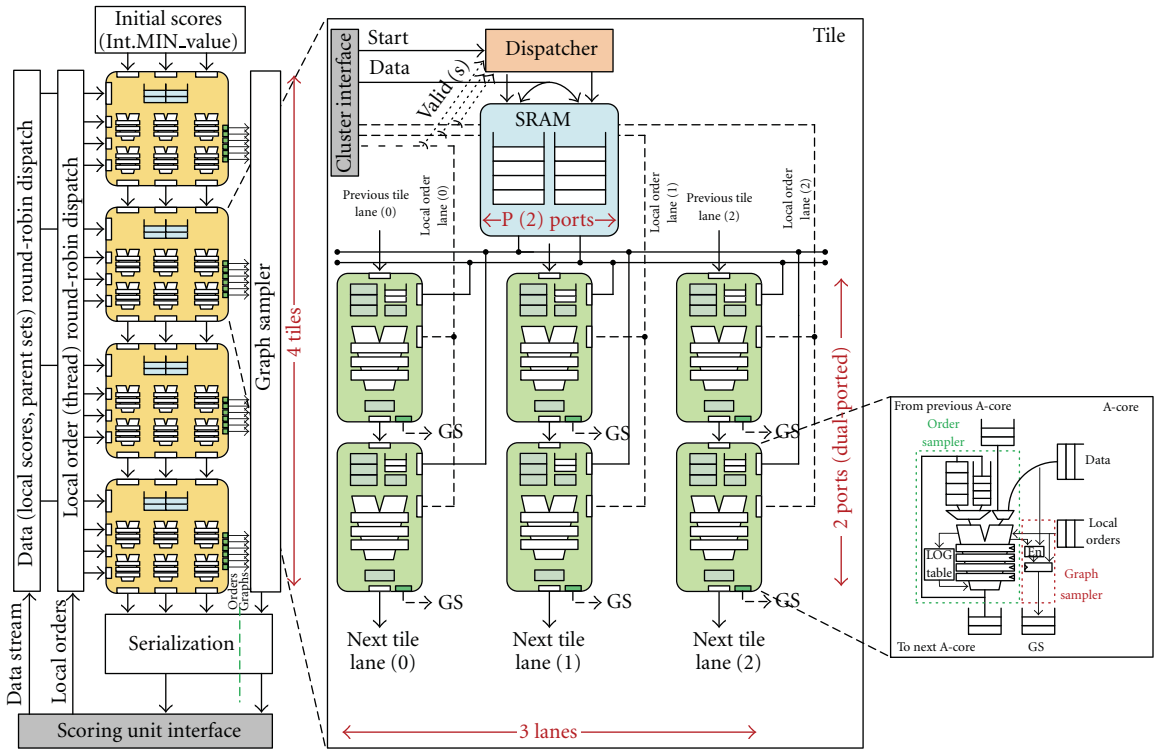

FIGURE 4: The hand-optimized A-core cluster. This example contains four tiles and three lanes and uses two RAM read ports per tile. "GS" stands for graph sampler.

Control Processor (C-core) and multiple Algorithmic Processing Cores (A-cores). Both the C-cores and the A-core can be implemented as conventional pipelined RISC processors. However, unlike embedded processors commonly found in modern SOCs, the processing cores in MARC are completely parameterized with variable bit width, reconfigurable multithreading, and even aggregate/fused instructions. Furthermore, A-cores can alternatively be synthesized as fully customized datapaths. For example, in order to hide global memory access latency, improve processing node utilization, and increase the overall system throughput, a MARC system can perform fine-grained multithreading through shift register insertion and automatic retiming. Finally, while each processing core possesses a dedicated local memory accessible only to itself, a MARC system has a global memory space implemented as distributed memories accessible by all processing cores through the interconnect network. Communication between a MARC system and its host can be realized by reading and writing global memory.

5.2.2. Execution Model and Software Infrastructure. Our MARC system builds upon both LLVM, a production-grade open-source compiler infrastructure [22] and OpenCL.

Figure 7 presents a high-level schematic of a typical MARC machine. A user application runs on a host according to the models native to the host platform—a high-performance PC in our study. Execution of a MARC program occurs in two parts: kernels that run on one or more A-cores of the MARC devices and a control program that runs on the C-core. The control program defines the context for the kernels and manages their execution. During the execution, the MARC application spawns kernel threads to run on the A-cores, each of which runs a single stream of instructions as SPMD units (each processing core maintains its own program counter).

5.2.3. Application-Specific Processing Core. One strength of MARC is its capability to integrate fully customized application-specific processing cores/datapaths so that the kernels in an application can be more efficiently executed. To this end, a high-level synthesis flow depicted by Figure 8 was developed to generate customized datapaths for a target application.

The original kernel source code in C/C++ is first compiled through `llvm-gcc` to generate the intermediate representation (IR) in the form of a single static assignment graph (SSA), which forms a control flow graph where instructions are grouped into basic blocks. Within each basic block, the instruction parallelism can be extracted easily as all false dependencies have been removed in the SSA representation. Between basic blocks, the control dependencies can then be transformed to data dependencies through branch predication. In our implementation, only memory operations are predicated since they are the only instructions that can generate stalls in the pipeline. By converting the control dependencies to data dependencies, the boundaries between basic blocks can be eliminated. This results in a single data flow graph with each node corresponding to a single instruction

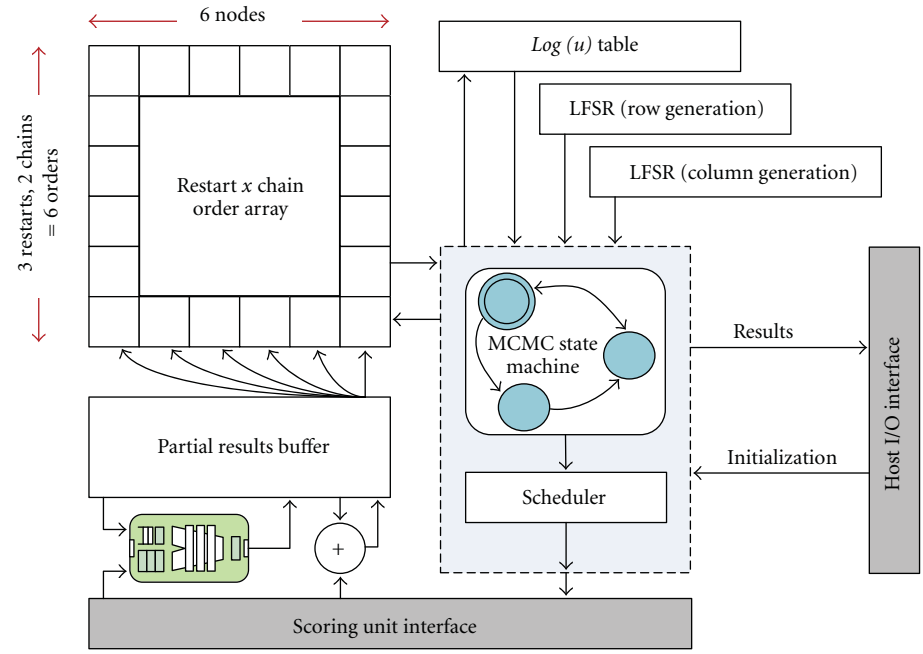

Figure 5: The hand-optimized control unit.

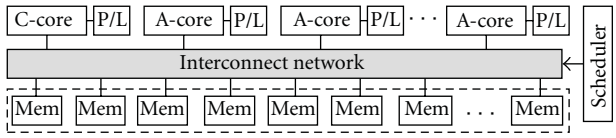

Figure 6: Diagram of key components in a MARC machine.

in the IR. Creating hardware from this graph involves a one-to-one mapping between each instruction and various predetermined hardware primitives. To utilize loop level parallelism, our high-level synthesis tool also computes the minimal interval at which a new iteration of the loop can be initiated and subsequently generates a controller to pipeline loop iterations. Finally, the customized cores have the original function arguments converted into inputs. In addition, a simple set of control signals is created to initialize a C-core and to signal the completion of the execution. For memory accesses within the original code, each nonaliasing memory pointer used by the *C* function is mapped to a memory interface capable of accommodating variable memory access latency. The integration of the customized cores into a MARC machine involves mapping the input of the cores to memory addresses accessible by the control core, as well as the addition of a memory handshake mechanism allowing cores to access global and local memories. For the results reported in this paper, the multithreaded customized cores are created by manually inserting shift registers into the single-threaded, automatically generated core.

5.2.4. Host-MARC Interface. Gigabit Ethernet is used to implement the communication link between the host and the MARC device. We leveraged the GateLib [23] project from Berkeley to implement the host interface, allowing the physical transport to be easily replaced by a faster medium in the future.

5.2.5. Memory Organization. Following OpenCL, A-core threads have access to three distinct memory regions: private, local, and global. Global memory permits read and write access to all threads within any executing kernels on any processing core (ideally, reads and writes to global memory may be cached depending on the capabilities of the device, however in our current MARC machine implementation, caching is not supported). Local memory is a section of the address space shared by the threads within a computing core. This memory region can be used to allocate variables that are shared by all threads spawned from the same computing kernel. Finally, private memory is a memory region that is dedicated to a single thread. Variables defined in one thread's private memory are not visible to another thread, even when they belong to the same executing kernel.

Physically, the private and local memory regions in a MARC system are implemented using on-chip memories. Part of the global memory region also resides on-chip, but we allow external memory (i.e., through the DRAM controller) to extend the global memory region, resulting in a larger memory space.

5.2.6. Kernel Scheduler. To achieve high throughput, kernels must be scheduled to avoid memory access conflicts. The MARC system allows for a globally aware kernel scheduler, which can orchestrate the execution of kernels and control access to shared resources. The global scheduler is controlled via a set of memory-mapped registers, which are implementation specific. This approach allows for a range of

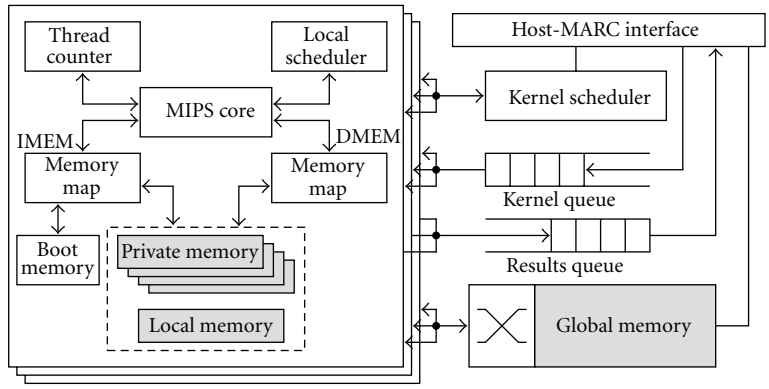

Figure 7: Schematic of a MARC machine's implementation.

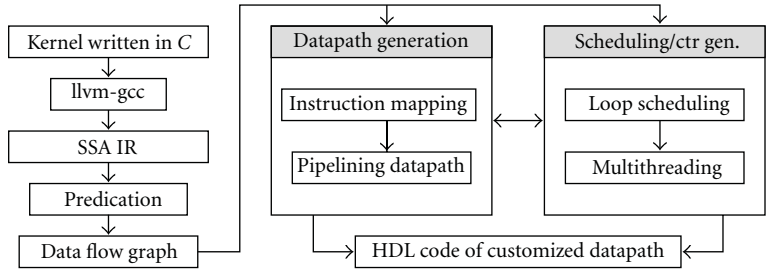

Figure 8: CAD flow of synthesizing application-specific processing cores.

schedulers, from simple round-robin or priority schedules to complex problem-specific scheduling algorithms.

The MARC machine optimized for Bayesian inference uses the global scheduler to dispatch threads at a coarse grain (ganging up thread starts). The use of the global scheduler is therefore rather limited as the problem does not greatly benefit from a globally aware approach to scheduling.

5.2.7. System Interconnect. One of the key advantages of reconfigurable computing is the ability to exploit application-specific communication patterns in the hardware system. MARC allows the network to be selected from a library of various topologies, such as mesh, H-tree, crossbar, or torus. Application-specific communication patterns can thus be exploited by providing low-latency links along common routes.

The MARC machine explores two topologies: a pipelined crossbar and a ring, as shown in Figure 9. The pipelined crossbar contains no assumptions about the communication pattern of the target application—it is a nonblocking network that provides uniform latency to all locations in the global memory address space. Due to the large number of endpoints on the network, the crossbar is limited to 120 MHz with 8 cycles of latency.

The ring interconnect only implements nearestneighbor links, thereby providing very low-latency access to some locations in global memory, while requiring multiple hops for other accesses. Nearest neighbor communication is important in the Bayesian inference accumulation phase and helps reduce overall latency. Moreover, this network topology is significantly more compact and can be clocked at a much higher frequency—approaching 300 MHz in our implementations. The various versions of our MARC machine, therefore, made use of the ring network because of the advantages it has shown for this application.

5.2.8. Mapping Bayesian Inference onto the MARC Machine. The order-graph sampler comprises a C-core for the serial control logic and A-cores to implement the *score*() calls. Per iteration, the C-core performs the node swap operation, broadcasts the proposed order, and applies the Metropolis-Hastings check. These operations consume a negligible amount of time relative to the scoring process.

Scoring is composed of (1) the parent set compatibility check and (2) an accumulation across all compatible parent sets. Step 1 must be made over every parent set; its performance is limited by how many parent sets can be simultaneously accessed. We store parent sets in on-chip RAMs that serve as A-core private memory and are therefore limited by the number of A-cores and attainable A-core throughput. Step 2 must be first carried out independently by each A-core thread, then across A-core threads, and finally across the A-cores themselves. We serialize cross-thread and cross-core accumulations. Each accumulation is implemented with a global memory access.

The larger order-graph sampler benchmark we chose (see Section 7) consists of up to 37 nodes, where each of the nodes has 66712 parent sets. We divide these 66712 elements into 36 chunks and dedicate 36 A-cores to work on this data set. After completion of the data processing for one node, data from the next node is paged in, and we restart the A-cores.

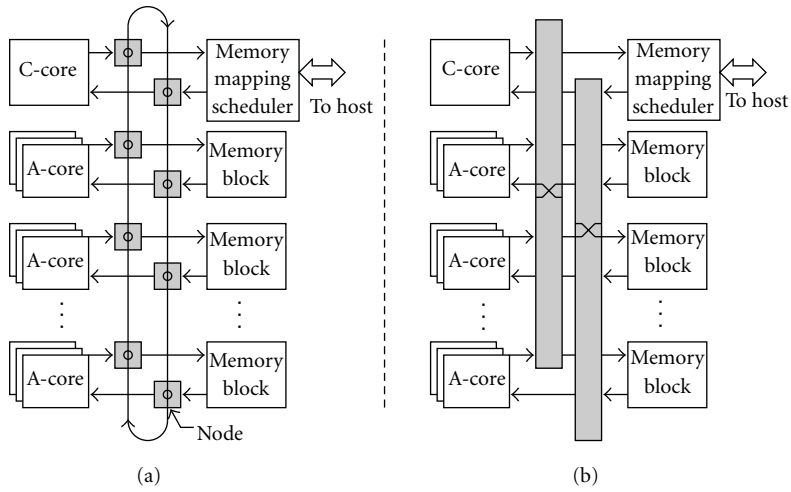

Figure 9: System diagram of a MARC system. (a) Ring network. (b) Pipelined crossbar.

6. Hardware Prototyping

For this research, both the hand-optimized design and MARC machines are implemented targeting a Virtex-5 (XCV5LX155T-2) of a BEEcube BEE3 module for FPGA prototyping. We also evaluate how each design performs when mapped through a standard ASIC design flow targeting a TSMC 65 ns CMOS process. A design point summary, that we will develop over the rest of the paper, is given in Table 1.

The local memory or "RAMs", used in each design point, were implemented using block RAMs (BRAMs) on an FPGA and generated as SRAM (using foundry-specific memory generators) on an ASIC. All of our design points benefit from as much local memory read bandwidth as possible. We increased read bandwidth on the FPGA implementation by using both ports of each BRAM block and exposing each BRAM as two smaller single-port memories. For the ASIC platform, the foundry-specific IP generator gives us the capability to create small single-ported memories suitable for our use.

In addition to simple memories, our designs used FIFOs, arbiters, and similar hardware structures to manage flow control and control state. On an FPGA, most of these blocks were available on the Virtex-5 through Xilinx Coregen while the rest were taken from the GateLib library. On an ASIC, all-of these blocks were synthesized from GateLib Verilog or generated using foundry tools.

To obtain all FPGA measurements, we designed in Verilog RTL and mapped the resulting system using Synplify Pro (Synopsys) and the Xilinx ISE flow for placement and routing (PAR). To obtain ASIC measurements, we used a standard cell-bawed Synopsis CAD flow including Design Compiler and IC Compiler.

No manual placement or hierarchical design was used for our studies. We verified the resulting system post-PAR by verifying (a) timing closure, and (b) functionality of the flattened netlist. The tools were configured to automatically retime the circuit to assist timing closure, at the expense of hard-ware resources. It is worth noting that the automatic retiming did not work as well with the MARC multithreaded cores because of a feedback path in the core datapath. Therefore, manual retiming was required for performance improvement with the MARC multithreaded design points.

6.1. Hand-Optimized Configurations. On the FPGA platform, the best performing configurations were attained when using 48 cores per FPGA running at a 250 MHz core clock and 36 cores at 300 MHz (the former outperforming the latter by a factor of 1.1 on average). Both of these points were used between 65% and 75% of available device LUTs and exactly 95% of device BRAMs. We found that implementing 48 cores at 300 MHz was not practical due to routing limitations and use of the 48 core version at 250 MHz for the rest of the paper.

For the ASIC implementation, because performance is a strong function of the core clock's frequency, we optimize the core clock as much as possible. By supplying the Verilog RTL tp the Synopsys Design Compiler with no prior optimization, the cores can be clocked at 500 MHz. Optimizing the core clock requires shortening the critical path, which is in the datapath. By increasing the number of threads from 4 to 8 and performing manual retiming for the multithreaded datapath, the core clock achieves 1 GHz.

6.2. MARC Configurations. The MARC implementation comprises one C-core and 36 A-cores. While the C-core in all MARC machines is a fully bypassed 4-stage RISC processor, MARC implementations differ in their implementation of the A-cores. For example, fine-grained multithreaded RISC cores, automatically generated application-specific datapaths, and multithreaded versions of the generated cores are all employed to explore different tradeoffs in design effort and performance. To maintain high throughput, the better performing A-cores normally execute multiple concurrent threads to saturate the long cycles in the application dataflow graph.

6.2.1. Memory System. As in other computing platforms, memory accesses significantly impact the overall performance of a MARC system. In the current MARC implemen-

Table 1: A-core counts, for all design points, and a naming convention for all MARC configurations used in the study. If only one core count is listed, it is the same for both 32 and 37 node ($32n$ and $37n$) problems (see Section 7). All A-core counts are given for area normalized designs, as discussed in Section 7.

Alias	Description	Number of cores ($32n$, $37n$)
Hand design FPGA	—	48
Hand design ASIC	—	2624, 2923
MARC-Ropt-F	RISC A-core with optimized network on FPGA	36
MARC-C1-F	Customized A-core on FPGA	36
MARC-C2-F	Customized A-core (2-way MT) on FPGA	36
MARC-C4-F	Customized A-core (4-way MT) on FPGA	36
MARC-Ropt-A	RISC A-core with optimized network on ASIC	1269, 1158
MARC-C1-A	Customized A-core on ASIC	1782, 1571
MARC-C2-A	Customized A-core (2-way MT) on ASIC	1768, 1561
MARC-C4-A	Customized A-core (4-way MT) on ASIC	1715, 1519
GPGPU	—	512

tation, private or local memory accesses take exactly one cycle, while global memory accesses typically involve longer latencies that are network dependent. We believe that given different applications, such discrepancies between local and global memory access latencies provide ample opportunities for memory optimization and performance improvements. For example, the MARC machine in this work has been optimized for local memory accesses, reflecting the needs of the Bayesian inference algorithm.

6.2.2. Clock Speed and Area on FPGA and ASIC. For better throughput, we have implemented single-threaded, two-way multithreaded, and four-way multithreaded application-specific A-cores for MARC devices. When individually instantiated on the Virtex-5 FPGA, these cores are clocked at 163 MHz, 234 MHz, and 226 MHz, respectively. There is a decrease in clock frequency when the number of threads is changed from two to four. This is due to the increased routing delay to connect LUT-FF pairs further apart in a larger physical area. When the completely assembled MARC machines traverse the hardware generation flow, the cores' clock frequency decreases further to 144 MHz, 207 MHz, and 206 MHz, respectively due to added FPGA resource utilization. The same A-cores are used for the ASIC implementation, where they operate at 526 MHz, 724 MHz, and 746 MHz, respectively. Due to a higher degree of freedom in ASIC place and route, we do not see the performance dip observed when the two-threaded FPGA implementation is changed to four-threaded. However, it is apparent that despite the decrease in levels of logic in the critical path, it is difficult to squeeze out more performance by simple register insertion and retiming.

With respect to area, the overhead of multithreading is more pronounced on an FPGA relative to an ASIC. For the 37 node benchmark, the MARC machines with single, two-way, and four-way multithreaded customized A-cores utilize 47%, 65%, and 80% of the flip-flops on Virtex-5. Since they operate on the same amount of data, 85% of BRAMs are used for each of the three design points. Meanwhile, on an ASIC we only observe an area increase from 6.2 mm^2 in the single-threaded case to 6.4 mm^2 for the four-way multithreaded design. This is because the ASIC implementation exposes the actual chip area, where the increase in number of registers is dwarfed by the large SRAM area.

7. Performance and Area Comparisons

We compare the performance of the hand-optimized design and the MARC machines on FPGA as well as ASIC platforms. For both the hand-optimized and the MARC implementations on an ASIC, we normalize our area to the FPGA's die area. FPGA die area was obtained by X-ray imaging the packaged dies and estimating the die area from the resulting photographs. For the remainder of the paper, all devices whose die areas and process nodes are relevant are given in Table 2.

For the FPGA designs, we packed the device to its limits without performance degradation. Effectively, the designs are consuming the entire area of the FPGA. We then performed a similar evaluation for the ASIC platform by attempting to occupy the same area as an FPGA. This is achieved by running the design for a small number of cores and then scaling up. This technique is valid as the core clock is not distributed across the network, and the network clock can be slow (50–100 MHz) without adversely affecting performance.

The specific Bayesian network instances we chose consist of 32 and 37 nodes, with dataset of 36457 and 66712 elements, respectively. The run times on each hardware platform are shown in Tables 4 and 5, for the 32 and 37 node-problem, respectively. The execution time for each platform is also normalized to the fastest implementation—hand-optimized design on ASIC—to show the relative performance of every design point.

7.1. Benchmark Comparison. The large gap between the amount of data involved in the two problems gives each distinct characteristics, especially when mapped to an ASIC platform. Because data for the 32 node problem can fit on an ASIC for both MARC and the hand-optimized design, the problem is purely compute bound. The hand-optimized solution benefits from the custom pipelined accumulation

Table 2: Device die areas and process nodes.

Device	Die area (mm^2)	Process (nm)
Virtex-5 LX155T FPGA	270	65
Nvidia GeForce GTX 580	520	40

and smaller and faster cores, resulting in its 2.5x performance advantage over the best MARC implementation. The 37 node problem, on the other hand, could not afford to have the entire dataset in the on-chip SRAMs. The required paging of data into the on-chip RAMs becomes the performance bottleneck. Having exactly the same DRAM controller as the MARC machines, the hand-optimized design only shows a small performance advantage over MARC, which can be attributed to its clever paging scheme. For the FPGA platform, both the 32 and 37 node problems involve paging of data, but as the run time is much longer, data transfer only accounts for a very small fraction of the execution time (i.e., both problems are compute bound).

7.2. MARC versus Hand-Optimized Design. For compute-bound problems, it is clear that MARC using RISC instruction processors to implement A-cores achieves less than 2% of the performance exhibited by the hand-optimized implementation, even with optimized interconnect topology (a ring network versus a pipelined crossbar). Customizing the A-cores, however, yields a significant gain in performance, moving MARC to within a factor of 4 of the performance of the hand-optimized implementation. Further optimizing the A-cores through multithreading pushes the performance even higher. The best performing MARC implementation is within a factor of 2.5 of the hand-optimized design and corresponds to two-way multithreaded A-cores. Like the FPGA platform, further increase to four threads offers diminishing returns and is outweighed by the increase in area, and therefore the area-normalized performance actually decreases.

7.3. Cross-Analysis against GPGPU. We also benchmark the various hardware implementations of the order-graph sampler against the GPGPU reference solution, running on Nvdia's GeForce GTX 580.

As the GTX 580 chip has a much larger area than Virtex-5 FPGA and is also on 40 nm process rather than 65 nm, we scaled its execution time according to the following equations, following Table 2:

$$\text{Scaled Area}_{\text{GPU}} = \text{Area}_{\text{GPU}} * S^2 = 520 * \left(\frac{65}{40}\right)^2 = 1373, \tag{4}$$

$$T_{\text{scaled}} = \frac{\text{Scaled Area}_{\text{GPU}}}{\text{Area}_{\text{FPGA}}} * S * T = 8.264 * T. \tag{5}$$

To make sure the comparison is fair, the technology scaling [24] takes into account the absolute area difference between the GPU and FPGA, as well as the area and delay scaling (i.e., S, the technology scaling factor) due to different processes. Our first assumption is that the performance scales linearly with area, which is a good approximation due to our Bayesian network problem and device sizes. Second, we assume zero wire slack across both process generations for all designs. The original and scaled execution times are displayed in Table 3.

Table 3: Scaled GPGPU design for 65 nm process.

Problem	Per-iteration time 40 nm (μs)	Scaled per-iteration time 65 nm (μs)
32-Node	21.0	174
37-Node	37.8	312

It can be seen from Tables 4 and 5 that MARC on FPGA can achieve the same performance as the GPGPU when application-specific A-cores are used. With multithreading, the best MARC implementation on FPGA can achieve more than a 40% performance advantage over the GPGPU. Hand-optimized designs, with more customization at the core and network level, push this advantage even further to 3.3x. The reason for this speedup is that each iteration of the inner loop of the *score*() function takes 1 cycle for A-cores on MARC and the hand-optimized design, but 15 to 20 cycles on GPGPU cores. It is apparent that the benefit from exploiting loop level parallelism at the A-cores outweighs the clock frequency advantage that the GPGPU has over the FPGA.

When an ASIC is used as the implementation platform, the speedup is affected by the paging of data as explained in Section 7.1. For the 32 node problem where paging of data is not required, the best MARC implementation and the hand-optimized design achieve 156x and 412x performance improvement over the GPGPU, respectively. For the 37 node problem, which requires paging, we observe a 69x and 84x performance advantage from the best MARC variant and hand-optimized implementation, respectively. Using only a single dual channel DRAM controller, we have about 51.2 Gb/sec of memory bandwidth for paging. However the GPGPU's memory bandwidth is 1538.2 Gb/sec—30x that of our ASIC implementations. As a result, the GPGPU solution remains compute bound while our ASIC implementations are getting constrained by the memory bandwidth. Thus, the performance gap between the 32 and 37 node problems is because of the memory-bound nature of our ASIC implementations.

It is also interesting that the MARC implementation with RISC A-cores on ASIC is about 6 times faster for the 32 node problem and 8 times faster for 37 node problem, compared to the GPGPU. With both MARC RISC A-cores and GPGPU cores, the kernel is executed as sequence of instructions rather than by a custom datapath. In addition, the clock frequency gap between MARC on ASIC and the GPGPU is small. We claim that the performance gap is due to the application-specific nature of the MARC design—MARC is able to place more cores per unit area (see Table 1) while still satisfying the requirements of local caching. In addition, the network structure in MARC machines is also optimized to the Bayesian inference accumulation step. The combined effect results in a significantly better use of chip area for this application.

Table 4: 32-Node. Performance comparison between MARC, hand-optimized, and GPGPU.

Configuration	Per iteration Time (μs)	Relative Perf.
GPGPU scaled reference	174	0.0024
MARC-Ropt-F	2550	0.0002
MARC-C1-F	172	0.0025
MARC-C2-F	124	0.0034
MARC-C4-F	136	0.0031
Hand design FPGA	51.4	0.0082
MARC-Ropt-A	27.6	0.0152
MARC-C1-A	1.47	0.2863
MARC-C2-A	1.11	0.3808
MARC-C4-A	1.17	0.3608
Hand design ASIC	0.422	1.0000

Table 5: 37-Node. Performance comparison between MARC, Hand-optimized, and GPGPU.

Configuration	Per iteration Time (μs)	Relative Perf.
GPGPU scaled reference	312	0.0119
MARC-Ropt-F	5130	0.0007
MARC-C1-F	310	0.0120
MARC-C2-F	221	0.0169
MARC-C4-F	235	0.0158
Hand design FPGA	110	0.0339
MARC-Ropt-A	38.1	0.0978
MARC-C1-A	5.02	0.7429
MARC-C2-A	4.53	0.8231
MARC-C4-A	4.61	0.8083
Hand design ASIC	3.73	1.0000

8. Conclusion

MARC offers a methodology to design FPGA and ASIC-based high-performance reconfigurable computing systems. It does this by combining a many-core architectural template, high-level imperative programming model [19], and modern compiler technology [22] to efficiently target both ASICs and FPGAs for general-purpose, computationally intensive data-parallel applications.

The primary objective of this paper is to understand whether a many-core architecture is a suitable abstraction layer (or execution model) for designing ASIC and FPGA-based computing machines from an OpenCL specification. We are motivated by recently reemerging interest and efforts in parallel programming for newly engineered and upcoming many-core platforms, and feel that if we can successfully build an efficient many-core abstraction for ASICs and FPGAs, we can apply the advances in parallel programming to high-level automatic synthesis of computing systems. Of-course, constraining an execution template reduces degrees of freedom for customizing an implementation using application-specific detail. However, we work under the hypothesis that much of the potential loss in efficiency can be recovered through customization of a microarchitectural template designed for a class of applications using application-specific information. The study in this paper represents our initial effort to quantify the loss in efficiency incurred for a significant gain in design productivity for one particular application.

We have demonstrated via the use of a many-core microarchitectural template for OpenCL that it is at least sometimes possible to achieve competitive performance relative to a highly optimized solution and to do so with considerable reduction in development effort (days versus months). This approach also achieves significant performance advantage over a GPGPU approach—a natural platform for mapping this class of applications. In this study, the most significant performance benefit came from customization of the processor cores to better fit the application kernel—an operation within reach of modern high-level synthesis flows.

Despite these results, the effectiveness of MARC in the general case remains to be investigated. We are currently limited by our ability to generate many high-quality hand-optimized custom solutions in a variety of domains to validate and benchmark template-based implementations. Nonetheless, we plan to continue this study, exploring more application domains, extending the many-core template tailored for OpenCL and exploring template microarchitectures for other paradigms. We are optimistic that a MARC-like approach will open new frontiers for rapid prototyping of high-performance computing systems.

Acknowledgments

The authors wish to acknowledge the contributions of the students, faculty, and sponsors of the Berkeley Wireless Research Center and the TSMC University Shuttle Program. This work was funded by the NIH, Grant no. 1R01CA130826-01 and the Department of Energy, Award no. DE-SC0003624.

References

[1] M. Lin, I. Lebedev, and J. Wawrzynek, "Highthroughput Bayesian computing machine with reconfigurable hardware," in *Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '10)*, pp. 73–82, ACM, Monterey, California, USA, 2010.

[2] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: from sea-of-gates to sea-of-cores," in *Proceedings of the 20th IEEE International Conference on Field Programmable Logic and Applications*, Milano, Italy, 2010.

[3] Wikipedia, "C-to-hdl," November 2009, http://en.wikipedia.org/wiki/C_to_HDL/.

[4] M. Gokhale and J. Stone, "Napa c: compiling for a hybrid risc/fpga architecture," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, Napa, Calif, USA, 1998.

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

[6] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI '04)*, pp. 14–26, New York, NY, USA, October 2004.

[7] J. Wawrzynek, D. Patterson, M. Oskin et al., "RAMP: research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.

[8] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and M. W. Hwu, "Fcuda: enabling efficient compilation of cuda kernels onto fpgas," in *Proceedings of the 7th IEEE Symposium on Application Specific Processors (SASP '09)*, San Francisco, Calif, USA, 2009.

[9] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from opencl programs," in *Proceedings of the 19th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, Salt Lake City, Utah, USA, 2011.

[10] J. Friedman, T. Hastie, and R. Tibshirani, "Sparse inverse covariance estimation with the graphical lasso," *Biostatistics*, vol. 9, no. 3, pp. 432–441, 2008.

[11] D. Heckerman, D. Geiger, and D. M. Chickering, "Learning Bayesian networks: the combination of knowledge and statistical data," *Machine Learning*, vol. 20, no. 3, pp. 197–243, 1995.

[12] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, San Francisco, Calif, USA, 1988.

[13] C. Fletcher, I. Lebedev, N. Asadi, D. Burke, and J. Wawrzynek, "Bridging the GPGPU-FPGA efficiency gap," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, pp. 119–122, New York, NY, USA, 2011.

[14] N. Bani Asadi, C. W. Fletcher, G. Gibeling et al., "Paralearn: a massively parallel, scalable system for learning interaction networks on fpgas," in *Proceedings of the 24th ACM International Conference on Supercomputing*, pp. 83–94, ACM, Ibaraki, Japan, 2010.

[15] D. M. Chickering, "Learning Bayesian Networks is NP-Complete," in *Learning from Data: Artificial Intelligence and Statistics V*, pp. 121–130, Springer, New York, NY, USA, 1996.

[16] B. Ellis and W. H. Wong, "Learning causal Bayesian network structures from experimental data," *Journal of the American Statistical Association*, vol. 103, no. 482, pp. 778–789, 2008.

[17] M. Teyssier and D. Koller, "Ordering-based search: a simple and effective algorithm for learning Bayesian networks," in *Proceedings of the 21st Conference on Uncertainty in AI (UAI '5)*, pp. 584–590, Edinburgh, UK, July 2005.

[18] N. Friedman and D. Koller, "Being Bayesian about network structure," in *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pp. 201–210, Morgan Kaufmann, San Francisco, Calif, USA, 2000.

[19] Khronos OpenCL Working Group, The OpenCL Specification, version 1.0.29, December 2008, http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[20] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: low-power high-performance computing with reconfigurable devices," in *Proceedings of the 18th International Symposium on Field Programmable Gate Array*, 2010.

[21] NVIDIA OpenCL Best Practices Guide, 2009, http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[22] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, pp. 75–86, Palo Alto, Calif, USA, March 2004.

[23] G. Gibeling et al., "Gatelib: a library for hardware and software research," Tech. Rep., 2010.

[24] J. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*, chapter 5, Prentice Hall, New York, NY, USA, 2nd edition, 2003.