Lecture Notes in Computer Science 5992

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Phaophak Sirisuk Fearghal Morgan
Tarek El-Ghazawi Hideharu Amano (Eds.)

Reconfigurable Computing: Architectures, Tools and Applications

6th International Symposium, ARC 2010
Bangkok, Thailand, March 17-19, 2010
Proceedings

Springer

Volume Editors

Phaophak Sirisuk
Mahanakorn University of Technology
Department of Electronic Engineering
Bangkok 10530, Thailand
E-mail: phaophak@mut.ac.th

Fearghal Morgan
National University of Ireland
Department of Electronic Engineering
Galway, Ireland
E-mail: fearghal.morgan@nuigalway.ie

Tarek El-Ghazawi
The George Washington University
Department of Electrical and Computer Engineering
Washington, DC, 20052, USA
E-mail: tarek@gwu.edu

Hideharu Amano
Keio University
Department of Information and Computer Science
Yokohama, Kanagawa, 223–8522, Japan
E-mail: hunga@am.ics.keio.ac.jp

Library of Congress Control Number: Applied for

CR Subject Classification (1998): C.2, D.2, I.4, H.3, F.1, I.6

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-12132-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-12132-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

Reconfigurable computing (RC) systems have generated considerable interest in the embedded and high-performance computing communities over the past two decades, with field programmable gate arrays (FPGAs) as the leading technology at the helm of innovation in this discipline. Achieving orders of magnitude performance and power improvements using FPGAs over traditional microprocessors is not uncommon for well-suited applications. But even with two decades of research and technological advances, FPGA design still presents a substantial challenge and often necessitates hardware design expertise to exploit its true potential. Although the challenges to address the design productivity issues are steep, the promise and the potential of the RC technology in terms of performance, power, size, and versatility continue to attract application design engineers and RC researchers alike.

The International Symposium on Applied Reconfigurable Computing (ARC) aims to bring together researchers and practitioners of RC systems with an emphasis on practical applications and design methodologies of this promising technology. This year's ARC symposium (The sixth ARC symposium) was held in Bangkok, Thailand during March 17–19, 2010, and attracted papers in three primary focus areas: RC applications, RC architectures, and RC design methodologies. A total of 71 papers were submitted to the symposium from 23 countries: Japan (11), Germany (7), UK (6), Spain (6), Belgium (4), China (4), The Netherlands (4), Thailand (4), France (3), Republic of Korea (3), Singapore (3), Canada (2), Republic of India (2), Isle of Man (2), USA (2), Austria (1), Denmark (1), Greece (1), Islamic Republic of Iran (1), Malaysia (1), Myanmar (1), Poland (1), and Tunisia (1). This distribution is reflective of the international engagement in the disciplines related to RC systems.

In all cases, submitted papers were evaluated by at least three members of the Program Committee. After careful selection, 26 papers were accepted as full papers (acceptance rate of 36.6%) and 16 as short papers (global acceptance rate of 59.1%). Out of the total 42 accepted papers, the topic breakdown is as follows: practical applications of the RC technology(17), RC architectures(11), RC design methodologies and tools(13), and RC education(1). The diversity of results and research presented at the symposium led to a very interesting program, which we consider to constitute a representative overview of the on-going research efforts in this field. This LNCS volume includes all accepted papers.

We would like to extend our gratitude to all authors who submitted research papers to the symposium. We would also like to acknowledge the support and contribution of the Steering and Program Committee members towards reviewing papers, paper selection, and offering valuable suggestions and guidance. We thank the Organizing Committee members for their untiring efforts toward making this year's ARC symposium a grand success. We also thank Springer for their

continued support of this event. Special thanks are due to the distinguished invited speakers for their contributions to the technical program.

January 2010

Phaophak Sirisuk
Fearghal Morgan
Tarek El-Ghazawi
Hideharu Amano

Organization

Organizing Committee

General Chairs:	Phaophak Sirisuk (MUT, Thailand) Fearghal Morgan (National University of Ireland, Galway, Ireland)
Program Chairs:	Hideharu Amano (Keio University, Japan) Tarek El-Ghazawi (The George Washington University, USA)
Local Chair:	Theerayod Wiangtong (MUT, Thailand)
Publicity:	Philip H.W. Leong (University of Sydney, Australia) Piyawut Srichaikul (NECTEC, Thailand) Surin Kittitorakul (KMITL, Thailand) Akkarat Boonpoonga (MUT, Thailand)
Local Arrangements:	Peerapol Yuvapoositanon (MUT, Thailand) Supakorn Siddichai (NECTEC, Thailand) Suchada Sitjongsataporn (MUT, Thailand)
Proceedings Chair:	Sampan Prompichai (MUT, Thailand) Panan Poputhipong (MUT, Thailand)
Review Process Chair:	Saumil Merchant (The George Washington University, USA)

Steering Committee

George Constantinides	Imperial College, UK
Hideharu Amano	Keio University, Japan
João M. P. Cardoso	University of Porto/INESC-ID, Portugal
Juergen Becker	Universitaet Karlsruhe (TH), Germany
Katherine Compton	University of Wisconsin-Madison, USA
Koen Bertels	Delft University of Technology, The Netherlands
Mladen Berekovic	Braunschweig University of Technology, Germany
Pedro C. Diniz	Technical University of Lisbon (IST) / INESC-ID, Portugal
Philip H.W. Leong	University of Sydney, Australia
Roger Woods	The Queen's University of Belfast, UK
Walid Najjar	University of California Riverside, USA

Program Committee

Hideharu Amano	Keio University, Japan
Peter Athanas	Virginia Tech, USA

Michael Attig	Xilinx Research Labs, San Jose, USA
Nader Bagherzadeh	University of California, Irvine, USA
Jürgen Becker	Universität Karlsruhe (TH), Germany
Khaled Benkrid	University of Edinburgh, UK
Mladen Berekovic	Braunschweig University of Technology, Germany
Neil Bergmann	University of Queensland, Australia
Koen Bertels	Delft University of Technology, The Netherlands
Christos-Savvas Bouganis	Imperial College London, UK
Joao M. P. Cardoso	University of Porto/INESC-ID, Portugal
Mark Chang	Olin College, USA
Paul Chow	University of Toronto, Canada
Katherine Compton	University of Wisconsin-Madison, USA
George Constantinides	Imperial College London, UK
Oliver Diessel	University of New South Wales, Australia
Pedro C. Diniz	Technical University of Lisbon (IST) / INESC-ID, Portugal
Tarek El-Ghazawi	The George Washington University, USA
Robert Esser	Apple Inc., USA
Suhaib Fahmy	Nanyang Technological University, Singapore
Antonio Ferrari	University of Aveiro, Portugal
Brendan Glackin	University of Ulster, Magee, UK
Guy Gogniat	Université de Bretagne Sud, France
Jim Harkin	University of Ulster, Magee, UK
Reiner Hartenstein	University of Kaiserslautern, Germany
Michael Hübner	Karlsruhe Institute of Technology (KIT), Germany
Ryan Kastner	University of California, San Diego, USA
Andreas Koch	TU Darmstadt, Germany
Philip Leong	University of Sydney, Australia
Eduardo Marques	University of São Paulo, Brazil
Liam Marnane	University College Cork, Ireland
Kostas Masselos	University of the Peloponnese, Greece
John McAllister	Queen's University of Belfast, Ireland
Seda Ö. Memik	Northwestern University, USA
Saumil Merchant	The George Washington University, USA
Fearghal Morgan	National University of Ireland, Galway, Ireland
Walid Najjar	University of California Riverside, USA
Vikram Narayana	The George Washington University, USA
Horácio Neto	Technical University of Lisbon (IST) INESC-ID, Portugal
Joon-seok Park	Inha University, Seoul, South Korea
Andy Pimentel	University of Amsterdam, The Netherlands
Joachim Pistorius	Altera Corp., USA

Marco Platzner	University of Paderborn, Germany
Tsutomu Sasao	Kyushu Institute of Technology, Japan
Yuichiro Shibata	Nagasaki University, Japan
Alastair Smith	Imperial College London, UK
Pedro Trancoso	University of Cyprus, Cyprus
Ranga Vemuri	University of Cincinnati, USA
Markus Weinhardt	PACT Informationstechnologie AG, Germany
Stephan Wong	Delft University of Technology, The Netherlands
Roger Woods	The Queen's University of Belfast, UK

Table of Contents

Keynotes (Abstracts)

Session 1: Architectures 1

Session 2: Applications 1

Session 3: Architectures 2

Session 4: Applications 2

Session 5: Design Tools 1

Session 6: Design Tools 2

Session 7: Applications 3

Posters

High-Performance Energy-Efficient Reconfigurable Accelerators/Co-processors for Tera-Scale Multi-core Microprocessors

Ram Krishnamurthy

Senior Principal Engineer, Intel Corp., OR, USA
ram.krishnamurthy@intel.com

Abstract. With the emergence of high-performance multi-core microprocessors in the sub-45nm technology era, specialized hardware accelerator engines embedded within the core architecture have the potential to achieve 10-100X increase in energy efficiency across a wide domain of compute-intensive signal processing and scientific algorithms. In this talk, we present multi-core microprocessors integrated with on-die energy-efficient reconfigurable accelerator and co-processor engines to achieve well beyond tera-scale performance in sub-45nm technologies. Recent trends and advances in multi-core microprocessors will be presented, followed by key enablers for reconfigurability of specialized hardware engines to support multiple protocols while substantially improving time-to-market and amortizing die area cost across a wide range of compute workloads and functions. Specific design examples and case studies supported by silicon measurements will be presented to demonstrate reconfigurable engines for wireless baseband, signal processing and graphics/media applications. Power efficient optimization of reconfigurable processors to support fine-grain power management, dynamic on-the-fly configurability and standby-mode leakage reduction and low-voltage operability will also be described.

Process Variability and Degradation: New Frontier for Reconfigurable

Peter Y.K. Cheung

Professor and head of department,
Electrical and Electronic Engineering,
Imperial College,
London SW7 2AZ UK
p.cheung@imperial.ac.uk

Abstract. With the emergence of high-performance multi-core microprocessors in the sub-45nm technology era, specialized hardware accelerator engines embedded within the core architecture have the potential to achieve 10-100X increase in energy efficiency across a wide domain of compute-intensive signal processing and scientific algorithms. In this talk, we present multi-core microprocessors integrated with on-die energy-efficient reconfigurable accelerator and co-processor engines to achieve well beyond tera-scale performance in sub-45nm technologies. Recent trends and advances in multi-core microprocessors will be presented, followed by key enablers for reconfigurability of specialized hardware engines to support multiple protocols while substantially improving time-to-market and amortizing die area cost across a wide range of compute workloads and functions. Specific design examples and case studies supported by silicon measurements will be presented to demonstrate reconfigurable engines for wireless baseband, signal processing and graphics/media applications. Power efficient optimization of reconfigurable processors to support fine-grain power management, dynamic on-the-fly configurability and standby-mode leakage reduction and low-voltage operability will also be described.

Towards Analytical Methods for FPGA Architecture Investigation

Steven J.E. Wilton

Professor and Associate Head Academic,
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, B.C
stevew@ece.ubc.ca

Abstract. In the past 20 years, the capacity of FPGAs has grown by 200x and the speed has increased by 40x. Much of this dramatic improvement has been the result of architectural improvements. FPGA architectural enhancements are often developed in a somewhat ad-hoc manner. Expert FPGA architects perform experiments in which benchmark circuits are mapped using representative computer-aided design (CAD) tools, and the resulting density, speed, and/or power are estimated. Based on the results of these experiments, architects use their intuition and experience to design new architectures, and then evaluate these architectures using another set of experiments. This is repeated numerous times, until a suitable architecture is found.

During this process, there is virtually no body of theory that architects can use to speed up their investigations. Such insight, however, would be extremely valuable. A better understanding of the tradeoff between flexibility and efficiency may allow FPGA architects to uncover improved architectures quickly. Although it is unlikely that such an understanding would immediately lead to an optimum architecture, it may provide the means to "bound" the search space so that a wider variety of "interesting" architectures can be experimentally evaluated.

In this talk, I will describe recent work towards the development of such a theory. The current approach is to supplement the experimental methodology with a set of analytical expressions that relate architectural parameters to the area, speed, and power dissipation of an FPGA. Optimizing these analytical expressions is done using techniques such as geometric programming. I will summarize current research in this area, as well as try to provide some insight into how far we can go with these techniques.

Generic Systolic Array for Run-Time Scalable Cores

Andrés Otero, Yana E. Krasteva, Eduardo de la Torre, and Teresa Riesgo

Centro de Electrónica Industrial, Universidad Politécnica de Madrid
{andres.otero,yana.ekrasteva,eduardo.delatorre,
teresa.riesgo}@upm.es

Abstract. This paper presents a scalable core architecture based on a generic systolic array. The size of this kind of cores can be adapted in real-time to cover changing application requirements or to the available area in a reconfigurable device. In this paper, the process of scaling the core is performed by the replication of a single processing element using run-time partial reconfiguration. Furthermore, rather than restricting the proposed solution to a given application, it is based on a generic systolic architecture which is adapted using a design flow which is also proposed. The paper includes a related work discussion, the proposal and definition of a systolic array communication approach, which does not require the use of specific macro structures and permits to achieve higher flexibility, and a design flow used to adapt the generic architecture. Further, the paper also includes an image filter application as a simple use case, along with implementation results for Virtex 5 FPGA.

Keywords: Digital signal processing, adaptable cores, scalability, systolic array, partial runtime reconfiguration.

1 Introduction

Current multimedia applications are offered on heterogeneous terminals, with a broad range of features, using different communication networks and variable bandwidth availability capabilities [1]. As a result, single devices are supposed to deal with multiple coding standards, which evolve and emerge in short time. Consequently, devices lifetime is shortened and their replacement with new ones, with advanced features, requires speeding up time-to-market.

This challenge can be solved providing more flexibility to devices by including adaptability capabilities. Device adaptation can be based on different parameters, like the battery level, the available computational power, the target coding standard or even a profile within a standard. The need of adaptability could be easily fulfilled by the use of software implementations. However, most of the multimedia related tasks are compute-intensive and demand high performance and fast execution, which can be achieved in hardware. In this context, reconfigurable computing can fulfill both, performance and flexibility, requirements.

Among the flexibility requirements, there is a wide interest in proving solutions that permit to scale in real-time the functionality of a hardware block. Functional scaling is achieved by modifying the size of the operation performed by a core, depending on the application requirements at a given moment. Such solutions can be advantageous in many domains. Among others, in coding standards, where scaling is oriented

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 4–16, 2010.

to variable-size hardware operations, like the Discrete Wavelet Transform (DWT) presented in [2], the variable-size Discrete Cosine Transform (DCT) in [3] or in motion estimation and filters [4], and also, in tasks scaling for multi-standard communication systems [5] and [6].

This paper addresses a solution where the functional scalability of a hardware block is achieved by means of spatially scaling the physical implementation of a core. This means modifying the area occupied by the core inside a reconfigurable system. In addition, with this kind of solutions, different tradeoffs between the area occupied by a core and its performance can be set. An example of such system can be found in the scalable window based image filter proposed in [7], or in the scalable DCT presented in [8].

A direct approach to create variable-size scaling cores is to implement the same task in several cores, with different performance and area requirements, and load a suitable one in the system depending on the available hardware resources. Differently, highly parallel, modular and regular architectures have been studied as a scalable core architecture alternative to reduce the overhead of the adapting process. These architectures can be scaled by means of the addition and removal of parallel blocks resulting in lower adaptation times. Among the architectures with these characteristics, scalable cores based on systolic arrays and distributed arithmetic are the most common, like the ones presented in [9] and [10]. Distributed arithmetic provides scalable solutions to perform arithmetic operations, while systolic architectures can solve full computing-intensive tasks in a broad range of fields. An interesting summary of different systolic arrays, each one for a specific application field, can be found in [11].

There are several alternatives to implement the process of scaling a core, like the use of parameterizable HDL code, which results in different core implementations once synthesized [12], or to use a clock gating technique for the unused elements [3]. However, the first solution does not permit real-time adaptation, while the second one does not release the unused logic.). Therefore, the exploitation of partial run-time reconfiguration capabilities of state of the art Field Programmable Gate Arrays (FPGA) is the widely adopted solution, since it overcomes these limitations.

This paper focuses on systolic-array-based scalable cores that permit run-time adaptability. However, differently from the related work discussed in the paper, it presents a general systolic architecture that can be customized, using a proprietary design flow, to solve concrete problems. The proposed solution permits, using a single processing element replication process, to scale the functionality of a core mapped at run-time, or even to create a new one. The replication of the basic element is carried out by means of dynamic reconfiguration. Additionally, an approach to communicate single reconfigurable elements of the array, which does not require the use of specific macro structures, is provided. The proposed communication approach permits to provide generality to the systolic array, to gain flexibility and also reduces the run-time reconfigurable systems implementation area overhead.

The rest of the paper is organized as follows. In section 2, the related work is described, highlighting the main differences with the proposed solution, which is presented in detail in section 3. Section 4, provides implementation results and a use case of the proposed architecture and finally, conclusions can be found in section 5.

2 Related Work

In this section, a review of run-time scalable cores based on systolic arrays is included. Some representative related works in this specific topic have been selected

and characterized based on two main criterions. The first criterion is related to the array implementation and its floorplaning on the dynamically reconfigurable system, which defines the overall system flexibility. In this aspect, one and two-dimensional solutions can be differentiated. The second criterion, which is related to the system generality and also influence in its flexibility, is the partial dynamic reconfiguration design flow used.

The scalable FIR filter introduced in [13] is an example of a one-dimensional architecture, which is scaled by means of the addition of independent modules. Each additional module increases the number of coefficients of the filter, adapting the filter response to the desired filtering mask in real-time. This is a good example of tradeoff between the filter response quality and resource occupation. In this work, the design flow was the Xilinx Modular Design, which restricted system flexibility significantly.

In contrast, the newer Early Access Partial Reconfiguration (EAPR) flow, also provided by Xilinx, has been selected in [14] and [8]. The first work proposes a Scalable two-dimensional DCT architecture, where the EAPR is used to: i) adapt the precision of the DCT coefficients and ii) to remove and/or add processing elements to achieve different types of zonal coding, from 1×1 up to 8×8. That work also allows the use of the area of the removed elements by other tasks, but in a restricted manner. The second work, that is also two-dimensional, introduces an interesting reconfigurable DCT where processing elements are modified in the FPGA at run-time according to the zigzag order. In that work, elements are added or removed depending on the desired compression quality.

Two main conclusions can be drawn from the analysis of the related work. First, all the proposals are focused on offering solutions to specific problems or applications. Apart from the ones described in this section, other examples of concrete purpose scalable works are the template matching reconfigurable architecture presented in [15], the FIR filter in [4] or the image filter in [7]. Second, the reconfigurable architecture implementations and the design flows do not permit to completely release the area that is not used by the core. Therefore, the use of this unused area is highly restricted and permits to load only a narrow set of cores or processing elements. On the contrary, this paper provides a general approach to create any scalable, two-dimensional and run-time reconfigurable systolic array architecture. Furthermore, the solution permits to load a broad type of processing elements and cores in the system, cores, which might belong to different applications.

The limitations that derive from the selected design flow, Modular Design or EAPR, will be further discussed in section 3 and section 4.

3 Scalable Systolic Array

Systolic arrays can be defined as pipelined arrays of processing elements that rhythmically compute and pass data through its structure. Differently from the related work, the approach adopted in this paper is the definition of a generic systolic array that is customized afterwards following a design flow described in subsection 3.2. The basic idea behind the generic systolic array is the definition of a unique fine or medium grain processing element that is replicated in two dimensions. This process is used for building new systolic arrays based scalable cores or for scaling up/down existing ones. This results not only in the scaling of the functionality and/or the area of the core, but also, due to the implementation solution proposed in subsection 3.1, permits to free the remaining portion of the reconfigurable fabric for loading other cores.

The general view of the scalable systolic array can be seen on Figure 1, where the specific region for allocating one or several scalable cores is shown.

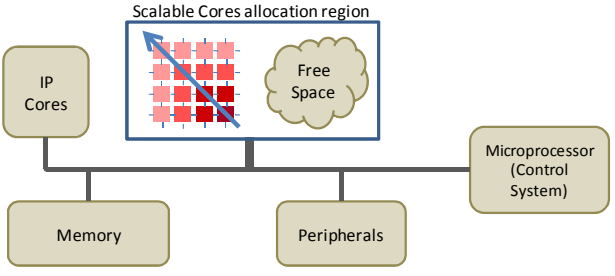


Fig. 1. System general view. The core is scaled by means of the addition of processing elements.

Similarly to the state of the art, the selected reconfiguration technique is partial dynamic reconfiguration. Due to this, the core scaling process does not disturb other cores that might be running in the system. Furthermore, the scalable core context switch is done at run-time, without stopping its functionality and, therefore pulling data out from the array is not required. In the following subsections, the proposed processing elements interconnections, as well as the design flow are shown.

3.1 Processing Elements Interconnection

An important advantage of systolic arrays is the fact that signals entering or leaving a processing element are mainly addressed to its closer neighbors. As a result, the existing interconnections are very regular. In this work, north, south, east and west ports have been considered for each element.

According to the Modular Design flow [16], different modules have to be connected through bus-macros that allow signals to cross reconfigurable module boundaries. These macros are instantiated in the top design, and in order to act as hard modules that do not change, they are constantly reloaded in the systems with partial reconfigurations. Due to the nature of these static bus-macros and the related design flow, an important tradeoff between the core scaling granularity and the area losses appears. The FPGA resources required for the bus-macro implementation cannot be used by the core itself and thus it is desirable to keep its number as low as possible in order to reduce area overheads. Meanwhile, the less bus-macros are included in the systems, the bigger the reconfiguration granularity, and this results in restricted system flexibility. From the systolic arrays design point of view, the use of macro structures forces the selection of processing elements with higher granularity restricting the systems flexibility and resulting in systolic arrays architectures that are tightly coupled to a specific application. Furthermore, the integration of bus-macros restricts the use of the free area, from the one reserved for loading scalable cores, by other cores as cores to be loaded have to strictly fit into the area defined by two consecutive macros.

The problems of resource usage have been partially solved in the latest version of the EAPR flow [17], where single slice bus-macros have been introduced. Single slice bus-macros are part of the reconfigurable area, instead of being part of the static base

design. As a result, fewer resources are consumed by the macros and also, resources are not used until a processing element is loaded in the reconfigurable area. However, this design flow does not permit to relocate a design along the defined reconfigurable area. Therefore, it is not suitable for the processing elements replication process, which is essential for achieving the flexibility and generality of the array proposed in this paper.

This paper provides a direct solution of the area/scaling granularity tradeoff by proposing a systolic array architecture that does not need bus-macros. This is achieved by exploiting the symmetry in the communications between processing elements in the systolic array. To achieve this, the north and west connections of an element have been designed using the same FPGA routing resources as the south and east connections. This symmetry has been also exploited to the design of the array processing elements that use the same routing resources to transmit the same signals. As a result, when a new element is added to the array in the reconfigurable area, its south routing wires will fit with the north wires of the element below, and their north wires with the south port of the element above. If the east-west connections keep the same conditions, this technique permits all the elements to be wire-compatible without the use of bus macros and the communications between the processing elements is guaranteed during dynamic reconfiguration. The symmetry of the north/south and east/west connections can be perceived from Figure 2, where three processing elements are included and the elements connections are highlighted.

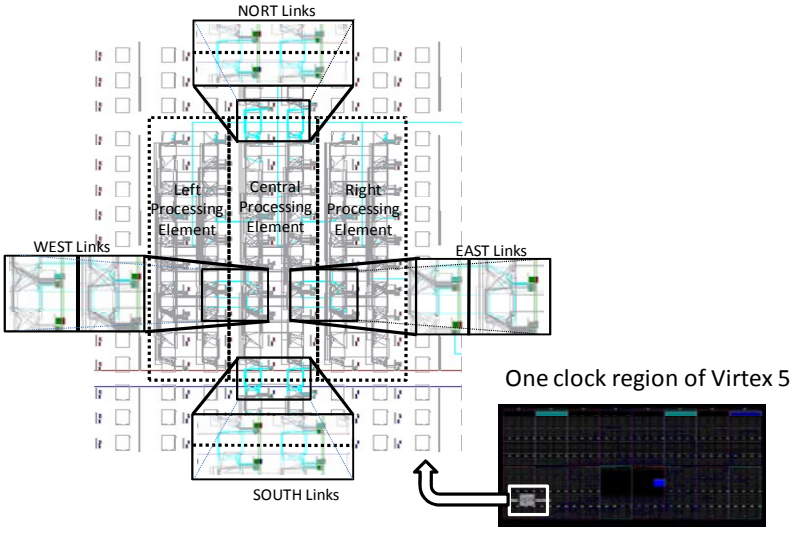


Fig. 2. Symmetry of the north/south and east/west connections of the processing element

Only when a block output has to be connected to several inputs of the adjacent element, an additional element, called anchor, is required. Anchor elements are implemented with look up tables and their main functionality is to distribute signals. The use of anchors permits to reduce the number of wires that cross the block boundaries. Anyway, anchor elements are considered part of each processing block.

In the use case and results section, some numerical results will be included to quantify the improvement of the non-bus macro approach.

3.2 Processing Elements Design Flow

In this subsection, a design flow to generate processing elements, customized to solve concrete problems and compatible with the generic systolic array approach is proposed. With this flow, specific scalable systolic solutions can be provided in a broad range of computational fields.

The first step of the flow is to define the systolic architecture that is required for the specific application. There exists extensive literature on this topic, including some automatic solutions [18]. Afterwards, the defined architecture is mapped into the reconfigurable system and divided in two parts: the systolic array itself and the control logic. The array is included in the reconfigurable region of the system and it is built and/or scaled with the method proposed in this paper. Differently, the control logic is included in the system static area, and is not affected by reconfigurations. Therefore, its design has to be valid for all the possible dimensions of the systolic array.

Once the architecture is mapped, the logic design of the basic elements of the array can be done. Each processing element is, indeed, a hard macro that can be directly designed with the FPGA Editor tool, or using a VHDL description that is tuned afterwards to define the shape of the element. After this, the processing element is instantiated five times in an ISE design with specific placement constraints, such that one of them is placed in the center and the other four blocks around it. Then, a step-by-step routing process is carried out in the FPGA Editor to define the connections between the central module and the adjacent ones. In this moment, the symmetry requirement of the communications, explained in the previous section, has to be accomplished. Finally, in order to generate the partial bitstream, the configuration frames that correspond to a single processing element are extracted from the full configuration with the five processing elements. This can be done using the bitgen Xilinx tool, or by means of a read-back operation of a processing element from the FPGA configuration memory.
By using this method, when the generated bitstream is replicated in the reconfigurable fabric, not only the processing elements logic content, but also, the communications among the blocks are automatically configured. This allows the creation and scaling of the systolic architecture by means of the replication, through relocation, of the unique single processing element. As a result, the total configuration data that has to be stored is that of a single processing element. The same relocation capacity is provided by the Modular Design flow, but with the expense of bus-macros. However, with the EAPR flow, a bitstream of the full systolic array has to be stored for each of the N possible scalability levels. This is because, on the contrary of the methodology proposed of this paper, the new macros that it uses do not permit to relocate modules and therefore module replication is not possible. The problem of the growth in the number of necessary bitstreams to manage the process of scaling with the EAPR flow can be seen in the experimental results of [14]. In the subsection 4.2, a quantitative estimation of these advantages will be provided.

Another important consequence of the relocation possibilities of the flow is that it permits to design processing elements independently from the system they will finally belong to, something that is not possible with the EAPR flow. This provides generality and permits: i) systolic cores to be configured from a library of processing elements or

even, ii) to self-replicate a processing element that is already configured in the device and does not belong to the library.

The relocation of processing elements is performed by means of specific software functions, combined with the ICAP (Internal Configuration Access port) drivers provided by Xilinx. Furthermore, small bit manipulation techniques can be used to tune specific parameters of the processing element in order to build heterogeneous systolic architectures where each element has different parameter values, like filter constants or transform coefficients.

4 Results and Use Case

In this section, a general evaluation of the proposed architecture is shown, including a theoretical comparison with other existing reconfigurable methodologies to deal with scalable architectures. In addition, a use case is provided to check the validity of the design flow in order to adapt the generic architecture to a particular problem. Finally, some numerical results will be shown related with the implementation of the use case. The design has been implemented using a Virtex-5 FX70T FPGA from Xilinx with a PPC440 embedded processor.

4.1 General Evaluation of the System

In this subsection, the proposed system will be evaluated by comparing its advantages in terms of system memory requirements and reconfiguration time with common solutions that are based on the Modular Design flow and the EAPR. All the provided results are restricted to completely homogeneous squared systolic arrays, but conclusions can be extended to heterogeneous architectures.

In the proposed solution, the size of the bitstream that has to be stored for the unique processing element is:

$$BitstreamSize = Frames\_per\_CLB \times Column\_per\_element \times \lceil Rows\_per\_element \rceil \quad (1)$$

being *Frames_per_CLB* the number of configuration frames of each CLB column of the FPGA and $\lceil n \rceil$ the integer division of n, in this case the number of rows of the basic element (Rows_per_element in (1)), by the number of CLB Rows per configuration Frame of the device. Typical values for this parameter are 16 CLBs for each Virtex-4 configuration frame and 20 CLBs for Virtex-5. A direct consequence of the relocation possibilities explained at the end of subsection 3.2, the total configuration data that has to be stored is that of a single processing element. The same result can be achieved with the Modular Design flow, but without considering the bus-macros overhead, which is high as it will be shown further in this section.

On the contrary, regarding the EAPR design flow, a bitstream of the full systolic array has to be stored for each of the N possible scalability levels. The size of each bitstream can be calculated with:

$$BitstreamSize = Frames\_per\_CLB \times N \times Column\_per\_element \times \lceil Rows\_per\_element \times N \rceil \quad (2)$$

The total amount of configuration data that has to be stored in the system with EAPR design flow to allow the different levels of scalability is:

$$TotalConfigurationData = N \times BitstreamSize = N \times Frames_{per_{CLB}} \times N \times Column\_per\_element \times \lceil Rows\_per\_element \times N \rceil \quad (3)$$

Table 1 shows the improvements with respect to the total amount of necessary configuration data that derive from the proposed solution, comparing with the EAPR and the Modular Design flow. As a reference, improvements compared with a static, non scalable design of the maximum N×N size are included. As it can be seen, the exact value of this comparison depends on the relative size of the processing element respect to the size of the reconfiguration frame. However, the maximum bound is provided to give a general idea of the achieved advantages.

Table 1. Comparison of the total amount of configuration bits of each technique

Flow	Relation with the proposed flow	Maximum Bound
Static Design	$\frac{N \times \lceil Rows\_per\_element \times N \rceil}{\lceil Rows\_per\_element \rceil}$	N^2
Modular Design	1	1
EAPR	$\frac{N \times N \times \lceil Rows\_per\_element \times N \rceil}{\lceil Rows\_per\_element \rceil}$	N^3

Regarding the time overhead for the scaling process, the use of the EAPR and the modular design flows will be compared with the proposal of this paper. In the present approach, the necessary reconfiguration time to scale the systolic array from (N-1) × (N-1) to N×N dimensions is the time to reconfigure the 2N-1 new elements. That is:

$$ReconfigurationTime = (2N-1) \times ReconfigurationTime_{per_{CLB}} \times Column_{per_{element}} \times \lceil Rows\_per\_element \rceil \quad (4)$$

The same result can be obtained with the modular design flow, again not considering the area overflow of bus-macros. However, with the EAPR, it is necessary to reconfigure the full core ($N_{max} \times N_{max}$ dimensions) in order to perform the process of scaling. The time overhead is:

$$ReconfigurationTime = ReconfigurationTime_{per_{CLB}} \times N_{max} \times Column_{per_{element}} \times \lceil Rows\_per\_element \times N_{max} \rceil \quad (5)$$

This parameter has no meaning in the case of the non scalable systolic design. The comparison with the proposed method results is shown in Table 2.

Table2. Comparison of the reconfiguration time overheads respect to the proposed method

Flow	Relation with the proposed flow	Maximum Bound
Modular Design	1	1
EAPR	$\frac{N_{max} \times \lceil Rows\_per\_element \times N_{max} \rceil}{(2 \times N - 1) \times \lceil Rows\_per\_element \rceil}$	$\frac{N \times N}{(2 \times N - 1)}$

4.2 Use Case Design

An image filter that performs window–based operations using a reconfigurable mask has been developed as an example of the proposed architecture as well as the design flow. This operator is the base of several image processing applications. Its principle is the application of an N×N pixels window to the image, operating the selected pixels according to a mask, and obtaining an output result. Usually, the output of the operation is the result in the position of the central point of the window. This window is slid across the whole image, generating all the points of the processed image. In [7], a very good review on this kind of operators is provided.

The operation developed in this paper is the 2D convolution, which is a special case of these windows-based operations. The output is a weighted average of the input pixels inside the window, using the mask like the weights. With the technique proposed in this paper, it is possible to create a scalable two-dimensional reconfigurable convolution, with the property of modifying its weights and its size in real time.

The systolic structure developed for the filter includes a static region with some control logic and memory elements to provide data in the correct order, and the systolic array itself, which is the scalable element. Following the provided design flow, the basic processing element of the systolic region of the filter has been designed using the FPGA Editor tool. Afterwards, the symmetric connections have been created and a partial bitstream for the element has been generated by reading back the corresponding portion of the FPGA.

Finally, to communicate the core with the static region and to bypass the columns of the FPGA whose content are not CLBs, as a first approach, static bus-macros have been used. Designed bus-macros have to fulfill also with the symmetry requirement for

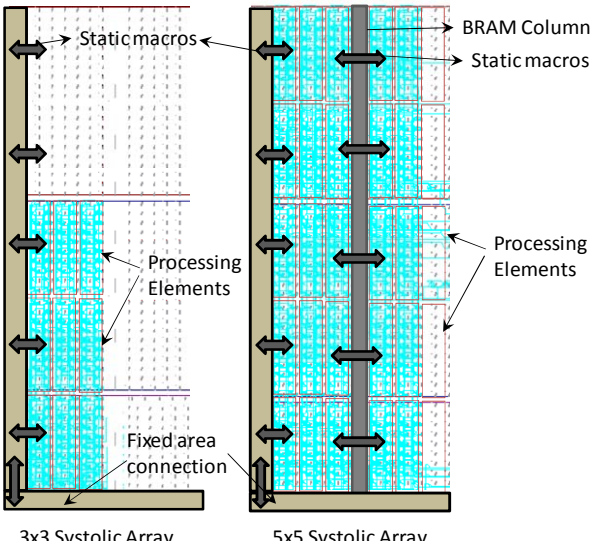


Fig. 3. FPGA Editor layout of the core scaled from dimensions 3×3 to 5×5 whith highlighted static macros

the communications among the processing elements. This guarantees that the bordering processing elements fit with the static bus-macros, in the same way they fit with the other processing elements. The main drawback of this approach is that it renders DSPs and BRAMs columns. In addition, the size of each core is limited to the number of columns between each two non CLB columns. However this problem, that is common in runtime reconfigurable system, is minor in the proposed in this paper architecture, since it focuses on small grain processing elements.

The system has been tested in terms of the scaling process by loading subsequently several processing elements in the systems using partial reconfiguration. As a result the systolic architecture and the no-macro based approach have been successfully validated.

Finally, in the Figure 3, layout captures of the FPGA Editor are provided, in order to show the result of the process of scaling the systolic architecture from dimensions 3 × 3 to 5 × 5. The selected FPGA layout allows a maximum size of 7 × 7 elements.

4.3 Use Case Results

While in the previous sections, some quantitative advantages of the proposed architecture have been underlined, in this subsection, some implementation results obtained from the use case will be provided, in order to prove the mentioned advantages in this particular design.

The first comparison is done in terms of logic consumption, comparing the proposed connections among elements without bus-macros, with respect to an implementation using Xilinx bus-macros. As it has been already mentioned, the basic processing element has four connections: North and south connections are 10 bits wide, while east and west connections are 16 bits wide. Since Xilinx dual-slice bus-macro allows an 8-bit communication between two reconfigurable regions, two bus-macros per interface would be required to allow all necessary interconnections. Consequently, the overhead of including these bus-macros are 8 CLBs for each processing element. Since each basic element occupies 20 CLBs, the area consumption of the elements with bus-macros would increase a 40%, compared with the solution proposed in this paper.

Additionally, it can be shown that the total amount of configuration bits is also reduced. As it is shown in Figure 3, each processing element occupies 2 columns and 10 rows in the reconfigurable device. Since each Virtex-5 CLB column requires 36 reconfiguration frames, 2×36 full frames have to be stored for the basic processing element. Differently, in a static and non-scalable solution, a bitstream of the area that corresponds to the biggest possible block should be stored. Regarding this use case, the 7 × 7 core occupies 14 columns of 70 CLBs each one. To configure each column, data corresponding to 4 clock regions are necessary. Since each column has 36 frames, 2016 frames have to be stored to configure the core. Moreover, in the EAPR design flow, considering the 7 scalability possibilities, the storage necessity is 7×2016 frames. A summary of this comparison is shown in Table 3. The final measured memory occupation of the basic processing element is 11 Kbytes.

Table 3. Amount of configuration frames with each technique for $N_{max} = 7$

Flow	Number of necessary frames	Relation with the proposed flow
Proposed	72	1
Static Design	2016	28
Modular Design	72	1
EAPR	14112	196

Finally, regarding the reconfiguration time, the EAPR design flow always consumes the necessary time to reconfigure the full 7 × 7 architecture, while both the proposed and the modular flows, only have to reconfigure the new elements. The FPGA Frames that have to be reconfigured to scale the systolic array from (N-1) × (N-1) to N×N dimensions can be seen in Table 4. A comparison between the two options is also shown, for different N values. The number of frames to reconfigure, in the case of the design flow proposed in this paper, depends on the value of N, but in the worst case, it is 2.15 times better than the achieved with the EAPR flow. The measured time to reconfigure each processing element is 0.34 ms.

Table 4. Comparison of the reconfiguration time of the EAPR flow respect to this proposal

N	Number of frames to reconfigure with the proposed flow	Number of frames to reconfigure with EAPR	Relation
1	72	2016	28
3	360	2016	5.6
5	648	2016	3.11
7	936	2016	2.15

5 Conclusions and Future Work

This paper describes the architecture of a generic systolic array with spatial scalability capability. The method is based on the replication and relocation of the single element using dynamic partial reconfiguration a basic element to generate an array which size could be adapted at runtime to the available area in the device, or to the application requirements of the executed task. The released area in the device can be freely used to load other cores. To allow the process of scaling, a communication structure that does not require the use of bus-macros is proposed, resulting in important area savings and improved FPGA occupation. Scalability of the solution is guaranteed in non homogeneous FPGAs (with embedded RAMs and other predefined blocks) by a symmetric bus-macro based feed-through structure, compatible with the scalable part of the architecture. In addition, a proprietary design flow is provided to adapt the generic architecture to the solution of specific problems. This allows flexibility enhancements with respect to the state of the art alternatives. In addition, with this approach, improvements are also achieved for both, the reconfiguration time overhead and the amount of configuration data. An image filter has been developed as a use case example.

Future work will include the development of a library of basic processing elements to provide scalable solutions in different data treatment fields, as well as to automate the decision of run-time scaling the core, according to changing application requirements or system conditions.

Acknowledgements

This work was supported by the Spanish Ministry of Science and Research under the project DR.SIMON (Dynamic Reconfigurability for Scalability in Multimedia Oriented Networks) with number TEC2008-06486-C02-01.

References

1. Zhang, X., Rabah, H., Weber, S.: Auto-adaptive reconfigurable architecture for scalable multimedia applications. In: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems, pp. 139–145 (2007)
2. Syed, S.B., Bayoumi, M.A.: A scalable architecture for discrete wavelet transform. In: Proceedings of the Conference on Computer Architectures for Machine Perception, CAMP 1995, September 18-20, pp. 44–50 (1995)
3. Huang, J., Lee, J., Ge, Y.: An array-based scalable architecture for DCT computations in video coding. In: Proceedings of the International Conference on Neural Networks and Signal Processing, June 7-11, pp. 451–455 (2008)
4. Meher, P.K., Chandrasekaran, S., Amira, A.: FPGA Realization of FIR Filters by Efficient and Flexible Systolization Using Distributed Arithmetic. IEEE Transactions on Signal Processing 56(7), 3009–3017 (2008)
5. Hwang, Y.-T., Chen, Y.-J., Chen, W.-D.: Scalable FFT kernel designs for MIMO OFDM based communication systems. In: TENCON 2007 - 2007 IEEE Region 10 Conference, October 30 -November 2, pp. 1–4 (2007)
6. Krishnaiah, G., Engin, N., Sawitzki, S.: Scalable Reconfigurable Channel Decoder Architecture for Future Wireless Handsets. In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2007, April 16-20, pp. 1–6 (2007)
7. Saldana, G., Arias-Estrada, M.: FPGA-based customizable systolic architecture for image processing applications. In: Proceedings of the International Conference on Reconfigurable Computing and FPGAs, ReConFig 2005, p. 3–8 (2005)
8. Huang, J., Lee, J.: A Self-Reconfigurable Platform for Scalable DCT Computation Using Compressed Partial Bitstreams and BlockRAM Prefetching. In: Proceedings of the IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2009, pp. 67–72 (May 2009)
9. Mehendale, M., Sherlekar, S.D., Venkatesh, G.: Area-delay tradeoff in distributed arithmetic based implementation of FIR filters. In: Proceedings of the Tenth International Conference on VLSI Design, January 1997, pp. 124–129 (1997)
10. Danne, K.: Distributed arithmetic FPGA design with online scalable size and performance. In: Proceedings of the 17th Symposium on Integrated Circuits and Systems Design, SBCCI 2004, pp. 135–140 (September 2004)
11. Selepis, I.N., Bekakos, M.P.: VHDL Code Automatic Generator for Systolic Arrays. In: Information and Communication Technologies, ICTTA 2006, vol. 2, pp. 2330–2334 (2006)
12. Junchao, Z., Weiliang, C., Shaojun, W.: Parameterized IP core design. In: Proceedings of the 4th International Conference on ASIC, pp. 744–747 (2001)
13. Oh, Y.-J., Lee, H., Lee, C.-H.: A reconfigurable FIR filter design using dynamic partial reconfiguration. In: Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2006, 4 pages, p. 4854 (2006)
14. Huang, J., Parris, M., Lee, J., DeMara, R.F.: Scalable FPGA Architecture for DCT Computation using Dynamic Partial Reconfiguration. In: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, USA (July 2008)
15. Gause, J., Cheung, P.Y.K., Luk, W.: Reconfigurable shape-adaptive template matching architectures. In: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 98–107 (2002)

16. Xilinx Corp., XAPP 290: Two flows for Partial Reconfiguration: Module Based or Difference Based (September 2004), http://www.xilinx.com
17. Xilinx Inc., Early Access Partial Reconfiguration User Guide. Xilinx Inc., San Jose (2006)
18. Ko, C.K., Wing, O.: Mapping strategy for automatic design of systolic arrays. In: Proceedings of the International Conference on Systolic Arrays, May 25-27, pp. 285–294 (1988)

Virtualization within a Parallel Array of Homogeneous Processing Units

Marc Stöttinger, Alexander Biedermann, and Sorin A. Huss

Technische Universität Darmstadt,
Department of Computer Science,
Integrated Circuits and Systems Lab,
64289 Darmstadt, Germany
{stoettinger,biedermann,huss}@iss.tu-darmstadt.de
http://www.iss.tu-darmstadt.de

Abstract. Our work aims at adapting the concept of virtualization, which is known from the context of operating systems, for concurrent hardware design. By contrast, the proposed concept applies virtualization not to processors or applications but to smaller processing units within a parallel array of homogeneous instances and individual tasks. Thereby, virtualization during runtime enables fault tolerance without the need for spare redundancy: The proposed architecture is able to switch seamlessly between parallelism for execution acceleration and redundancy for fault tolerance. In addition, faulty instances are completely decoupled from the system. This allows for an easy dynamic and partial reconfiguration. Using this concept, self-healing mechanisms can be implemented, as decoupled, faulty instances may be replaced by operational instances during reconfiguration. We present this hardware-based virtualization concept on the basis of a parallel array of multipliers used for ECC point-multiplications.

Keywords: virtualization, middleware, fault tolerance, partial reconfiguration, self-healing systems, parallel hardware systems.

1 Introduction

In order to execute a task, it has to be assigned to an instance of a resource. This mapping is called binding. Often, many tasks have to share the same instance. The imbalance between tasks and available resources requires scheduling, whereas resource conflicts have to be avoided. Furthermore, the contexts of tasks sharing the same instance may not mix up. One of the several mechanisms uses virtualization to accomplish this. Virtualization permits several tasks to use the same instance of a resource. Its fundamental aspect is that each task believes to be the sole user of an instance. This is done by inserting an abstraction layer between tasks and resources. As an advantageous side-effect, a task can be transparently executed on another instance of the same resource. Therefore, a task becomes independent from an explicit binding to an instance of a resource. Thus,

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 17–28, 2010.

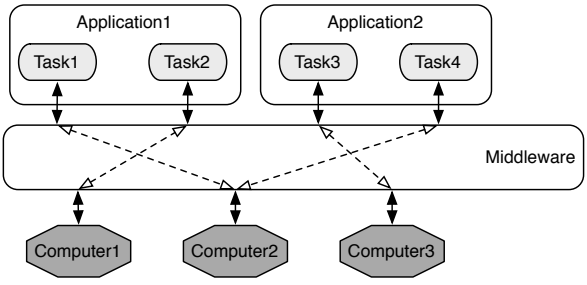


Fig. 1. Visualization of middleware in distributed computing

in case of a fault, another instance may be utilized to execute the task without any side-effects. As a result, virtualization is not only suited to manage resource binding, but also as a basis to implement fault tolerance.

Fault tolerance is an important aspect in the development of embedded systems. A frequently applied approach is to allocate additional, redundant components, which take over the tasks of defective components in case of a fault, as described in [6], but the drawback is the high resource consumption caused by redundancy. Furthermore, in normal operation, there is no benefit from these spare resources. Other mechanisms, additionally, use redundant components for fault detection [11]. Here, *Triple Modular Redundancy* (TMR) is used to detect variations in the output of one of three redundant components. The authors of [2] and [9] utilize reconfigurable spare areas on the chip, which do not have to be explicitly held ready for dedicated components. However, the approach mentioned in [9] needs two FPGAs, thus is resource intensive, too. To circumvent the high resource consumption caused by redundancy, the virtualization concept lends itself to the purpose of resource saving. Using virtualization to gain fault tolerance is already discussed in [14]. While this approach motivates virtualization by (sub-)networking, we focus on embedded, parallel hardware systems, which do not require intensive rerouting in case of a fault. We exploit the concept of virtualization for the development of a system, which dynamically combines the advantages of parallel execution and redundancy for fault tolerance without dedicated spare resources.

The paper is structured as follows: In section 2 we introduce the concept of a middleware approach as the basic platform for our virtualization technique of hardware resources. Then we describe our virtualization procedure and the related hardware architecture. Section 3 demonstrates this novel approach by means of an ECC point-multiplication, which uses the virtualization architecture described before. After an application example, we discuss advantages, limitations, and possible improvements in section 4. Finally, section 5 subsumes the achieved results and lists some advanced aspects, which will be addressed in future research work.

2 Methods

In this section we present the advocated layer between data path and control unit, respectively between the resources and the task, which is used in our virtualization approach. This layer can be viewed as a kind of middleware, a well known concept in the area of distributed computing [1]. Firstly, we explain this specific middleware concept. Secondly, we apply it to a hardware design using a bus system, then we will show how to exploit the proposed middleware-enhanced design to virtualize resource entities in a homogeneous multi-processor array.

2.1 Middleware Concept

A middleware structure, as applied in the area of distributed computing [12], is mainly a protocol aimed to control data transfer and task assignment between different applications and processing units, as depicted in figure 1. Therefore, this type of protocol controls the data flow as well as the control flow of the distributed computing elements in the network. The advantage of this intermediate layer is an abstraction of both the control and the data flow within the processed application.

Middleware on Top of a Bus System. Traditionally, bus systems only perform data transfers without manipulating data or control flow. For instance, data and control flow are managed by a central control unit. A middleware layer, however, not only performs data transfers, but also has the ability to dynamically change the data and control flow of an application. Therefore, we extend a common bus structure to provide a middleware functionality as follows:

A bus system-based middleware unit consists of a bus system with several data transmission buses, one arbiter unit, and one control bus. The arbiter unit controls the data transfer between the units of the architecture, e. g., between distinct processing units and the memory, via the underlying data bus structure. It also manages the molding processes of each processing unit via the control

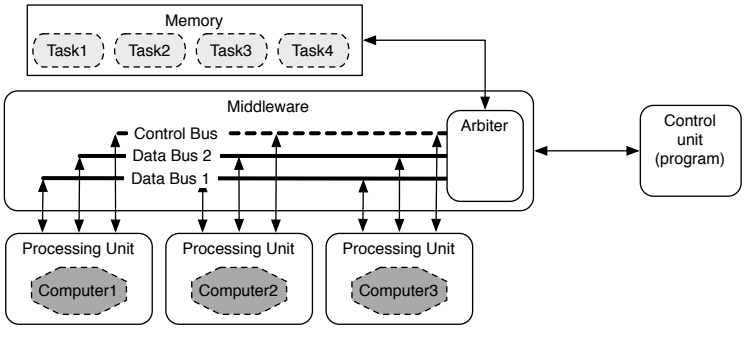


Fig. 2. Middleware functionality mapped to hardware

Algorithm 1. Virtualization by middleware concept

```
Require: (current schedule)
  n, m := 0
  S_cur ← current schedule
  t ← getNumOfAllTasksOfCurrentSchedule(S_cur)
  T[t] ← getAllTasksOfCurrentSchedule(S_cur)
  for i = 0 to t do
    if (!(checkAvailabilityOfResourceOfTask(T[i]))) then
      R_f[n] ← getResourceOfTask(T[i])
      T_f[n] ← (T[i])
      n++
    else
      R_h[m] ← getResourceOfTask(T[i])
      T_h[m] ← (T[i])
      m++
    end if
  end for
  if (isEmpty(R_f[ ])) then
    ExecuteTasksOfSchedule(S_cur)
    return
  else
    S_aux1 ← RescheduleTasks(S_cur, T_h[m])
    S_aux2 ← makeScheduleOf(R_h[m], T_f[n])
    while (ExecuteTasksOfSchedule(S_aux2)) do
      stallOtherTasks()
    end while
    ExecuteTasksOfSchedule(S_aux1)
    return
  end if
```

bus. The arbiter starts the operation of a processing unit, when all task specific data are available at this unit. The bidirectional control bus also informs the arbiter unit if a processing unit has finished its task so that the calculated result can either be transferred to the global memory or to another processing unit to prepare its next task execution. The central control unit no longer has the ability to directly change data and control flow, as it only communicates with the middleware unit. An example of the scheme of such a bus system-based middleware is depicted in figure 2. As a result, the middleware layer allows to dynamically bind tasks to resources, a property being an essential prerequisite for the proposed virtualization concept.

2.2 Virtualization Concept

By the introduction of task management using the mentioned middleware approach, we are able to virtualize the hardware instances of this architecture. We first define several terms in this context: A *virtualizable unit* is an unit, usually an instance of a resource, whose assigned task may be executed on another unit.

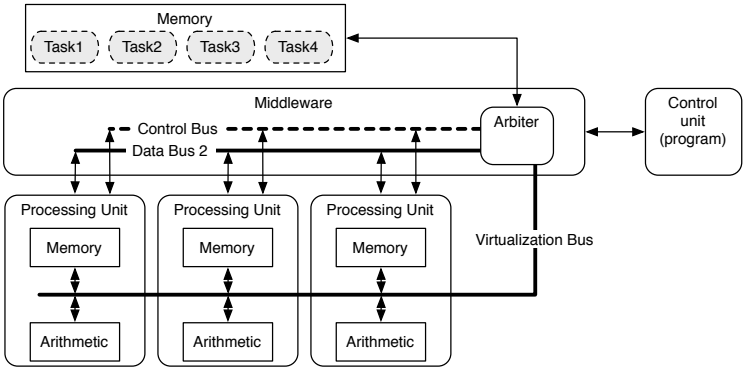


Fig. 3. Middleware architecture featuring distributed memory

A *virtualizing unit* is an unit, usually an instance of a resource, that took over the execution of a task initially assigned to another unit. Finally, a *virtualized unit* is an unit, usually an instance of a resource, whose assigned task is currently being executed on another unit. Thus, the virtualization concept features the following properties:

- *Dynamic resource allocation during runtime*
  Detecting available entities of the needed resource to execute a task.
- *Dynamic resource binding during runtime*
  Mapping the virtualized tasks to an available appropriate resource entity.
- *Dynamic rescheduling of the control flow*
  Adapting the control flow of the application during and after the virtualization phase without changing the original program of the application.

To realize the above stated functionalities in a bus system-based middleware approach, we extended the basic arbiter functionality within the middleware unit by algorithm 1. In the beginning of the algorithm, all tasks T of the current schedule S_{cur}, which still have to be processed at the same point in time, are identified. After getting all unprocessed tasks, their binding is checked. These tasks are then sorted into two groups. The members of the first group are all tasks T_h that are bound to correctly working resource instances R_h. In the second group all the tasks T_f with a binding to defective instances R_f are listed.

If the group R_f is empty, the actual schedule S_{cur}, most likely the initial schedule, is proceeded. Otherwise, a new schedule of both the tasks T_h and T_f is needed. Firstly, the old schedule S_{cur} is reorganized with the tasks T_h to the schedule S_{aux1}. Secondly, a new schedule S_{aux2} is composed based on the tasks T_f and the resources R_h.

When both schedules are set up, they are executed sequentially to consider possible dependencies between the processes of S_{aux1} and S_{aux2}. The schedule S_{aux2} is first processed while stalling all the other processes. After the schedule S_{aux2} is completed, schedule S_{aux1} is executed and the algorithm starts all over again.

Virtualization by the Middleware. To execute a virtualization based on the above-mentioned procedure, we use one of the multiple data buses, as depicted in figure 2. This bus is utilized to transfer the task-dependent data from the original, virtualized instance of a resource to the virtualizing instance. Furthermore, the control bus is used by the arbiter unit for the virtualization process. It manages the new schedule and binding of the virtualizing resource without altering the control flow of the components, which are not involved in the virtualization.

The control flow in this architecture is realized by two units, namely a controller and a middleware unit. The instructions of the application are routed into the middleware unit before the instructions and control signals are forwarded to the respective components in the data path. To program this parallel processor architecture, we used an instruction set of *very large instruction words* (VLIW): All parallel executable instructions of all tasks are concatenated. Additionally, the virtualizable processing units can be addressed directly by the VLIW-instruction set.

Memory Architecture and Consistency. Also the memory consistency has to be considered. One of the costs of the middleware architecture in figure 2 is the context switching of both virtualized and virtualizing processing unit. Side-effects may appear during or after virtualization of a processing unit because the access order of the data may change with respect to the original control flow of the application. However, these side-effects of the memory due to the switching procedure have to be avoided in any case. Therefore, methods have to be applied to assure data consistency and to prevent context switching as depicted in figure 4.

Each processing unit has a small local scratch pad memory for caching the task-dependent data. Focusing on the virtualization of functionality, we moved one data bus of the middleware architecture in figure 2 now to be located between the local memory of the processing units and their arithmetic part of the data path. With such an architecture, as depicted in figure 3, no explicit context switching is needed during a task-transparent virtualization phase.

Based on the distributed memory architecture, the constraints for the data integrity are simpler compared to multi-threading techniques based on a single

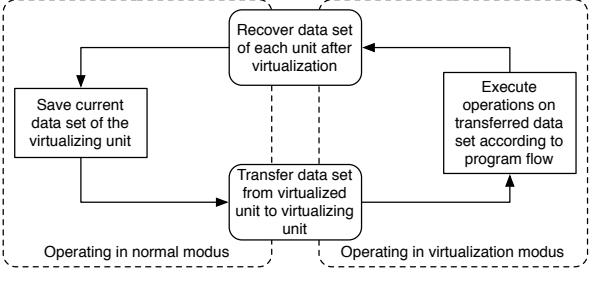


Fig. 4. Process steps to assure data consistency

memory block: Processes connected by the virtualization bus and exclusively using their own local working memory highly reduce the need for control by avoiding context switching for the virtualization phase.

3 Application Example: ECC Point-Doubling Operation

A suitable application for these concepts may be found in car-2-car communication: A fault tolerant and fast signing procedure for prioritized rescue vehicles is essential to verify the rights to influence the traffic flow in case of emergency [13]. Due to the complexity of this scenario we abstracted of this application example to a smaller and essential component - a part of signing by *Elliptic Curve Cryptography* (ECC).

ECC is one of the most established asymmetrical cryptography schemes. It is based on the difficulty of calculating the discrete logarithm on elliptic curves [3]. An elliptic curve is an additive, abelian group with a defined addition operation and a neutral element called point of infinity.

The elliptic curve crypto system is composed from a mathematical strictly layered architecture, as depicted in figure 5. The top layer exhibits the operations of the *Elliptic Curve Arithmetic*, in general point-addition, point-doubling, and point-multiplication. These operations originate from the arithmetic operations in the underlying *Finite Field Arithmetic*. Each Elliptic Curve Arithmetic operation requires the execution of multiple lower-level operations. Thus, a cryptographic operation, e. g., a digital signature, requires just two point-multiplications on the elliptic curve, but hundreds of point-addition and point-doubling operations, i. e. correspondingly many operations on the Finite Field Arithmetic.

For comparison purposes, we implemented and synthesized two elliptic curve arithmetic-coprocessors for the point-double operation on the Virtex-II Pro platform. One coprocessor design is based on the architecture, as depicted in figure 6, with three virtualizable multiplication units for finite field multiplications and a middleware-extended control unit. The middleware concept is realized by an additional control unit and an additional bus between the memory components and the multiplier, which is accessible by each component via multiplexer and tri-state buffer. The *Virtualization CBus* manages the control signals between the processing units and the middleware component. The second design features three non-virtualizable finite field multiplication units and a normal control unit. This design basically originates from [7].

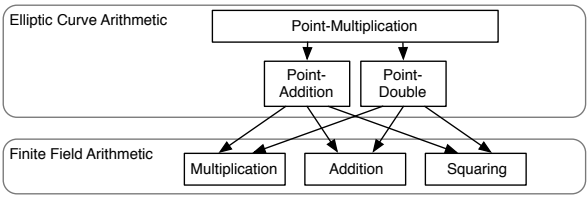


Fig. 5. Layered mathematical structure of elliptic curve arithmetic

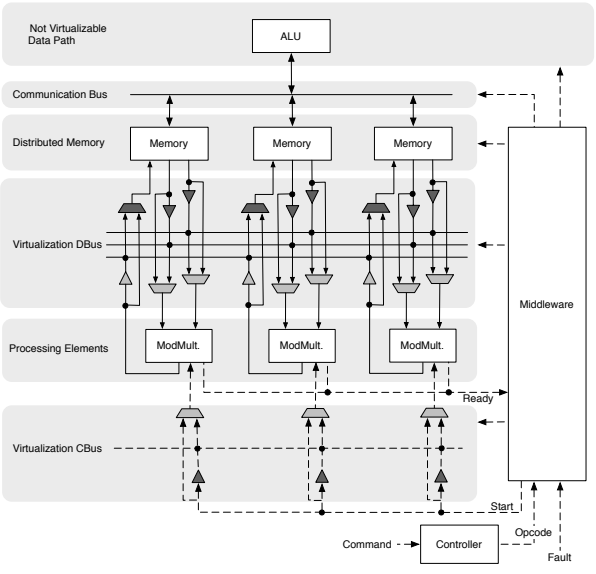


Fig. 6. Scheme of the virtualized ECC architecture

We selected a manual fault-injection as an application scenario to analyze the virtualization concept of our middleware approach. On both architectures we simulate a *point-doubling* operation with no fault, with one fault, and with two faults, respectively, on a processing unit during runtime. Figure 7 depicts the data flow graph of the simulated application.

On the left side of the figure the original data flow of a point doubling algorithm in the López-Daham projective coordinates [8] is displayed. The different background colors of each task node highlight different processing units utilized to execute the task. On the other side, the rescheduled data flow is schemed for virtualizing the most left multiplication task running task at t_2 of the original schedule. The virtualization is forced by a manually injected, static soft error of the multiplication unit. During the virtualization of the multiplication task at point in time t_2, the multiplier resource symbolized by the white background color can be reconfigured without stalling the whole application. The non-virtualizable architecture has to start the application all over, if a fault is detected. Table 1 depicts the resource consumption of both designs, while figure 8 depicts runtime performance for the three described simulated cases.

The high resource consumption of slices is caused by the discrete and simultaneous architecture of the local memory units. In case of the non-virtualizable architecture, the structure of the local memories uses 52 % of the resources, while the local memories in the virtualizable architecture sum up to 44 %. Table 1 clearly depicts the higher resource amount of the virtualizable architecture. The additional controllable bus for the virtualization and the middleware unit for managing resource binding during the partial reconfiguration only needs

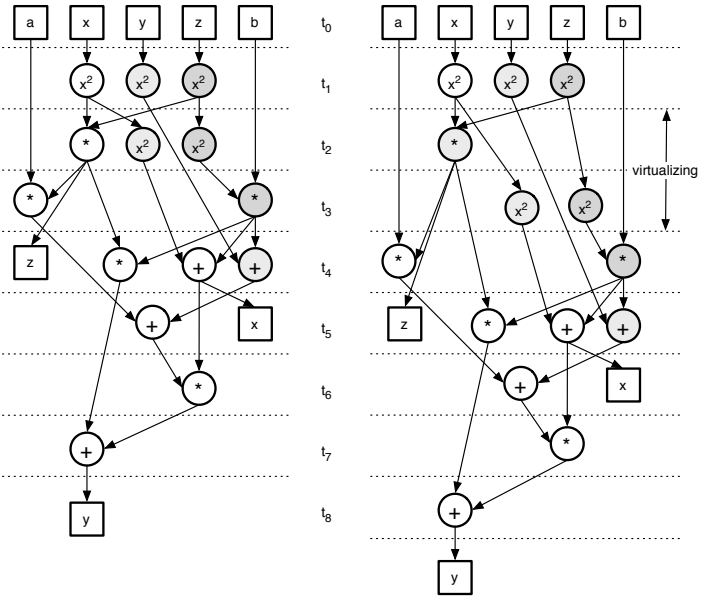


Fig. 7. Two different schedules for the same point-doubling operation

15 % of the slices of the virtualizable design and 15 % of its look-up tables. The additional amount of resources is a reasonable price to pay in order to achieve the ability of reconfiguring defective units during runtime. The additional logic of the bus structure increases the delay time of the combinatorial way, thus the maximum working frequency in our example of the virtualized architecture is decreased by 26 MHz to 94 MHz instead of 120 MHz of the non-virtualized.

The point of the fault-injection for the runtime simulation with one fault was placed before the second squaring section, which points to the time instant t_2 in the diagram given in figure 7. On the left hand side of figure 7 the data flow graph of a point doubling operation with no virtualization is given, while on the right hand side the data flow of the same point doubling operation with one virtualizing task is schemed. In the other fault simulation cases the faults were injected at the time points t_2 and t_4, respectively, of the schedule depicted in figure 7.

The process duration of point-doubling operation on both the virtualizable and the non-virtualizable architecture is the same, if no fault is injected, as

Table 1. Resource consumption of the architectures

Resources	virtualizable	non-virtualizable
Slices	10634	9030
FlipFlops	6294	6294
LUTs	20142	17051

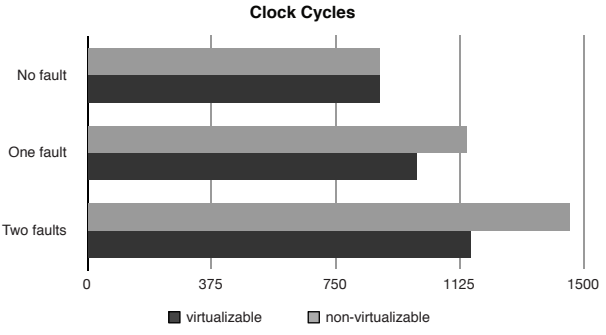


Fig. 8. Processing durations of point-doubling on both architectures

depicted in figure 8. In case of a fault, the virtualizing architecture needs one clock cycle more to start the execution of the virtualized task than starting the same process on the non-virtualizable architecture on the same processing unit type. In case of a fault-injection the virtualizable architecture is faster than the architecture without virtualizable units. This result is caused by the restart behavior of the non-fault-tolerant architecture after a fault detection. The non-virtualizable architecture will be faster, if the fault is injected in the first processing phase: The processing units, which are not affected by virtualization, are at least stalled as long as the execution of the virtualized task lasts, whereas the non-virtualizable architecture is immediately restarted after detecting a fault.

4 Discussion

The example scenario in section 3 shows that this concept of virtualization of a bus system-based middleware is well-suited to provide fault tolerance in a homogeneous parallel working processing unit array without additional redundancy techniques. This virtualizable architecture has the advantage of correcting defective instances of resources without changing the control flow of the application or restarting the entire application at all. For fault-tolerant usage, we premise already existing, application-specific fault recognition structures.

Especially homogeneous multi-processor architectures take advantage from using this virtualization concept, as they may switch between parallelism either for execution acceleration or for dynamically providing redundancy for fault tolerance. No additional redundant processing units are needed to specifically implement fault tolerance.

Moreover, the general virtualization approach provides new methods and concepts in the area of self-healing systems based on reconfigurable hardware. Self-healing systems have the ability not only to tolerate faults, but to recover from them. As described in section 2.2, the application schedule in the presented concept does not have to include a non-interrupting reconfiguration schedule between the different task executions on a resource instance, like in [4]. Otherwise, no additional generic neighbor cell structures as in [10] are needed to assure

a self-healing architecture in case of permanent errors, as further described in [15]. Based on the decoupled control flow of the application and the virtualization procedure for the data path as proposed in this paper, the virtualized components can be reconfigured during runtime without any context switching at all. This offers the possibility to reconfigure processing units on task level rather than on application level. Therefore, this a much finer granularity with respect to partial reconfiguration.

The presented architecture relies on a distributed memory architecture, aimed to assure memory consistency during virtualization. Distributed memory implies distributed code for each task or program on a dedicated memory. Some programming paradigms handle distributed memories, as for example for the programming of the IBM Cell architecture [5], which is somewhat similar to our approach with respect to homogeneous, virtualizable processing units. At the moment, the user has to write individual programs for each memory unit and to combine them into complex VLIW-code as already detailed before. Furthermore, no explicit resource binding is actually necessary, since the middleware abstracts tasks from resources. However, the user has to declare an initial binding. For the two given reasons, it would be an advantage, if a further layer would be introduced: In this layer programs could be analyzed regarding the possible level of parallelization, accordingly many resources could be allocated, and the initial binding could be automated.

During virtualization the application example presented in section 3 stalls. Thus, extensive rescheduling is avoided at the expense of throughput. An extension of the presented concept may reschedule the program flow during virtualization in such a manner that a stall of units, which are not affected by the virtualization, is avoided. By combining and implementing these upgrades an easy to use, fault-tolerant framework for parallel processing will result.

5 Conclusion

In this paper we introduced a virtualization concept for hardware architectures on top of a bus system-based middleware approach that is very suitable for homogeneous multi-processor architectures. According to the transparent task management of our middleware approach, the gap of the resource utilization between redundancy and parallelism is narrowed. Virtualization allows to switch between these two dynamically. Furthermore, virtualization completely decouples any virtualized components. This allows for easy partial and dynamic reconfiguration. As virtualization in general degrades throughput of the system to some extent, defective and thus decoupled components may be reconfigured, i. e., repaired. As a result, our concept not only offers a framework for easy partial and dynamic reconfiguration, but also provides a basis to implement systems with self-healing abilities. In future work, we will apply a high-level programming paradigm on the distributed memory approach. Some process steps will thus be automated, such as generating initial control flow and binding. Optimization of the reschedule algorithm in case of a fault appearance concludes the envisaged future work.

Acknowledgment

This work was supported by DFG grant HU 620/12, as part of the Priority Program 1148, in cooperation with CASED (http://www.cased.de).

References

1. Bernstein, P.A.: Middleware: a model for distributed system services. Commun. ACM 39(2), 86–98 (1996)
2. Bobda, C., Majer, M., Ahmadinia, A., Haller, T., Linarth, A., Teich, J.: The Erlangen Slot Machine: A highly flexible FPGA-based reconfigurable platform. In: 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2005, pp. 319–320 (2005)
3. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus (2003)
4. Huang, M., Simmler, H., Serres, O., El-Ghazawi, T.A.: Rdms: A hardware task scheduling algorithm for reconfigurable computing. In: IPDPS. IEEE, Los Alamitos (2009)
5. International Business Machines Corporation: Cell Broadband Engine Architecture
6. Lala, P., Kumar, B.: An architecture for self-healing digital systems. Journal of Electronic Testing 19(5), 523–535 (2003)
7. Laue, R., Huss, S.A.: A novel memory architecture for elliptic curve cryptography with parallel modular multipliers. In: IEEE International Conference on Field Programmable Technology, Bangkok, Thailand, pp. 149–156 (2006)
8. López, J., Dahab, R.: Improved algorithms for elliptic curve arithmetic in gf(2n). In: Tavares, S., Meijer, H. (eds.) SAC 1998. LNCS, vol. 1556, pp. 201–212. Springer, Heidelberg (1999)
9. Mitra, S., Huang, W., Saxena, N., Yu, S., McCluskey, E.: Reconfigurable architecture for autonomous self-repair. In: IEEE DESIGN & TEST, pp. 228–240 (2004)
10. Nollet, V., Marescaux, T., Avasare, P., Mignolet, J.Y.: Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In: DATE 2005: Proceedings of the conference on Design, Automation and Test in Europe, pp. 234–239. IEEE Computer Society, Washington (2005)
11. Paulsson, K., Hubner, M., Becker, J.: Strategies to on-line failure recovery in self-adaptive systems based on dynamic and partial reconfiguration. In: AHS 2006: Proceedings of the first NASA/ESA conference on Adaptive Hardware and Systems, pp. 288–291. IEEE Computer Society, Washington (2006)
12. Tanenbaum, A.S., Steen, M.V.: Distributed Systems: Principles and Paradigms. Prentice Hall PTR, Upper Saddle River (2001)
13. Wojtowicz, T., Bogucki, P., Stochel, M., Wisniewski, L., Sztando, R., Kalenczuk, M., Swierczynski, J.: Adaptive traffic control for emergency vehicals. Tech. rep.
14. Yeh, C.: The robust middleware approach for transparent and systematic fault tolerance in parallel and distributed systems. In: Proceedings of International Conference on Parallel Processing, pp. 61–68 (2003)
15. Zhang, X., Dragffy, G., Pipe, A.G., Gunton, N., Zhu, Q.M.: A reconfigurable self-healing embryonic cell architecture. In: Proceedings of ERSA 2003: International Conference on Engineering of Reconfigurable Systems and Algorithms, pp. 134–140 (2003)

Feasibility Study of a Self-healing Hardware Platform

Michael Reibel Boesen, Pascal Schleuniger, and Jan Madsen

DTU Informatics, Technical University of Denmark, Richard Petersens Plads
Bldg. 322
{mrb,pass,jan}@imm.dtu.dk

Abstract. The increasing demand for robust and reliable systems is pushing the development of adaptable and reconfigurable hardware platforms. However, the robustness and reliability comes at an overhead in terms of longer execution times and increased areas of the platforms. In this paper we are interested in exploring the performance overhead of executing applications on a specific platform, eDNA [1], which is a bio-inspired reconfigurable hardware platform with self-healing capabilities. eDNA is a scalable and coarse grained architecture consisting of an array of interconnected cells aimed at defining a new type of FPGAs. We study the performance of an emulation of the platform implemented as a PicoBlaze-based multi-core architecture with up to 49 cores realised on a Xilinx Virtex-II Pro FPGA. We show a performance overhead compared to a single processor solution, which is in the range of 2x - 18x, depending on the size and complexity of the application. Although this is very large, most of it can be attributed the limitations of the PicoBlaze-based prototype implementation. More important, the overhead after self-healing, where up to 30-75% faulty cells are replaced by spare cells on the platform, is less than 20%.

1 Introduction

Reliability and consequently, fault-tolerance are becoming key factors for a high fraction of embedded applications, because they are becoming more and more pervasive. When you press the brake in your car on an icy day you hope that the embedded system implementing the ABS has not succumbed to the cold weather. The classical way of introducing fault-tolerance in hardware is to have three redundant copies of the hardware plus a voter to detect which output is the correct one. But what will you do if one of the redundant copies fails permanently or what if the voter itself is faulty? Then you cannot decide which of the output is the correct one and the system needs replacement. Certainly, if the application in question is located alone on Mars you have a problem, but even if the application is more down-to-earth for instance a chip in a traffic light it certainly would be nice not having to pay for a repair guy to go and fix it. Therefore proposals for reconfigurable self-healing hardware platforms have emerged in the last decade. The eDNA platform as presented in [1] is our proposal. In this work we have

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 29–41, 2010.

implemented the first prototype of eDNA - called eDNA 1.0 and we will show that eDNA is indeed feasible for industrial applications even though it is still on an early stage and limited by several factors.

1.1 Related Work

eDNA is inspired from the way eukaryotic cells (cells in multicellular organisms) read and interpret the DNA in order to build and maintain an organism. Other bio-inspired self-healing hardware architectures are emerging from different groups such as [2,3,4,5]. [2,4] present a fine-grained reconfigurable self-healing architecture aimed at an FPGA implementation and is inspired by eukaryotic cells, just like the eDNA platform. They demonstrate the self-healing capabilities of their system with several examples. However, the architecture suffers from being too fine-grained, which makes the overhead when programming larger applications big. A similar approach [5] is inspired by prokaryotic cells (cells in unicellular organism), however, the paper is conceptual and doesn't describe specific implementation details. The eDNA platform distinguishes itself from these by having a higher logical granularity and the fact that it is aimed at an ASIC implementation and not an FPGA implementation.

A part from being bio-inspired eDNA utilise dynamic reconfigurability in order to perform self-healing. With the increasing demand for adaptive hardware platforms there have been a lot of research effort in dynamical reconfigurable FPGAs and MPSoCs, similar in type to that of eDNA. In order to decrease the cost of reconfigurability work such as [6,7,8] investigate the advantages of coarse-grained dynamical reconfigurable FPGAs and MPSoCs. [6] propose a hardware platform consisting of two Xilinx MicroBlaze[9] cores as well as hardware co-processors to handle encryption and compression of package payload for network applications. They show that the use of dynamical reconfigurable co-processors gains a speed-up of 12-35% compared to a static version. [7] propose an architecture called QUKU which is a mix of an FPGA and a Coarse-Grained Reconfigurable Array (CGRA), by overlaying an FPGA with a network of PEs. Furthermore the architecture has a MicroBlaze soft-core which controls the reconfiguration of the architecture. They show that the QUKU architecture achieves a very low configuration time compared to the execution time of two edge-detection algorithms. Finally, [8] has explored how to tune a CGRA to its application domain by doing a design space exploration on how each application intends to use the resources on the CGRA.

The contribution of this paper is a description of the first prototype of the eDNA platform as well as an illumination of the performance to be expected from it.

The next section will introduce the eDNA design methodology, section 3 will describe the implemented hardware prototype, section 4 will present and discuss the performance results obtained and finally, section 5 will present the conclusion.

2 eDNA Design Methodology

The purpose of this section is to introduce the features of our proposed system in order to motivate why we study this type of hardware platform. eDNA is an abbreviation for *electronic DNA* and the eDNA *platform* is a bio-inspired reconfigurable hardware platform utilising the eDNA to provide two features: Self-organisation and self-healing. The self-organisation is the process where the eDNA hardware platform configures itself to execute the application provided by the user. The self-healing is the process where faults are detected and eliminated. The eDNA platform consists of an array of homogenous eCells (abbrev. for *electronic cell*) connected via a Network-on-Chip (NoC). In the following the self-organisation and self-healing will be explained.

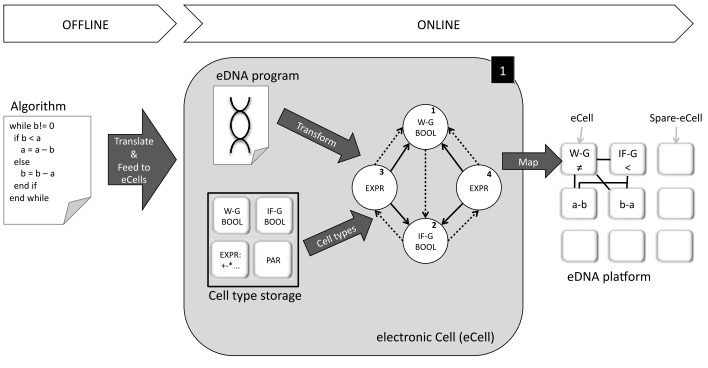


Fig. 1. Overview of the eDNA design flow

2.1 Self-Organisation

Figure 1 summarises how the self-organisation works. The eDNA design flow starts from a behavioral specification of the application, expressed as a software implementation. This software implementation is translated into the eDNA language as outlined in figure 1. The eDNA language can be seen in Backus-Nour Form notation below.

```
dna        ::= <statement>* |  <parallel>*
statement  ::= <assignment> | <while> | <if> | return <var/c> |<parallel>
parallel   ::= parallel <statement>* endparallel
assignment ::= <var/c> = <exp>
while      ::= while <bexp> do <statement>* endwhile
if         ::= if <bexp> then <statement>* else <statement>* endif
exp        ::= <var/c> [<op> <exp>]*
bexp       ::= <var/c> [<bop> <bexp>]*
op         ::= AND | OR  | + | - | ...
bop        ::= AND | OR | < | <= | == | != | ...
var/c      ::= Letters{A-Z}* | <const> | RAM <var/c>
const      ::= 0<const>* | 1<const>*
```

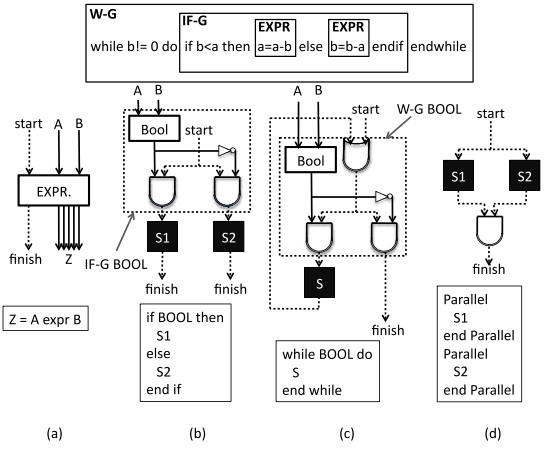


Fig. 2. (a)-(c) Modified SW→HW blocks inspired by Ian Page [10]. (d) The parallel block introduced with the eDNA, which is not a part of [10].

A converter program then encodes the program and feeds it to all of the eCells. This is the only part which is performed offline, the rest of the process is performed by the eCells online. Once an eCell has received the eDNA program it starts *transforming* the program into a task-graph (figure 1). The transformation from algorithmic description to hardware is done following a translational model inspired by a paper by Ian Page [10]. Ian Page proposed a way of translating software code directly to hardware. The translational model of the constructs of the eDNA language can be seen in figure 2. Note that figure 2(b) and (c) abbreviates part of the hardware as `W-G BOOL` and `IF-G BOOL`, this will be needed later on. Each block on figure 2 displays a `start/finish` signal. The `start/finish` signal provides a sequencing mechanism to the system. This means that the order of operation of the eDNA code is maintained by this signal. For instance as seen on figure 2(b) and (c) it is clear that the `if` and `while` block respectively is inactive as long as the `start` signal coming from the preceding block is 0. At the top of figure 2 the example code from figure 1 is seen. The sections of the code which corresponds to the individual blocks of figure 2 is clearly marked. Note that the communication paths in the task graph (figure 1) is derived from the communication between the blocks in figure 2. The dashed lines in the task-graph in figure 1 corresponds to the start-signal lines of figure 2 and the solid lines corresponds to data coming from the `EXPR` block of figure 2(a). In order to decide which eCell should perform which part of the task graph, the concept of *cell types* will be introduced. The unprogrammed eCell correspond to a biological stem cell in the sense that stem cells can differentiate into multiple cell types. The unprogrammed eCells can differentiate into 4 different cell types (seen in the left-middle part of figure 1). Just as a biological organism uses cellular differentiation to partition different cell tasks, our eCells uses these predefined cell types to partion the task graph into cell types. The four eCell

types are named; `EXPR`, `IF-G BOOL`, `W-G BOOL` and `PAR` (see figure 1). The `EXPR` implements the `<exp>` part of the eDNA language also seen as the `EXPR` on figure 2(a). The `IF-G BOOL` and `W-G BOOL` is the part on figure 2(b)-(c) marked by the dashed box called `IF-G BOOL` and `W-G BOOL` respectively. In addition to [1] the logical granularity have been raised in order to reduce the number of eCells and reduce the communication overhead. Finally, the `PAR` is the cell type which is responsible for joining two parallel parts of the code.

At the end of the transformation step each eCell has partioned the application into multiple cell tasks consisting of the cell types. The final step of the self-organisation is the *mapping* step. At the boot-up each eCell is assigned a unique identifier known as an eCell number (seen in the top right corner of the eCell on figure 1). The eCells then perform a Breadth-First-Search algorithm on the task graph and assigns increasing integers to the tasks as it goes (the resulting numbering of the example in figure 1 is shown in the task-graph). If two or more eCells have equal distance from the source it looks at which node comes first in the eDNA program and the first one is given the lowest number and so on. The eCells then compares the numbers of the task graph with its eCell number. Once it finds the one which is equal, it differentiates to this type of eCell. Furthermore, at boot-up the eCells are also given a routing table, which tell the eCells where all the other eCells are located. By looking at the task graph each eCell can determine which eCells it should communicate with and the position of the those eCells can be found in the routing table. The eDNA platform is now fully functioning and executing the application written in the first step of figure 1.

2.2 Self-healing

The self-healing is the process in which a permanent or transient fault is detected in an eCell (through a Built-in-Self-Test mechanism) and a spare eCell is selected to perform the self-organisation with same eCell number as the faulty eCell had. In that way it will differentiate to become the same cell type as the faulty eCell had. This is possible because each of the eCells have access to the eDNA program. Furthermore, eCells who were communicating with the faulty eCell now communicates with the new version of the faulty eCell. Figure 3 shows the self-healing reconfiguration for the eDNA program of figure 1. In this case three eCell dies and consequently, the functionality is moved to spare eCells.

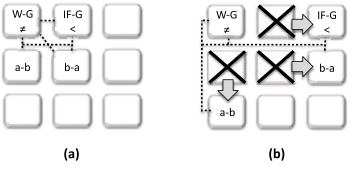


Fig. 3. Self-healing reconfiguration for the eDNA program of figure 1. (a) The optimal placement, (b) Self-healing after three eCell failures.

It is clearly seen that the connections between eCells possibly becomes longer by self-healing. At boot-up every eCell contain a table of coordinates of nearest spare eCells. Each time a spare eCell takes over functionality, it informs all other eCells about this event.

In [1] we showed through simulation how self-organisation and self-healing can be done on the eDNA platform. In this paper the main focus is on how applications running on the eDNA platform performs, consequently we have chosen not to implement the self-organisation and self-healing algorithms on the hardware prototype. For more information aboout the self-organisation and self-healing see [1]. In this work the eDNA program is placed manually in the individual cells in order to observe the performance. The partioning of the code and the mapping of it into different eCells distribute the logic of the program spatially between eCells, this adds an overhead. And it is this overhead which this paper aims at elucidating.

3 Hardware Architecture

The eDNA platform consists of an array of eCells connected through a network (NoC). In the eDNA approach the major overhead is induced by the communication between eCells. Hence, we will focus on evaluating an efficient network structure. The eCell is the basic programming block in the system and it consists of a NA (NA) and a CPU. While the CPU implements the functionality as described by the eDNA program, the NA is responsible for package transfer and routing. This architecture is in contrast to classical NoC's that consists of separated routers and NA's. The combination of router and NA is meaningful for this setup due to the chosen level of granularity of the eCells, the processing time where the CPU blocks the NA is short and the package size is small. Therefore eCells got extended with a store-and-forward (SAF) routing functionality. This leads to a simplified homogeneous structure of the system, reduces the number of hops and eases the failure detection which will be implemented in the final design.

3.1 eCell Architecture

The eCell 1.0 architecture is shown in figure 4. The NA consists of a pair of peripheral switches, a number of registers that are capable of storing a single

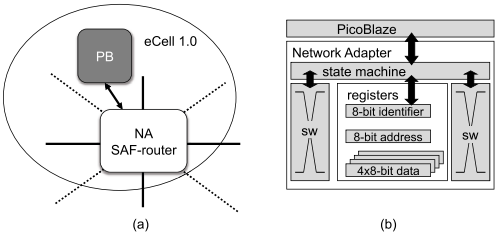


Fig. 4. (a) eCell overview PB: PicoBlaze, (b) eCell architecture

datagram and a state machine that is capable of handling signaling, package transfers, routing algorithm, triggering the CPU interrupt and controling the register accesses.

In order to implement the self-organisation and self-healing algorithm (described in section 2 and in detail in [1]) in the most flexible way and to explore different networks and applications we decided to run a soft core CPU connected to the NA in each eCell. In the final version of the eDNA platform this CPU will be substituted by dedicated logic in order to reduce chip area and complexity. Due to the small amount of resources required and its flexibility the PicoBlaze was chosen. The Xilinx PicoBlaze [11] is based on a 8 bit RISC architecture that is optimized in size for Virtex and Spartan series of FPGAs. It is provided as a synthesisable source-level VHDL file which is easy to extend with additional ports to fit the desired application. There are assembler and C compiler available. It was possible to implement a test setup consisting of a 7x7 array of eCells (PicoBlaze and NA) in a mesh-8 6x8 bit (6 parallel 8 bit registers) datagram architecture on a Digilent XUPV2P board. This setup also includes dedicated input and an output eCells that connect to a serial interface, allowing the user to input data and commands to the system and read the results on a terminal.

The datagram architecture is varied between 6x8 bit, 12x4 bit and 24x2 bit in order to investigate the effect of a higher bandwidth and thus increased parallelism.

The first 8 bit of the datagram are used as package identifier which describes the content of the data (e.g `start/finish` signaling, variable read/write...). The 8 address bits are split up in 4 bit X/Y-coordinates, the address space is therefore limited to 225 cells (0 is no valid address) which is meaningful for test implementations on FPGAs. The remaining 4x8 bit are used to carry data.

3.2 Network Topology and Routing

The distributed approach used in the eDNA platform creates a significant overhead by transmitting data packages among the eCells. In addition to the data exchange, the `start/finish` signaling is implemented purely package-based and will generate additional traffic in the NoC. It is therefore very important that the prospective network is able to forward packages in a simple and fast manner since the performance of the whole system greatly depends on the network properties.

In this paper we will investigate two different versions of the mesh-topology; the mesh-4 and mesh-8. 4 and 8 are the number of neighbors that each eCell has. Both networks offers a good compromise between used chip area and number of hops. In addition they offer the possibility of implementing a simple and efficient routing algorithm. Both topologies is plausible to fabricate on modern multi-layer chips.

When a NA of an eCell receives a package, it checks whether the destination address of the package is reached. If not, then the NA determines the direction of the destination eCell and sets the output switch to the corresponding position. Only if the package destination address is reached, the CPU is interrupted to

perform the eDNA functionality. When the CPU has generated a new package, the NA checks whether the destination is available by handshaking. This ensures package rerouting in case of network faults.

Before transmitting, the NA makes sure that the next eCell on the direct way to the destination eCell is ready. This is done by handshaking and also includes a dedicated signal which reports whether the corresponding eCell is alive or not. In case the next eCell is busy, the package is sent to one of the neighboring eCells. This mechanism ensures that dead or busy eCells on the way to the destination can be sidestepped. In the eDNA prototype platform, SAF routers are used and a datagram is equal to a flit. This flow control digit is defined as the smallest unit of flow control. Due to these design decisions and the fact that only one package per `parallel` statement is routed in the network at the time, deadlocks and livelocks can be avoided.

3.3 Limitations

It is important to realise that the eDNA 1.0 prototype has some properties, which limits the performance of it:

eDNA synchronisation protocol. In order to ensure that no race conditions exist (between data and start/finish-signaling) the eCells uses a standard handshaking protocol. Each time an eCell sends data to another eCell it waits on an acknowledge-signal from the receiving eCell before sending the start/finish signal.

Sequential data processing in eCells. Even though each eCell is connected to 4 or 8 (depending on the topology) neighbors and the fact that all eDNA operations require at least two operands, the eCell is unable to process incoming data in parallel. This is because in the NA there is only space for storing one data package at a time, which only allows serial operation. This architecture was chosen to match the PicoBlaze functionality.

8-bit PicoBlaze. Due to the 8-bit data bus width of the PicoBlaze, higher bit operations becomes computational heavy and slow. Additionally the interrupt response time of maximum 5 clock cycles and the sequential analysis of the package identifier add a significant delay.

4 Experimental Results

A key contributor to the overhead of the eDNA platform is the network. Consequently, it is important to find the right network setup for the job. First of all, we want to figure out whether it is most valuable to have a high bandwidth or a high amount of neighbors. Secondly, we investigate the performance of the benchmarks in order to show that eDNA 1.0 performs at a level where it would be feasible to use it in an industrial application, and finally, we investigate the performance impact of self-healing.

The benchmarks used are; Euclids Greatest Common Divisor (GCD), Fibonacci number generator (FIB), CORDIC-algorithm (CORDIC), Binary Search

Table 1. The number/connections of eCells in each benchmark

	GCD	FIB	CORDIC	BS	RSA
# cells	4	5	11	7	5
# connections	16	19	40	22	14

Table 2. #Wires pr. eCell for different architectures

	6x8 bit	12x4 bit	24x2 bit
Mesh4	56	104	200
Mesh8	112	208	400

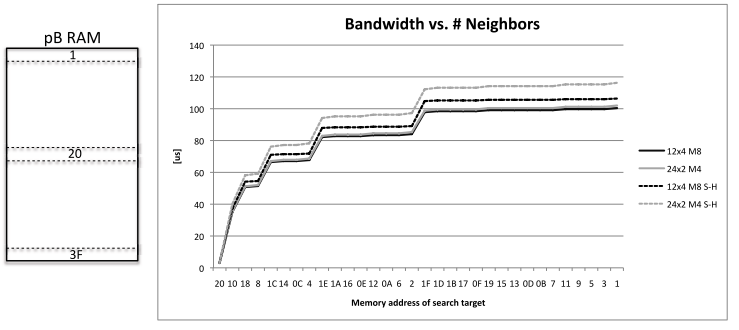


Fig. 5. BS run on two different architectures with optimal placement and after self-healing (S-H). The lefthand side of figure displays the meaning of the x-axis.

(BS) and RSA-encryption (RSA). Table 1 shows how many eCells the different benchmarks occupy and the number of connections between them. In the experiments the benchmarks were run with all possible inputs. The benchmarks were implemented to fit the PicoBlaze architecture in order to execute as efficient as possible.

In order to find out whether having a high bandwidth or a high amount of neighbors is the best, BS is implemented on two architectures of eDNA 1.0, assuming a fixed available area; a 12x4 bit mesh-8 and a 24x2 bit mesh-4. It is clear, that having 24x2 mesh-8 would be the fastest, but the cost of wires is a lot bigger as illustrated in 2. It is also interesting to look at the performance degradation due to self-healing, i.e. making the distance a package has to travel between eCells longer. When the BS algorithm is executed, we search for targets at different addresses in the PicoBlaze RAM block and measure the execution time. Figure 5 shows the performance of BS for these architectures.

The figure shows that there is nearly no difference when running an optimally placed application. In this case "optimal" means that the eCells are placed in a way, such that the number of package transfers are minimised. Due to handshaking and routing overhead the 24x2 bit architecture is able to transmit a datagram

just slightly faster than the 12x4 bit architecture (2 clock cycles). Therefore having more neighbours has a higher positive impact on the performance than the higher bandwidth, which can clearly be seen after self-healing where the number of hops is higher. In this case two eCells were moved to a neighboring spare eCell as shown in figure 3. Consequently, it is clear that the mesh-8 is the best NoC topology for this application and it will be used in the following experiments.

The next set of experiments will uncover the overhead of running applications on eDNA 1.0 and also investigate for all the benchmarks, whether it pays off to go for the highest bandwidth. In order to get reliable performance results, we ran the benchmarks with all possible inputs and computed the avareage execution time. In order to make a fair comparison of the execution times of the benchmarks we compared the performance to that of the implementation of the benchmark on a single PicoBlaze, because the single PicoBlaze implementations do not suffer from communication overhead. So by comparing to a single PicoBlaze we get the overhead related to the distributed logic approach of the eDNA platform.

To elaborate on the overhead by the eDNA implementation, the averaged execution times of the different benchmarks are normalised to the execution time of the single PicoBlaze implementation (where normalisation is $\frac{time_{eDNA}}{time_{pB}}$). Figure 6 clearly shows that the overhead is large, but due to the benefits gained from this architecture it can be argued that these results are acceptable as will be discussed in section 5. The figure also shows that in all cases the 24x2 is the best as expected. However, the positive impact of having the higher bandwidth is higher for the BS, GCD and FIB. The reason for this is that the BS, GCD and FIB algorithms are communication intensive. Therefore, the impact of having a higher bandwidth is higher in these cases. CORDIC and the RSA are computational intensive and not communication intensive. While CORDIC uses 32 bit precision, the RSA contains a multiplier, which is why these are more computational

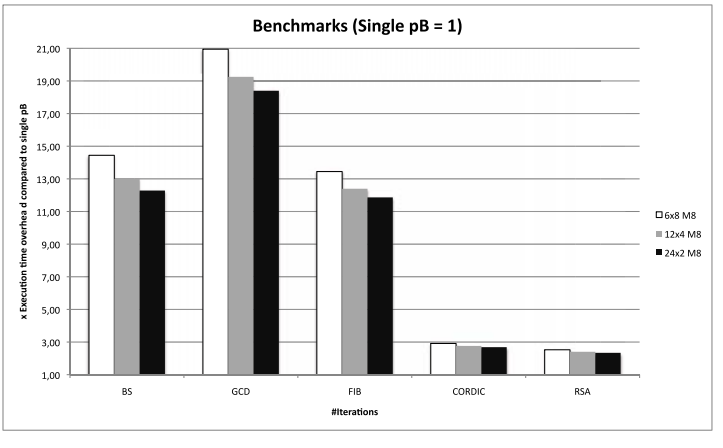


Fig. 6. Execution time overhead for each of the 5 benchmarks normalised to the benchmark running on a single PicoBlaze. The eCells are placed optimally for each of the benchmarks.

heavy. Consequently, we can conclude that the more communication intensive an application is the bigger the overhead becomes. For a computational heavy application the overhead is around 2x and for a computational light application the overhead is at most 18x for the various tested benchmarks. Furthermore, by looking at table 1 and figure 6 it is clear that there are no direct relation between the size of the application and the overhead.

Now we want to investigate how big the performance degradation is when the system is self-healing. When an eCell die its functionality is taken over by a neighboring spare-eCell which tears the original setup apart, thus creating longer communication paths. Consequently, when the eCells are spread over a bigger area, the network traffic and latency increases and lowers its performance. Figure 3 demonstrates the self-healing reconfiguration for the GCD benchmark. A similar procedure is used for all the benchmarks, i.e. picking three eCells and moving them to their neighboring spare eCell. Three eCells were chosen because they correspond to a significant amount of functionality for all the benchmarks. The results for this self-healing is compared to the optimal placement from the preceding experiment. Figure 7 shows the self-healing results. It is very clear that the impact of self-healing is very small - between 0%-20% and as expected the performance impact is greater for the more communication heavy applications.

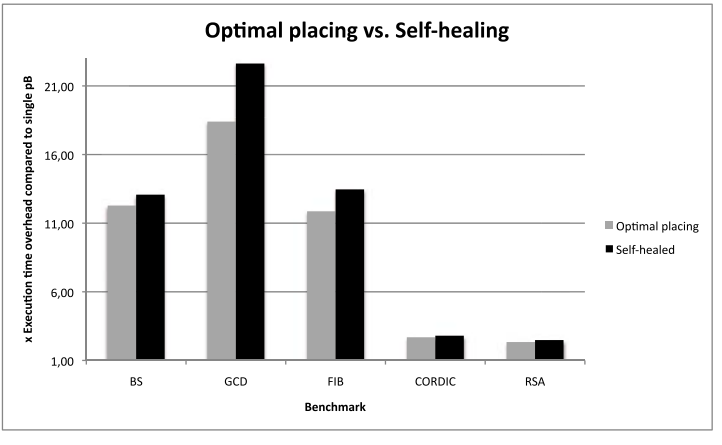


Fig. 7. Execution time overhead for each of the 5 benchmarks in a mesh-8 24x2 architecture when it is optimally placed and when it is self-healed. Normalised to the execution time of the benchmark running on a single PicoBlaze.

5 Discussion and Conclusion

To the best of our knowledge this is the only performance study of a reconfigurable self-healing hardware architecture so far. The results show that depending on the weight of the computation for the application implemented on the eDNA

1.0 the runtime overhead varies between 2x-18x compared to a single processor implementation. Self-healing of three eCells adds another overhead in the range of up to 20% in a realistic worst case scenario. The self-healing scenario simulated in the paper can be considered worst-case because in [1] we propose a protocol which also ways creates a spare eCell next to the dead eCell, simply by moving other eCells in a predictable manner. Those results could be reached mainly by selecting a suitable level of granularity and an efficient NoC design. These results are satisfying keeping the advantages of the bio inspired self healing and the previously mentioned design limitations in mind. In contrast to the static redundancy approach, the eDNA architecture offers dynamic reconfiguration, a homogeneous structure and the possibility to save power since spare eCells do not need to be clocked. Furthermore, the performance will be increased by removing the limitations imposed on eDNA 1.0. A major limitation on eDNA 1.0 is the synchronisation protocol. The current setup works in the way that when a value of a variable changes all eCells that carry this variable are updated sequentially with corresponding acknowledge messages in order to avoid synchronisation problems. The eDNA 2.0 will have a more advanced synchronisation protocol which does not rely on acknowledgements. In addition, the Picoblaze will be replaced by dedicated hardware that processes data in parallel. This requires a Network Adapter that is able to handle parallel incoming data packages. By removing the PicoBlaze we will also get rid of the interrupt response time and sequential package identifier analysis, which is a significant part of the overhead. These design decisions will improve the performance of the eDNA platform, while adding fault detection on the other hand will increase overhead.

In this work we have presented the eDNA 1.0 prototype. The paper shows that eDNA 1.0 performs within acceptable bounds especially when the limitations imposed by its prototype stage is taken into account. It has been argued that eDNA 2.0 will perform even better due to the improvement of the time consuming synchronisation protocol. In addition the absolute system performance will raise a lot when the serial 8-bit Picoblaze CPU is replaced by dedicated hardware. Thus our belief that eDNA 2.0 will result in an entirely new type of FPGA is still upheld.

References

1. Boesen, M.R., Madsen, J.: edna: A bio-inspired reconfigurable hardware cell architecture supporting self-organisation and self-healing. In: Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware Systems, pp. 147–154 (2009)
2. Mange, D., Sipper, M., Stauffer, A., Tempesti, G.: Toward robust integrated circuits: The embryonics approach. Proceedings of the IEEE 88, 516–543 (2000)
3. Zhang, X., Dragffy, G., Pipe, A., Gunton, N., Zhu, Q.: A reconfigurable self-healing embryonic cell architecture. In: International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA 2003, pp. 134–140 (2003)
4. Stauffer, A., Rossier, J.: Self-testable and self-repairable bio-inspired configurable circuits. In: 2009 NASA/ESA Conference on Adaptive Hardware Systems, pp. 155–162 (2009)

5. Samie, M., Dragffy, G., Popescu, A., Pipe, T., Melhuish, C.: Prokaryotic bio-inspired model for embryonics. In: 2009 NASA/ESA Conference on Adaptive Hardware Systems, pp. 163–170 (2009)
6. Kachris, C., Wong, S., Vassiliadis, S.: Design and performance evaluation of an adaptive fpga for network applications. Microelectronics Journal 40, 1103–1110 (2009)
7. Shukla, S., Bergmann, N.W., Becker, J.: Quku: A fast run time reconfigurable platform for image edge detection. In: Bertels, K., Cardoso, J.M.P., Vassiliadis, S. (eds.) ARC 2006. LNCS, vol. 3985, pp. 93–98. Springer, Heidelberg (2006)
8. Filho, J.O., Schweizer, T., Oppold, T., Kuhn, T., Rosenstiel, W.: Tuning coarse-grained reconfigurable architectures towards an application domain. In: 2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006), pp. 1–7 (2006)
9. Xilinx: Microblaze processor reference guide - edk 10.1i. Xilinx User Guide UG081, v9.0 (2008)
10. Page, I.: Constructing hardware-software systems from a single description. Journal of VLSI Signal Processing, 87–107 (1996)
11. Chapman, K.: Picoblaze 8-bit embedded microcontroller for spartan-3, virtex-ii, and virtex-ii pro fpgas. Xilinx User Guide UG129, v1.1.2 (2008)

Application-Specific Signatures for Transactional Memory in Soft Processors

Martin Labrecque, Mark Jeffrey, and J. Gregory Steffan

Department of Electrical and Computer Engineering, University of Toronto
{martinl,markj,steffan}@eecg.toronto.edu

Abstract. As reconfigurable computing hardware and in particular FPGA-based systems-on-chip comprise an increasing number of processor and accelerator cores, supporting sharing and synchronization in a way that is scalable and easy to program becomes a challenge. *Transactional memory* (TM) is a potential solution to this problem, and an FPGA-based system provides the opportunity to support TM in hardware (HTM). Although there are many proposed approaches to HTM support for ASICs, these do not necessarily map well to FPGAs. In particular in this work we demonstrate that while *signature*-based conflict detection schemes (essentially bit vectors) should intuitively be a good match to the bit-parallelism of FPGAs, previous schemes result in either unacceptable multicycle stalls, operating frequencies, or false-conflict rates. Capitalizing on the reconfigurable nature of FPGA-based systems, we propose an application-specific signature mechanism for HTM conflict detection. Using both real and projected FPGA-based soft multiprocessor systems that support HTM and implement threaded, shared-memory network packet processing applications, relative to signatures with bit selection we find that our application-specific approach (i) maintains a reasonable operating frequency of 125MHz, (ii) has an area overhead of only 5%, and (iii) achieves a 9% to 71% increase in packet throughput due to reduced false conflicts.

1 Introduction

As reconfigurable computing systems and in particular FPGAs become more dense, they are increasingly used to implement larger and more complex systems-on-chip composed of multiple processor and acceleration cores that must synchronize and share data. While systems based on shared memory can ease communication between cores, the programmer's job of inserting correct lock-based synchronization can be error-prone and difficult to debug, and the resulting *critical sections* of code within locks are serialized, thus reducing the overall parallelism and efficiency of the system.

Transactional memory (TM) [1, 2, 3] can potentially address both challenges. First, TM provides an easier programming model for synchronization, allowing programmers to specify more coarse-grain critical sections (transactions) to be executed atomically. Second, TM reduces contention on these larger critical

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 42–54, 2010.

sections by executing transactions in parallel so long as their memory accesses do not conflict. Hence we are motivated to implement TM for multiple-core reconfigurable computing systems; in this paper we focus on implementing TM for an FPGA-based soft multiprocessor. While TM can be implemented purely in software (STM), an FPGA-based system can be extended to support TM in hardware (HTM) with much lower performance overhead than an STM. There are many known methods for implementing HTM for an ASIC multicore processor, although they do not necessarily map well to an FPGA-based system.

In this paper we focus specifically on the design of the conflict detection mechanism for FPGA-based HTM, and find that an approach based on *signatures* [4] is a good match for FPGAs because of the underlying bit-level parallelism. A signature is essentially a bit-vector [5] that tracks the memory locations accessed by a transaction via hash indexing. However, since signatures normally have many fewer bits than there are memory locations, comparing two signatures can potentially indicate costly false-positive conflicts between transactions. Hence prior HTMs employ relatively large signatures—thousands of bits long—to avoid such false conflicts. One important goal for our system is to be able to compare signatures and detect conflicts in a single pipeline stage, otherwise memory accesses would take an increasing number of cycles and degrade performance. However, as we demonstrate in this paper, implementing previously proposed large signatures in the logic-elements of an FPGA can be detrimental to processor operating frequency. Or, as an equally unattractive alternative, one can implement large and sufficiently fast signatures using block RAMs but only if the indexing function is trivial—which can itself exacerbate false-positive conflicts and negate the value of larger signatures.

1.1 Application-Specific Signatures

To summarize, our goal is to implement a moderately-sized signature mechanism while minimizing the resulting false conflicts. We capitalize on the reconfigurable nature of the underlying FPGA and propose a method for implementing an application-specific signature mechanism that achieves these goals. An application-specific signature is created by (i) profiling the memory addresses accessed by an application, (ii) using this information to build and optimize a *trie* (a tree based on address prefixes) that allocates more branches to frequently-conflicting address prefixes, and (iii) implementing the trie in a conflict detection unit using simple combinational circuits.

Our evaluation system is built on the NetFPGA platform [6], comprising a Virtex II Pro FPGA, 4 1GigE MACs, and 200MHz DDR2 SDRAM. On it we have implemented a dual-core multiprocessor (the most cores that our current platform can accommodate), composed of 125MHz MIPS-based soft processors, that supports an *eager* HTM [7] via a shared data cache. We have programmed our system to implement several threaded, shared-memory network packet processing applications (packet classification, NAT, UDHCP, and intrusion detection).

We use a cycle-accurate simulator to explore the signature design space, and implement and evaluate the best schemes in our real dual-core multiprocessor implementation. For comparison, we also report the FPGA synthesis results for a conflict detection unit supporting 4 and 8 threads. Relative to signatures with bit selection (the only other signature implementation that can maintain a reasonable operating frequency of 125MHz), we find that our application-specific approach has an area overhead of only 5%, and achieves a 9% to 71% increase in packet throughput due to reduced false conflicts.

1.2 Related Work

There is an abundance of prior work on TM and HTM. Most prior FPGA implementations of HTM were intended as fast simulation platforms to study future multicore designs [1, 2], and did not specifically try to provide a solution tuned for FPGAs. Conflict detection has been previously implemented by checking extra bits per line in private [1, 2] or shared [3] caches. In contrast with caches with finite capacity that require complex mechanisms to handle cache line collisions for speculative data, signatures can represent an unbounded set of addresses and thus do not overflow. Signatures can be efficiently cleared in a single cycle and therefore advantageously leverage the bit-level parallelism present in FPGAs. Because previous signature work was geared towards general purpose processors [5, 8, 9], to the best of our knowledge there is no prior art in customizing signatures on a per-application basis.

1.3 Contributions

This paper makes the following contributions: (i) we describe the first soft processor cores integrated with transactional memory, and evaluated on a real (and simulated) system with threaded applications that share memory; (ii) we demonstrate that previous signature schemes result in implementations with either multicycle stalls, or unacceptable operating frequencies or false conflict rates; (iii) we demonstrate that application-specific signatures can allow conflict detection at acceptable operating frequencies (125MHz), single cycle operation, and improved false conflict rates—resulting in significant performance improvements over alternative schemes.

2 Previous Signature Implementations for HTM

A TM system must track read and write accesses for each transaction (the read and write sets), hence an HTM system must track read and write sets for each hardware thread context. The signature method of tracking read and write sets implements Bloom filters [5], where an accessed memory address is represented in the signature by asserting the k bits indexed by the results of k distinct hashes of the address, and a membership test for an address returns true only if all k bits are set. Since false conflicts can have a significant negative impact

on performance, the number and type of hash functions used must be chosen carefully. In this paper we consider only the case where each one of the k hash functions indexes a different partition of the signature bits—previously shown to be more efficient [5]. The following reviews the four known hash functions that we consider in this paper.

Bit Selection. [5] This scheme directly indexes a signature bit using a subset of address bits. An example 2-bit index for address $a = [a_3 a_2 a_1 a_0]$ could simply be $h = [a_3,\ a_2]$. This is the most simple scheme (i.e., simple circuitry) and hence is important to consider for an FPGA implementation.

$\mathbf{H_3}$. [5] The H_3 class of hash functions is designed to provide a uniformly-distributed hashed index for random addresses. Each bit of the hash result $\mathbf{h} = [h_1,\ h_0]$ consists of a separate XOR ($\oplus$) tree determined by the product of an address $a = [a_3 a_2 a_1 a_0]$ with a fixed random matrix H as in the following example with a 4-bit address and a 2-bit hash [9]:

$$[h_1,\ h_0] = \mathbf{a}H = [a_3 a_2 a_1 a_0] \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = [a_3 \oplus a_2 \oplus a_0,\ a_2 \oplus a_1] \tag{1}$$

Page-Block XOR (PBX). [8] This technique exploits the irregular use of the memory address space to produce hash functions with fewer XOR gates. An address is partitioned into two non-overlapping bit-fields, and selected bits of each field are XOR'ed together with the purpose of XOR'ing high entropy bits (from the low-order bit-field) with lower entropy bits (from the high order bit-field). Modifying the previous example, if the address is partitioned into 2 groups of 2 bits, we could produce the following example 2-bit hash: $[a_2 \oplus a_0,\ a_3 \oplus a_1]$.

Locality-sensitive XOR. [9] This scheme attempts to reduce hash collisions and hence the probability of false conflicts by exploiting memory reference spatial locality. The key idea is to make nearby memory locations share some of their k hash indices to delay filling the signature. This scheme produces k H_3 functions that progressively omit a larger number of least significant bits of the address from the computation of the k indices. When represented as H_3 binary matrices, functions require an increasing number of lower rows to be null. Our implementation, called LE-PBX, combines this approach with the reduced XOR'ing of PBX hashing. In LE-PBX, we XOR high-entropy bits with low-entropy bits within a window of the address, then shift the window towards the most significant (low entropy) bits for subsequent hash functions.

3 Application-Specific Signatures

All the hashing functions listed in the previous section create a random index that maps to a signature bit range that is a power of two. In Section 5 we

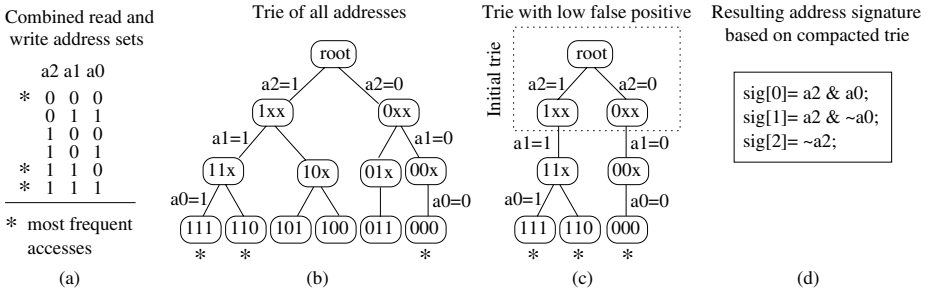


Fig. 1. Example trie-based signature construction for 3-bit addresses. We show (a) a partial address trace, where * highlights frequently accessed addresses, (b) the full trie of all addresses, (c) the initial and final trie after expansion and pruning to minimize false positives, and (d) the logic for computing the signature for a given address (i.e., to be AND'ed with read and write sets to detect a conflict).

demonstrate that these functions require too many bits to be implemented without dramatically slowing down our processor pipelines. To minimize the hardware resources required, the challenge is to reduce the number of false conflicts per signature bit, motivating us to more efficiently utilize signature bits by creating application-specific hash functions.

Our approach is based on *compact trie hashing* [10]. A trie is a tree where each descendant of a node has in common the prefix of most-significant bits associated with that node. The result of the hash of an address is the leaf position found in the tree, corresponding to exactly one signature bit. Because our benchmarks can access up to 64 Mbytes of storage (16 million words), it is not possible to explicitly represent all possible memory locations as a leaf bit of the trie. The challenge is to minimize false conflicts by mapping the most contentious memory locations to different signature bits, while minimizing the total number of signature bits.

We use a known greedy algorithm to compute an approximate solution to this NP-complete problem [11]. In the first step, we record in our simulator a trace of the read and write sets of a benchmark functioning at its maximum sustainable packet rate. We organize the collected memory addresses in a trie in which every leaf represents a signature bit. This signature is initially too large to be practical (Figure 1(b)) so we truncate it to an initial trie (Figure 1(c)), selecting the most frequently accessed branches. To reduce the hardware logic to map an address to a signature (Figure 1(d)), only the bits of the address that lead to a branch in the trie are considered. For our signature scheme to be safe, an extra signature bit is added when necessary to handle all addresses not encompassed by the hash function. We then replay the trace of accesses and count false conflicts encountered using our initial hashing function. We iteratively expand the trie with additional branches and leaves to eliminate the most frequently occurring false-positive conflicts (Figure 1(c)). Once the trie is expanded to a desired false positive rate, we greedily remove signature bits that do not negatively impact the

false positive rate (they are undesirable by-products of the expansion). Finally, to further minimize the number of signature bits, we combine signature bits that are likely (> 80%) to be set together in non-aborted transactions.

4 Evaluation Infrastructure

Table 1 describes our evaluation infrastructure including the platform and compilation, and how we do timing, validation, and measurement. Due to stringent timing requirements (there are no free PLLs after merging-in the NetFPGA support components), and despite some available area on the FPGA, (i) our caches are limited to 16KB each, and (ii) we are also limited to a maximum of two processors. These limitations are not inherent in our architecture, and would be relaxed in a system with more PLLs and a more modern FPGA. The rest of this section describes our system architecture and benchmark applications.

Table 1. Evaluation infrastructure

Aspect	Description
Compilation	Modified `gcc` 4.0.2, `Binutils` 2.16, and `Newlib` 1.14.0
Instruction set	32-bit MIPS-I ISA without delay slots [12], with software mul and div
FPGA	Virtex II Pro 50 speed grade 7ns
Platform	NetFPGA 2.1 [6] with 4 x 1GigE Media Access Controllers (MACs)
Synthesis	Xilinx ISE 10.1.03, high effort to meet timing constraints
Off-chip memory	64 Mbytes 200MHz DDR2 SDRAM, Xilinx MIG controller
Processor clock	125MHz, same as Ethernet MACs
Validation	Execution trace generated in RTL simulation and online in debug mode, compared against cycle-accurate simulator built on MINT [13]
Measuring host	Linux 2.6.18 Dell PowerEdge 2950 with two quad-core 2GHz Xeon processors
Packet source	Modified `Tcpreplay` 3.4.0 sending packet traces from a Broadcom NetXtreme II GigE NIC to an input port of the NetFPGA
Packet sink	NetXtreme GigE NIC connected to another NetFPGA port used for output
Max. throughput	Smallest fixed packet inter-arrival rate without packet drop, obtained through bisection search (we empirically found 5 second runs to be sufficient)

System architecture. Our base processor is a single-issue, in-order, single-threaded, 5-stage pipelined processor. To eliminate the critical path for hazard detection logic, we employ static hazard detection [14] in our architecture / compiler. The processor is big-endian which avoids having to do network-to-host byte ordering transformations. Each processor in Figure 2 has a 16 KB private instruction cache. The SDRAM controller services a merged load/store queue of up to 64 entries in-order; since this queue is shared by all processors it serves as a single point of serialization and memory consistency, hence threads need only block on pending loads but not stores. As described in Table 2, our multiprocessor architecture is bus-based and sensitive to the two-port limitation of block RAMs available on FPGAs. In its current form it will not easily scale to a large number of processors. However, as we demonstrate later in Section 5, our applications are mostly limited by synchronization and critical sections, and not contention on the shared buses; in other words, the synchronization inherent in the applications is the primary roadblock to scalability.

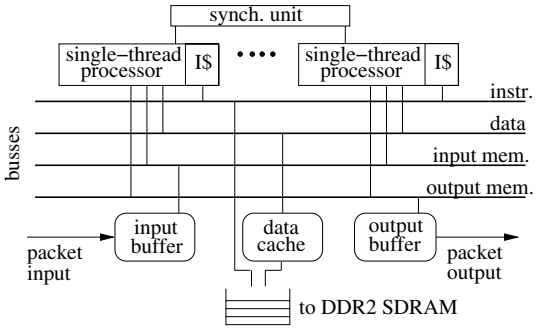


Fig. 2. The architecture of our soft multiprocessor with 2 single-threaded processor cores

Table 2. On-chip memory hierarchy

Memory	Description
Input buffer	Receives packets on one port and services processor requests on the other port, read-only, logically divided into ten fixed-sized packet slots
Output buffer	Sends packets to the NetFPGA MAC controllers on one port, connected to the processors via its second port
Data cache	Connected to the processors on one port, 32-bit line-sized data transfers with the DDR2 SDRAM controller (similar to previous work [15]) on the other port
All three	16KB, single-cycle random access, arbitrated across processors, 32 bits bus

Transactional memory support. The single port from the processors to the shared cache in Figure 2 implies that memory accesses undergo conflict detection one by one in transactional execution, therefore a single trie hashing unit suffices for both processors. Our transactional memory processor uses a shadow register file to revert its state upon rollback (versioning [16] avoids the need for register copy). Speculative memory-writes trigger a backup of the overwritten value in an undo-buffer [7] that we over-provision with storage for 2048 values per thread. Each processor has a dedicated connection to a synchronization unit that triggers the beginning and end of speculative executions when synchronization is requested in software.

Applications. Network packet processing is no longer limited solely to routing, with many applications that require deeper packet inspection becoming increasingly common and desired. We focus on *stateful* applications—i.e., applications in which shared, persistent data structures are modified during the processing of most packets. Our processors process packets from beginning-to-end by executing the same program, because the synchronization around shared data structures makes it impractical to extract parallelism otherwise (e.g. with a pipeline of balanced execution stages). To take full advantage of the software programmability of our processors, our focus is on the control-flow intensive applications described in Table 3. While we could enforce ordering in software, we allow packets to be processed out-of-order because our application semantics allow it.

Table 3. Applications and their mean statistics

Benchmark	Description	Dyn. Instr. /packet	Dyn. Instr. /transaction	Uniq. Sync. Addr. /transaction	
				Reads	Writes
`Classifier`	Regular expression matching on TCP packets for application recognition.	2553	1881	67	58
`NAT`	Network address translation plus statistics.	2057	1809	50	41
`UDHCP`	Modified open-source DHCP server.	16116	3265	430	20
`Intruder`	Network intrusion detection [17] modified to have packetized input.	12527	399	37	23

5 Results

In this section we first evaluate the impact of signature scheme and length on false-positive conflicts, application throughput, and implementation cost. These results guide the implementation and evaluation of our real system.

Resolution of Signature Mechanisms. Using a recorded trace of memory accesses obtained from a cycle-accurate simulation of our TM system that models perfect conflict detection, we can determine the false-positive conflicts that would result from a given realistic signature implementation. We use a recorded trace because the false positive rate of a dynamic system cannot be determined without affecting the course of the benchmark execution: a dynamic system cannot distinguish a false-positive conflict from a later true conflict that would have happened in the same transaction, if it was not aborted immediately. We compute the false positive rate as the number of false conflicts divided by the total number of transactions, including repeats due to rollback.

The signatures that we study are configured as follows. The bit selection scheme selects the least significant word-aligned address bits, to capture the most entropy. For H3, PBX and LE-PBX, we found that increasing the number of hash functions caused a slight increase in the false positive rate for short signatures, but helped reduce the the number of signature bits required to completely eliminate false positives. We empirically found that using four hash functions is a good trade-off between accuracy and complexity, and hence we do so for all results reported. To train our trie-based hash functions, we use a different but similarly-sized trace of memory accesses as a training set.

Figure 3 shows the false positive rate for different hash functions (bit selection, H3, PBX, LE-PBX and trie-based) as signature bit length varies. The false positive rate generally decreases with longer signatures because of the reduced number of collisions on any single signature bit—although small fluctuations are possible due to the randomness of the memory accesses. Our results show that LE-PBX has a slightly lower false positive rate than H3 and PBX for an equal number of signature bits. Bit selection generally requires a larger number of signature bits to achieve a low false positive rate, except for UDHCP for which most of the memory accesses point to consecutive statically allocated data. Overall, the trie scheme outperforms the others for Classifier, NAT

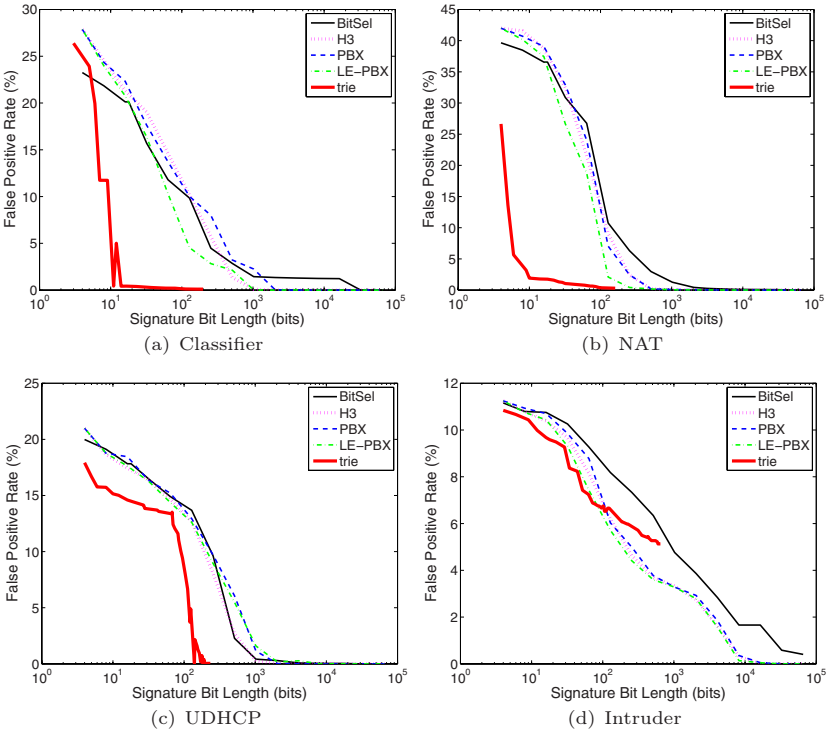
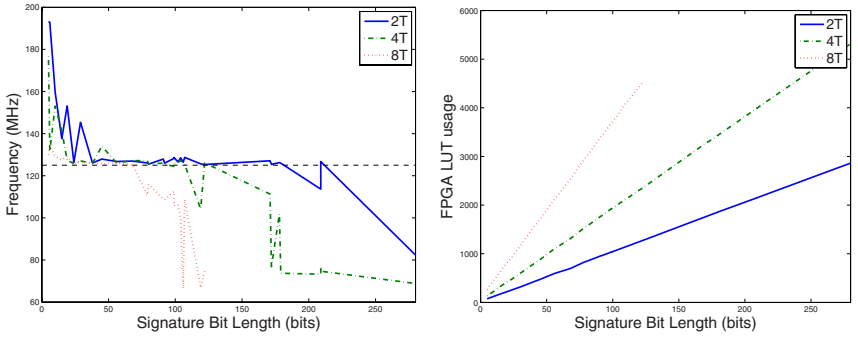


Fig. 3. False positive rate vs signature bit length. Trie-based signatures were extended in length up to the length that provides zero false positives on the training set.

and UDHCP by achieving close to zero false positive rate with less than 100 bits, in contrast with several thousand bits. For INTRUDER the non-trie schemes have a better resolution for signatures longer than 100 bits due to the relatively large amount of dynamic memory used, which makes memory accesses more random. Quantitatively we can compute the entropy of accesses as $\sum_{i=0}^{n-1} -p(x_i)\log_2 p(x_i)$ where $p(x_i)$ is the probability of an address appearing at least once in a transaction—with this methodology INTRUDER has an entropy 1.7 times higher on average than the other benchmarks, thus explaining the difficulty in training its trie-based hash function.

Implementation of a Signature Mechanism. Figure 4 shows the results of implementing a signature-based conflict detection unit using solely the LUTs in the FPGA for a processor system with 2 threads like the one we implemented (`2T`) and for hypothetical transactional systems with 4 and 8 threads (`4T` and `8T`). While the plot was made for a trie-based hashing function, we found that H3, PBX and LE-PBX produced similar results. As we will explain later, the bit selection scheme is better suited to a RAM-based implementation. In Figure 4(a) we observe that the CAD tools make an extra effort to meet our 125 MHz



(a) Frequency (b) LUT usage

Fig. 4. Impact of increasing the bit length of trie-based signatures on (a) frequency and (b) LUT usage of the conflict detection unit for 2, 4, and 8-thread (2T,4T,8T) systems. The results for H3, PBX and LE-PBX are similar. In (a) we highlight the system operating frequency of 125MHz.

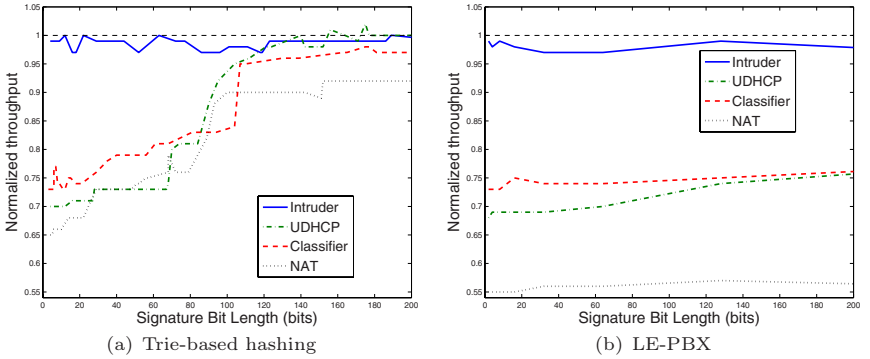
required operating frequency by barely achieving it for many designs. In a 2-thread system, two signatures up to 200 bits will meet our 125MHz timing requirement while a 4-thread system can only accommodate four signatures up to 100 bits long. For 8-threads, the maximum number of signature bits allowed at 125MHz is reduced to 50 bits. Figure 4(b) shows that the area requirements grow linearly with the number of bits per signature. In practice for 2-threads at 200 bits, signatures require a considerable amount of resources: approximatively 10% of the LUT usage of the total non-transactional system. When the conflict detection unit is incorporated into the system, we found that its area requirements—by putting more pressure on the routing interconnect of the FPGA—lowered the maximum number of bits allowable to less than 100 bits for our 2-thread system (Table 4). Re-examining Figure 3, we can see that the trie-based hashing function delivers significantly better performance across all the hashing schemes proposed for less than 100 signature bits.

An alternate method of storing signatures that we evaluate involves mapping an address to a signature bit corresponding to a line in a block RAM. On that line, we store the corresponding read and write signature bit for each thread. To preserve the 125MHz clock rate and our single-cycle conflict detection latency, we found that we could only use one block RAM and that we could only use bit selection to index the block RAM—other hashing schemes could only implement one hash function with one block RAM and would perform worse than bit selection in that configuration. Because the data written is only available on the next clock cycle in a block RAM, we enforce stalls upon read-after-write hazards. Also, to emulate a single-cycle clear operation, we version the read and write sets with a 2-bit counter that is incremented on commit or rollback to distinguish between transactions. If a signature bit remains untouched and therefore preserves its version until a transaction with an aliasing version accesses it (the version wraps over a 2-bit counter), the bit will appear to be

set for the current transaction and may lead to more false positives. The version bits are stored on the same block RAM line as their associated signature bits, thus limiting the depth of our 16Kb block RAM to 2048 entries (8-bits wide). Consequently, our best bit selection implementation uses a 11 bit-select of the word-aligned least-significant address bits.

Impact of False Positives on Performance. Figure 5 shows the impact on performance in a full-system simulation of a varying signature length, when using either a trie-based hashing function or LE-PBX, the scheme with the second-lowest false positive rate. The jitter in the curves is again explained by the unpredictable rollback penalty and rate of occurrence of the false positives, varying the amount contention on the system. Overall, we can see that signatures have a dramatic impact on system throughput, except for INTRUDER for which the false positive rate varies little for this signature size range (Figure 3(d)). We observe that for CLASSIFIER, UDHCP and NAT, although they achieve a small false positive rate with 10 bits on a static trace of transactional accesses (Figure 3), their performance increases significantly with longer signatures. We found that our zero-packet drop policy to determine the maximum throughput of our benchmarks is very sensitive to the compute-latency of packets since even a small burst of aborts and retries for a particular transaction directly impacts the size of the input queue which in turn determines packet drops. The performance of NAT plateaus at 161 bits because that is the design that achieves zero false positives in training (Figure 3(b)). As expected, Figure 5(b) shows that there is almost no scaling of performance for LE-PBX in the possible signature implementation size range because the false positive rate is very high.

Measured Performance on the Real System. As shown in Table 4 and contrarily to the other schemes presented, the size of the trie-based signatures can be adjusted to an arbitrary number of bits to maximize the use of the



(a) Trie-based hashing (b) LE-PBX

Fig. 5. Throughput with signatures using trie-based hashing with varying signature sizes normalized to the throughput of an ideal system with perfect conflict detection (obtained using our cycle-accurate simulator)

Table 4. Size, LUT usage, LUT overhead and throughput gain of our real system with the best application-specific trie-based hash functions over bit selection

Benchmark	Max. Signature bits	Total LUT usage	LUT overhead	Additional throughput
Classifier	92	20492	5%	12%
NAT	68	20325	4%	58%
UDHCP	84	20378	4%	9%
Intruder	96	20543	5%	71%

FPGA fabric while respecting our operating frequency. The maximum signature size is noticeably smaller for NAT because more address bits are tested to set signature bits, which requires more levels of logic and reduces the clock speed. In all cases the conflict detection with a customized signature outperforms the general purpose bit selection. This is coherent with the improved false positive rate observed in Figure 3. We can see that bit selection has the best performance when the data accesses are very regular as in UDHCP, as indicated by the low false positive rate in Figure 3(c). Trie-based hashing improves the performance of Intruder the most because the bit selection scheme suffers from bursts of unnecessary transaction aborts.

CAD Results. Comparing two-processor full system hardware designs, the system with trie-based conflict detection implemented in LUTs consumes 161 block RAMs and the application-specific LUT usage reported in Table 4. Block-RAM-based bit selection requires one additional block RAM (out of 232, i.e., 69% of the total capacity) and consumes 19546 LUTs (out of 47232, i.e. 41% of the total capacity). Since both kinds of designs are limited by the operating frequency, trie-based hashing only has an area overhead of 4.5% on average (Table 4). Hence the overall overhead costs of our proposed conflict detection scheme are low and enable significant throughput improvements.

6 Conclusions

In this paper we have studied several previously-proposed signature-based conflict detection schemes for TM. Among those, we found that bit selection provides the best implementation that avoids (i) degrading the operating frequency of an FPGA-based soft multiprocessor system or (ii) stalling the processors for multiple cycles. We have presented a method for implementing more efficient signatures by customizing them to match the access patterns of an application. Our scheme builds on trie-based hashing, and minimizes the number of false conflicts detected, improving the ability of the system to exploit parallelism. On a real FPGA-based packet processor, we measured packet throughput improvements of 12%, 58%, 9% and 71% for four applications, demonstrating that application-specific signatures are a compelling means to facilitate conflict detection for FPGA-based TM systems.

References

1. Wee, S., Casper, J., Njoroge, N., Tesylar, Y., Ge, D., Kozyrakis, C., Olukotun, K.: A practical FPGA-based framework for novel CMP research. In: Proc. of FPGA 2007, pp. 116–125 (2007)
2. Grinberg, S., Weiss, S.: Investigation of transactional memory using FPGAs. In: Proc. of EEEI 2006, pp. 119–122 (November 2006)
3. Kachris, C., Kulkarni, C.: Configurable transactional memory. In: Proc. of FCCM 2007, pp. 65–72. IEEE Computer Society, Los Alamitos (2007)
4. Ceze, L., Tuck, J., Torrellas, J., Cascaval, C.: Bulk disambiguation of speculative threads in multiprocessors. In: Proc. of ISCA 2006, pp. 227–238 (2006)
5. Sanchez, D., Yen, L., Hill, M.D., Sankaralingam, K.: Implementing signatures for transactional memory. In: Proc. of MICRO 2007, pp. 123–133 (2007)
6. Lockwood, J.W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., Luo, J.: NetFPGA - an open platform for gigabit-rate network switching and routing. In: Proc. of MSE 2007, June 3-4 (2007)
7. Yen, L., Bobba, J., Marty, M.R., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: LogTM-SE: Decoupling hardware transactional memory from caches. In: Proc. of HPCA 2007, pp. 261–272 (2007)
8. Yen, L., Draper, S., Hill, M.: Notary: Hardware techniques to enhance signatures. In: Proc. of Micro 2008, pp. 234–245 (November 2008)
9. Quislant, R., Gutierrez, E., Plata, O.: Improving signatures by locality exploitation for transactional memory. In: Proc. of PACT 2009, pp. 303–312 (2009)
10. Otoo, E.J., Effah, S.: Red-black trie hashing. Carleton University, Tech. Rep. TR-95-03 (1995)
11. Khuller, S., Moss, A., Naor, J.S.: The budgeted maximum coverage problem. Inf. Process. Lett. 70(1), 39–45 (1999)
12. Labrecque, M., Yiannacouras, P., Steffan, J.G.: Custom code generation for soft processors. In: Proc. of RAAW 2006 (December 2006)
13. Veenstra, J., Fowler, R.: MINT: a front end for efficient simulation of shared-memory multiprocessors. In: Proc. of MASCOTS 1994, pp. 201–207 (January 1994)
14. Labrecque, M., Steffan, J.G.: Fast critical sections via thread scheduling for FPGA-based multithreaded processors. In: Proc. of FPL 2009 (September 2009)
15. Teodorescu, R., Torrellas, J.: Prototyping architectural support for program rollback using FPGAs. In: Proc. of FCCM 2005, pp. 23–32 (April 2005)
16. Aasaraai, K., Moshovos, A.: Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming. In: Proc. of FPL (2009)
17. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: Proc. of IISWC 2008 (September 2008)

Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems

Christopher Claus*, Rehan Ahmed, Florian Altenried, and Walter Stechele

Institute for integrated Systems, Technische Universität München,
Arcisstr. 21, 80290 München, Germany
{Christopher.Claus,Walter.Stechele}@tum.de,
{Rehan.Ahmed,Florian.Altenried}@mytum.de

Abstract. Using dynamically reconfigurable hardware is useful especially when a high degree of flexibility is demanded and the application requires inherent parallelism to achieve real-time constraints. Depending on various driving conditions different algorithms have to be used for video processing. These different algorithms require different hardware accelerator engines, which are loaded into the AutoVision chip at runtime of the system. The novelties presented in this paper are the determination of the maximum frequency for dynamic partial reconfiguration of Xilinx Virtex-II Pro, Virtex-4 and Virtex-5 devices and a modified overclocked version of the ICAP controller. In addition an online verification approach is presented that can determine configuration errors that might be caused by configuring a device above the specified frequencies. This results in a reconfiguration throughput which is three times higher than the maximum throughput specified by Xilinx.

Keywords: Fast Dynamic Partial reconfiguration, FPGA, driver assistance, image processing.

1 Introduction

The work presented in this paper can be applied to any video and image processing application, where different accelerators for mutually exclusive situations are demanded. Video-based driver assistance has been chosen as one of several application areas as it requires real-time processing of different complex algorithms in diverse situations. A pure software implementation does not offer the required real-time processing, based on available hardware in automotive environments. Therefore hardware acceleration is necessary. Dedicated hardware circuits such as application-specific integrated circuits (ASICs) or off-the shelf application specific standard products (ASSPs) can offer the required real-time processing, but they do not offer the necessary flexibility. In addition, the design times for ASICs

* This work was supported by the German Research Foundation DFG (Deutsche Forschungsgemeinschaft) in the focus program No. SPP1148. Exceedingly the authors want to thank Xilinx for providing development boards and BMW for providing test video sequences for the research activities.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 55–67, 2010.

and their development costs have risen over the past years. As video algorithms for driver assistance are not standardized, design changes are quite frequent, which rarely makes an ASIC a suitable choice. Thus a flexible, programmable, situation adaptive hardware acceleration is required. Specific driving conditions (such as highway, country side, urban traffic, tunnel) require specific optimized algorithms. Using an ASIC or ASSP the chip area for all the functionality has to be allocated during design time, as changes after manufacturing are not possible any more. One can imagine that a large part of the functionality is unused most of the time. In contrast, reconfigurable hardware (FPGAs) and Graphic Processor Units (GPUs) offer high potential for real-time video processing. Although NVIDIA presented their low power GeForce G102M GPU, those devices are still considered very power hungry, especially when compared to state of the art Spartan6 devices. In addition a CPU has to be involved to transfer the data from the main memory to the local memory of the GPU, which in turn increases the overall monetary costs of the complete system. A performance comparison (pixels processed per clock cycle) between an algorithm implemented on a GPU and an FPGA can be found in [1].

In the AutoVision architecture [2], only the performance intense parts of the image processing algorithms are accelerated by coprocessor engines. The remainder of the algorithm, the so-called high level application code, is implemented fully in software on an embedded CPU core, such as the PowerPC (PPC) (available on various Virtex devices) or Microblaze, allowing it to remain easily updateable and to provide flexibility for new algorithms. In addition, the application code running on the embedded CPU is able to trigger a reconfiguration process. Hence the coprocessors available on the system can be exchanged during runtime, which allows a much larger set of hardware accelerated functionality than would normally fit onto a device. This process makes use of the dynamic partial reconfiguration (DPR) capabilities of Xilinx FPGAs. If a coprocessor is swapped the remainder of the video processing system, containing for example the video-input and video-output, remains fully operational. This extends the general idea of a system-on-chip to a situation adaptive integrated circuit. The AutoVision architecture is implemented on Xilinx Virtex devices as a proof-of-concept study. Target architecture will be a Spartan device, which is cost competitive in the high volume market.

1.1 State of the Art and Related Work

The idea of using hardware acceleration as a solution for computationally intensive applications for real-time visual recognition has been developed many years ago. Since then, companies have begun manufacturing System on Chips (SoCs) to be used in automotive or robotic environments. One of them is Mobileye [3], who presented their EyeQ chip with two ARM cores and 4 dedicated coprocessors for object classification, tracking, lane recognition and filter applications. To reduce the lack of flexibility, Mobileye introduced three programmable Vector Microcode Processors in their second generation of the EyeQ chip (EyeQ2). A comprehensive collection of recent vision-based on-road vehicle detection systems can be found in [4].

Unfortunately the problem that unused coprocessors persist on the device while they are not needed, thereby occupying a lot of resources, remains. FPGAs can be used to cope with that problem by updating their configuration information whenever needed. One major aim is to keep the reconfiguration overhead as low as possible (especially the reconfiguration time). Thus in this paper an approach is presented to speed up the reconfiguration on various Xilinx Virtex devices in a way that no video frame has to be dropped.

Various authors addressed the problem of fast DPR, to be used in the signal and image processing domain. Manet et. al. [5] present an evaluation of DPR for real signal and image processing applications. Although some other applications are mentioned, the focus is on applying DPR for Software Defined Radio (SDR). For fast reconfiguration in their system a controller for the Internal Configuration Access Port (ICAP), which allows read and write access to the configuration data, has been implemented on Virtex-4 (V4) devices. This implementation achieves a throughput of 350 MB/s at a frequency of 100 MHz. This is close to the theoretical maximum of 400 MB/s at 100 MHz on V4, as the input data width of the ICAP is 4 bytes.

Another project that requires fast DPR is reported in [6]. Shelburne et. al. present the Metawire approach that uses fast DPR to emulate a Network-on-Chip (NoC). As especially the NoC router nodes consume a lot of resources on an FPGA, the Metawire architecture uses the configuration circuitry as a relatively high-performance NoC. The configuration information of a Blockram (BRAM) is read by the ICAP and written to another BRAM afterwards. To accelerate this process an overclocked V4 ICAP is used, which is capable of providing a bandwidth in excess of 200 MB/s at a frequency of 144 MHz.

In [7] Bomel et. al present the fast downloading of partial bitstreams for DPR via a standard ethernet framework on a Virtex-II Pro (V2P) device. Similar to the implementation presented in [5], Bomel et. al. use an ICAP controller attached to the On-chip Peripheral Bus (OPB). Their measurements show an obtained ICAP throughput of up to 50 MB/s (400 Kbit/ms).

A comprehensive survey of run-time reconfiguration speeds by utilizing different ICAP controllers is presented in [8]. Miu et.al. have achieved an average reconfiguration speed of 332.1 MB/s and a maximum reconfiguration speed of 371.4 MB/s at a cost of a huge amount of 32 Block RAMs in their controller. One drawback of the presented approach is that only partial bitstreams up to a size of 64 KB can be used for the reconfiguration. This is due to the fact that the 32 Block RAMs ($512 \times 32bit \times 32BRAMs = 64KB$) are used to hold the complete partial bitstream.

This paper is organized in the following manner: In Section 2 a scenario is described where DPR is beneficial. Section 3 gives a brief overview on the AutoVision architecture. Requirements and implementation aspects of fast DPR are mentioned in Section 4. In Section 5 results and a comparative summary of the ICAP throughputs mentioned in the literature are presented. Finally the paper is concluded with an outlook on future activities in Section 6.

2 Typical Scenario and Hardware Accelerators

In general, a large number of different coprocessors can be implemented on a System-on-Chip (SoC) in parallel. However, this is not resource efficient, as depending on the application, only a subset of those coprocessors will be active at the same time. In this section, a typical scenario is presented which shows that it is beneficial to adapt the hardware to a changing environment. The hardware accelerators used for this scenario were implemented for a proof-of-concept study.

With hardware reconfiguration, hardware resources can be used more efficiently. Coprocessor configurations can be loaded into an FPGA whenever needed, depending on the application. This especially makes sense when the driving situations are mutually exclusive (day-time/night-time driving, forward/backward driving, highway/urban environments).

The following typical driving scenario is considered: A car is driving at daytime on a highway, where other cars can be detected by feature points on their silhouette or shape. Then the car is approaching a tunnel. Here it is meaningless to search for feature points as it is more important for the driver to see what is inside the tunnel. Thus an algorithm for contrast enhancement on the dark tunnel entrance is desirable. Inside the tunnel, due to the low luminance level, only the lights of other vehicles are promising features and have to be distinguished from tunnel lights. Therefore another algorithm is used. When the car is leaving the highway and enters an urban environment, a detection of pedestrians according to their motion seems beneficial. This scenario could be arbitrarily extended by other situations such as changing weather conditions (rain, fog, snow etc.). In the AutoVision project this specific typical scenario is used as a proof-of-concept.

The extraction of feature points on the shape or silhouette of a car for instance is done by a hardware accelerator called the ShapeEngine. The contrast enhancement near gloomy tunnel entrances is done by the ContrastEngine. The pixel-level processing inside the tunnel can be accelerated by a coprocessor, called the TaillightEngine. Finally in urban environments, Optical Flow, which is calculated using two separate accelerators, can be used to detect moving objects, such as pedestrians or cyclists. A detailed description about the organization and performance of the ShapeEngine and the OpticalFlow can be found in [1] and [9] respectively.

3 AutoVision Architecture

For the image processing in the situations described in Section 2, several different hardware accelerators for pixel processing are required. These so called Engines are attached as bus masters to the Processor Local Bus (PLB). This design allows for direct memory access (DMA) of the Engines without involving the PowerPC (PPC) core in the pixel transfer, which leads to a great offload of the CPU. Simultaneous read and write transfers are supported by the two separate read and write data buses of the PLB, each 64-bit wide. The AutoVision system has

been implemented on a Xilinx V2P (Xilinx XUP board) and a V5 device (ML507 board). The incoming pixel data is buffered in on-chip RAM resources (BRAMs) inside the Video-Input core. When enough data is buffered for a burst transfer, the Video-Input core transfers the data into the DDR SDRAM via a Multiport Memory controller (MPMC), which acts as frame buffer. The DMA-capability of the Video-Input core, and using bursts instead of single-pixel-transfers, leads to a reduced PLB utilization. The same applies to the Video-Output core which is used to fetch processed data and send it to the output pins of the FPGA e.g. for displaying it on an external monitor. Two SRAMs are used for double buffering. Between processing by the Video-Input and Video-Output, the Engines can operate on the pixel data. A block diagram of the complete architecture including the major components is shown in Figure 1.

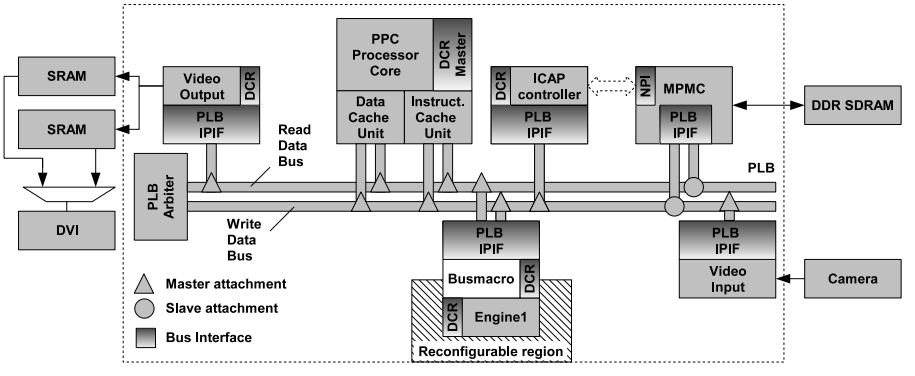


Fig. 1. Simplified block diagram of the AutoVision architecture

4 Fast Dynamic Partial Reconfiguration

Dynamic partial reconfiguration (DPR) is the ability to reconfigure a certain portion (partial) of the device during run-time (dynamic). Currently this feature is only offered by Xilinx Virtex and Spartan devices. The approaches for higher configuration data throughput and interconnect optimizations are generally applicable. A requirement for future video-based driver assistance, especially when used for safety critical applications, is that video frames must not be dropped. This requirement again leads to the fact that the reconfiguration of an Engine has to be as fast as possible, or at least so fast that the real-time requirement (processing of at least 25 fps = 40 ms per frame) is not violated. Depending on when and how often the system has to be reconfigured, the process is defined as *Inter Video Frame Reconfiguration* or *Intra Video Frame Reconfiguration*.

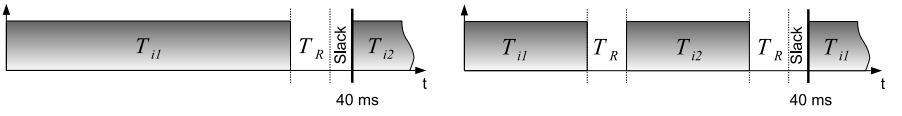
4.1 Inter Video Frame Reconfiguration

The Inter Video Frame Reconfiguration (InterVFR) is defined as swapping reconfigurable modules *between* two consecutive video frames. In Figure 2(a), the

time to process an image with one engine is denoted as T_{i1}. The processing time required by another engine is denoted as T_{i2}. The reconfiguration time for swapping is denoted as T_R, and includes clearing the reconfigurable region with a blank bitstream and loading the new module by using a second partial bitstream. If $T_{i1} + T_R < 40ms$ (25 fps), then InterVFR is possible, which means that processing an image with an engine and reconfiguring the device can be done before T_{i2} starts.

4.2 Intra Video Frame Reconfiguration

The Intra Video Frame Reconfiguration (IntraVFR) is defined as swapping multiple reconfigurable modules *within* one video frame. The hardware accelerator for the Optical Flow [9] can serve as an example here. It consists of two engines, namely the CensusEngine and the MatchingEngine, which are executed sequentially. The time to process an image with the CensusEngine is denoted as T_{i1}. The processing time required by the MatchingEngine is denoted as T_{i2}. As the size of the partial bitstreams for the CensusEngine and the MatchingEngine is considered to be the same the reconfiguration time for both Engines is equal and denoted as T_R (blank and module bitstream)in Figure 2(b). If $T_{i1} + T_{i2} + 2 \times T_R < 40ms$ (25 fps), then IntraVFR is possible.



(a) interVFR (b) intraVFR

Fig. 2. Inter and Intra Video Frame Reconfiguration

4.3 Modified ICAP Controller

To utilize the *ICAP* a controller has been implemented to achieve a throughput rate close to the theoretical maximum. This controller, presented in [10], consists of some logic to load the bitstream data, which is stored in an external memory, and provide the maximum amount of data per clock cycle to the ICAP. The *ICAP Input Width (IIW)* is 8 bit in V2P and 32 bit in V4 and V5 devices, which results in a maximum throughput of 100MB/s for V2P and 400MB/s for V4 and V5 if the ICAP is clocked at 100 MHz. The maximum frequencies specified by Xilinx are 50 MHz for V2P and 100 MHz for V4 and V5. In [10], a frequency of 100 MHz was used to send data to the ICAP. Higher frequencies and thus overclocking of the ICAP are possible by using a simple handshaking protocol. By indicting a *busy* status, the dedicated ICAP interface notifies the controller that it is currently not able to process incoming data. Compared to the implementation in [10], some additional optimizations, such as the usage of an asynchronous FIFO, have been made. The ICAP controller can now be easily connected to any V2P, V4 or V5 or Spartan device. The *IIW* as well as the *burst size (BS)* can be configured. Most of the approaches described in

literature use a FIFO as intermediate buffer inside the ICAP controller. In order to obtain a higher throughput and thus shorter reconfiguration times, many authors concentrate on techniques to keep the FIFO filled all the time. The *busy factor (BF)* is used to indicate what percentage of the configuration time the ICAP is not able to process incoming data.

$$TP[\frac{byte}{s}] = f_r[Hz] * IIW[byte] * BF[0,1] \tag{1}$$

In literature only Shelburne et. al. [6] use the ICAP on V4 beyond the specified 100 MHz. To safely pass data from one clock domain to another, asynchronous FIFOs are used. The frequency f_w used to write data to the asynchronous FIFO is usually that of the interconnect (PLB or MPMC) frequency. The frequency f_r used to read from the FIFO can be any frequency that can be generated with a Digital Clock Manager (DCM). The usage of those different clock domains subdivides the design into two different domains, namely the f_w and the f_r domain as depicted in Figure 3. Results of the throughput obtained by utilizing the ICAP controller on V2P, V4 and V5 devices with an asynchronous FIFO as described above, can be found in Section 5.

To achieve the maximum throughput one has to assure that the FIFO used for intermediate storage of the bitstream data remains filled at all times. This is the case when Equation 2is fulfilled.

$$\frac{BS[cycles] * DW[bit]}{BS[cycles] + L[cycles]} * f_w > IIW[bit] * f_r \tag{2}$$

The left side of Equation (2) determines how many bits per second can be written into the FIFO, which of course is dependent on the frequency f_w with which the FIFO is filled. The right side of the equation determines how many bits per second can be read from the FIFO and provided to the ICAP.

To keep the FIFO filled at all times high speed interconnects are necessary. Therefore it is possible to connect the ICAP controller either to a PLB Bus or to an MPMC via custom interfaces without any modification. On the V4 and V5 devices the ICAP controller is connected directly to a port of the MPMC which is indicated by the dashed arrow in Figure 1 and Figure 3. The number of bytes that can be written to the *ICAP* per clock cycle is dependent on the *IIW*. As long as the asynchronous FIFO is full, *IIW* bits can be written into the ICAP per clock cycle. The *data width (DW)* determines the input and output width of the asynchronous FIFO and is dependent on the incoming data. If a 64-bit wide PLB is used to connect the memory and the ICAP controller, the *DW* of the ICAP controller is set to 64 bits. *BS* determines how many words with a size of *DW* each are transferred within a burst. The *memory access latency (L)* is used to specify the number of cycles from the point in time the data is requested until the first word appears at the input of the ICAP controller. This value is strongly dependent on the implementation of the memory controller. Once it has been assured that the FIFO remains full during the whole configuration process, the throughput *TP* can be calculated using Equation 1. In all of the test designs

on V2P, V4 and V5, the FIFO remained filled at all times, which means that Equation 2 was always fulfilled.

4.4 Bitstream Verification

The fast reconfiguration, especially when overclocking the ICAP at non specified frequencies, imposes an additional task on such a system to verify the reconfiguration process and further verify the functionality of the reconfigurable modules. At higher reconfiguration speed the configuration packet processor *CPP* inside the ICAP may not write correctly into the configuration memory or the bitstream itself is corrupted when being transferred from the memory to the ICAP.

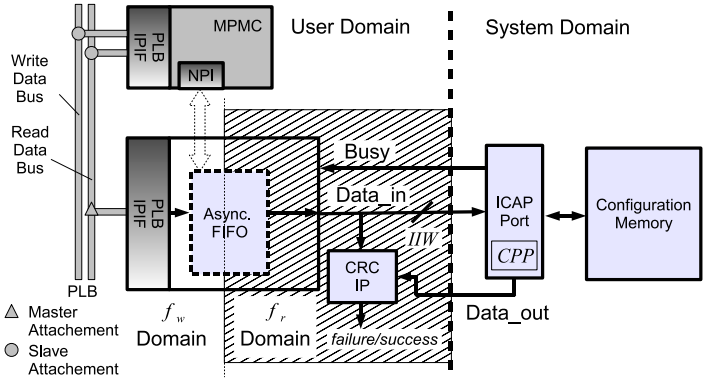


Fig. 3. System domains in a reconfigurable system

This breaks the problem of online verification into two main domains, namely the *user domain* and the *system domain*. The *user domain* contains all the configurable resources (memory, bus system, ICAP controller etc.) and ends before the ICAP interface as can be seen in Figure 3. The *system domain* starts from the ICAP interface and ends at the configuration memory. It contains the *CPP* (not accessible by the user) which is responsible for writing into the configuration memory. Both of these domains have to be verified during the online verification of reconfigurable modules. The *user domain* verification will ensure the integrity of the bitstream before it is fed into the ICAP. The verification of the *system domain* will ensure that the reconfigurable module has been uploaded successfully into the desired location in the configuration memory. By merging the verification information from both domains, information about the reconfiguration process at run-time can be generated, indicating the *success* or *failure* of the reconfiguration process.

In this approach a CRC IP module has been added in the *user domain*, right before the ICAP interface. The module calculates the CRC-16 value for the data that is fed into the ICAP. If the bitstream is corrupted during the transfer from the memory to the ICAP, even if a single bit is flipped, it is indicated by this

module and the configuration can be stopped. The signal indicating a configuration error in the *system domain* (which is combined with the error signal of the *user domain*) is the logical OR of several internal error flags including another CRC error check inside the ICAP. If a bit flip occurs when data is sampled by the ICAP, it will be detected. The ICAP produces distinct words at the output of the ICAP on V2P, V4 and V5 devices. The information from the *user domain* (CRC) and *system domain* (ICAP output) is merged together to indicate a configuration *success* or *failure* signal for the system. As the verification in the *user domain* is performed online, i.e. in parallel to the configuration process itself, it does not reduce the throughput. To analyze which actions have to be taken if a configuration error has been detected is part of future research. Up to now all reconfigurations at the frequencies mentioned in Section 5 were successful if the partial bitstreams are not manipulated manually.

5 Results

In [10] it has been shown that the maximum throughput for V2P has been achieved at 100 MHz. In order to achieve the same for V4 and V5 devices, the transfer from memory to ICAP has been optimized. Various test to obtain the maximum throughput by connecting the ICAP controller either to the PLB or directly to the MPMC have been performed. The maximum throughput specified by Xilinx (400 MB/s) for V4 and V5 devices at a frequency of 100 MHz is only possible by connecting the ICAP directly to the MPMC and by using address pipelining. As reported in Section 4.3, beside optimizing the transfer from the memory to the ICAP, the frequency f_r used to read from the asynchronous FIFO and provide the data to the ICAP can be increased.

As the ICAP on V2P is only specified up to 50 MHz without using the *busy* signal, but no upper boundary is provided when *busy* is used, the maximum frequency for ICAP on V2P has to be determined. Table 1 shows the result of an ICAP controller tested on 15 different XUPV2P boards from Xilinx. The bitstreams on this board are stored in a DDR SDRAM in order to have fast access to this data. In all designs the same size for the reconfigurable region was used, which results in the same bitstream size of 73600 bytes for all the designs. A hardware counter was used to measure the number of cycles for each reconfiguration process. The busy factor BF has been determined by averaging several measurements. Up to a frequency f_r of 150 MHz, the reconfiguration on all platforms was successful. Due to production tolerances some of the tested devices achieved a frequency of up to 170 MHz. This has been verified by the approach presented in Section 4.4 and also by reading back the configuration data via JTAG and comparing it against the initial bitstream. The latter method is not real-time capable as it takes too long and thus cannot be used in the AutoVision system. The test results which have been performed to determine the maximum frequency on V2P are depicted in Table 1. The cycle count differs between the measurements because the busy signal is asserted more or less often. Note that the busy signal is not proportional to f_r.

Table 1. Performance measurements on an XUPV2P board

f_w $[MHz]$	f_r $[MHz]$	cycle count	T_R $[\mu s]$	BF	T_P $[MB/s]$
100	80	80800	1010.00	0.911	72.87
100	90	81462	905.13	0.903	81.31
100	100	80871	808.71	0.910	91.01
100	110	81054	736.85	0.908	99.88
100	120	81222	676.85	0.906	108.74
100	130	81692	628.40	0.901	117.12
100	140	81935	585.25	0.898	125.75
100	150	80703	538.02	0.912	136.80

At a frequency of 150 MHz a throughput T_P of 136.8 MB/s has been achieved. The same tests were also performed on V4 and V5 devices. Measurements on V4 have shown that reconfiguration at frequencies of 140 MHz is possible. On V5 several measurements have shown that a throughput of 1200 MB/s is possible, which has been verified by means of a hardware demonstrator. This throughput is 3 times higher than the throughput that can be achieved with the frequency specified by Xilinx (400MB/s), and more than 3 times higher than every throughput that has been reported in literature so far. Without such a high throughput the realization of IntraVFR is unlikely to succeed. In Table 2 the ICAP throughput reported by different authors is depicted and compared with the results obtained with the presented approach.

Table 2. Throughput of ICAP controller on XUPV2P, ML405 and ML507 board

Source	Device Type	IIW [bit]	f_r [MHz]	inter-connect	M. T_P [MB/s]	T. T_P [MB/s]
[5]	V4	32	100	OPB	350	400
[6]	V4	32	144	cust. Link	219.31	576
[7]	V2	8	100	ethernet	50	90-100
[8]	V4	32	100	PLB	371.4	400
authors	V2	8	100	PLB	93.94	90-100
authors	V2	8	150	PLB	136.8	140
authors	V4	32	100	MPMC	400	400
authors	V4	32	140	MPMC	560	560
authors	V5	32	100	MPMC	400	400
authors	V5	32	200	MPMC	800	800
authors	V5	32	300	MPMC	1200	1200

As can be seen in Table 2, the theoretical throughput (T. T_P) is equal to the measured throughput (M. T_P) for frequencies of 100 MHz and 140 MHz, which means that the maximum throughput for these frequencies has been achieved on V2P, V4 and V5. As already mentioned, on V2P 150 Mhz is considered to be the maximum frequency that can be used to reconfigure the device safely. The throughput of the V2P device is strongly dependent on the *busy* signal of the

ICAP. The measurements indicate a maximum value around 91 KB/ms due to the *busy* status of the ICAP. The theoretical throughput at 100 MHz (which is 100 KB/ms) can never be achieved due to delays internal to the ICAP. Thus the measured throughput is considered as the maximum achievable, since every clock cycle that the ICAP is not busy, the maximum amount of data possible is provided to the ICAP. None of the authors in [5], [6] or [7] mention why the maximum is not achieved, but it is most likely due to problems with transferring bitstream data to the ICAP. The busy signal has neither been appeared on V4 nor on V5 at frequencies between 140 MHz and 300 MHz.

In Table 3 the resources utilized by ICAP controller including the verification logic are depicted in the third column. The resources are compared against the results from [8] (second column), as this implementation has achieved the closest results in terms of throughput. Depending on which interface is chosen, the resource utilization of the PLB IPIF (fourth column) and the direct connection to the MPMC (fifth column) is shown.

Table 3. Resource utilization of ICAP designs and IP Interfaces on Virtex-4 FX20

Resources	ICAP of [8]	authors' ICAP	authors' PLB IPIF	authors' MPMC IF
4-LUT (total)	963 (5.6%)	354 (2.1%)	671 (3.9%)	29 (0.2%)
4-LUT used as logic	614 (3.6%)	354 (2.1%)	671 (3.9%)	29 (0.2%)
4-LUT used as shift registers	320 (1.9%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Slice Flip-Flops	469 (2.7%)	197 (1.2%)	347(2.0%)	7 (0.1%)
Block RAM (BRAM)	32 (47.1%)	2 (2.9%)	0 (0.0%)	0 (0.0%)

According to the results in Table 3 connecting the ICAP controller directly to the MPMC seems preferable. On the other hand adding another port to the MPMC will increase its overall resource utilization, especially the BRAMs.

Finally it can be shown that Inter as well as IntraVFR is possible by means of hardware demonstrators. In the demonstrator for the InterVFR, the reconfiguration is not noticeable as there is no frame dropped. However, it is possible to visualize the changes caused by the reconfiguration by utilizing the approach presented in [11]. In the demonstrator for IntraVFR, 124 reconfigurations are performed per second (rps). Swapping two modules within one frame, and clearing the reconfigurable region with blank bitstream in each case results in 4 reconfigurations per video frame. Given a frame rate of 31 fps results in the above mentioned ($31 \times 4 =$) 124 rps.

6 Conclusion and Outlook

In this paper it has been shown that fast DPR can be utilized in a real-time environment, such as video based driver assistance, without violating the real time constraints. This has been achieved through modifications of the ICAP controller and optimizations in memory transfer. The presented concept has been verified in demonstrators which show that Inter as well as IntraVFR is possible. The results obtained outperform all other approaches mentioned in literature in terms of reconfiguration speed and throughput. Additionally the correctness of the (re-)configuration is being assured through online verification.

With the demonstrators for InterVFR and IntraVFR (see Section 4.2) it can be shown that fast dynamic reconfiguration can be applied in real-time systems in order to save precious on-chip resources. The demonstrators show that at least two reconfigurations of sequentially working hardware accelerators within 40 ms are possible without the loss of video frames. In addition, as the ICAP is overclocked, the configuration data is verified during the configuration process. Finally, the maximum reconfiguration frequency will be determined for V4 and V5 devices. The demonstrator implemented on a V5 platform serves as a proof of concept. The target platform is an automotive qualified Xilinx Spartan device, which will consume less power, and is cost competitive in the high volume market. Additional measurements to determine if the reconfiguration will increase or decrease the power consumption compared to a non-reconfigurable system is part of future research.

References

1. Claus, C., Huitl, R., Rausch, J., Stechele, W.: Optimizing the SUSAN corner detection algorithm for a high speed FPGA implementation. In: Proceedings of FPL 2009, Prague, Czech Republic (2009)
2. Claus, C., Stechele, W., Herkersdorf, A.: AutoVision - a run-time reconfigurable MPSoC architecture for future driver assistance systems. It-Journal 49(3), 181–187 (2007)
3. `http://www.mobileye-vision.com`
4. Sun, Z., Bebis, G., Miller, R.: On-road vehicle detection: a review. IEEE Transactions on Pattern Analysis and Machine Intelligence 28(5), 694–711 (2006)
5. Manet, P., Maufroid, D., Tosi, L., Gailliard, G., Mulertt, O., Ciano, M.D., Legat, J.-D., Aulagnier, D., Gamrat, C., Liberati, R., Barba, V.L., Cuvelier, P., Rousseau, B., Gelineau, P.: An Evaluation of Dynamic Partial Reconfiguration for Signal and Image Processing in Professional Electronics Applications. EURASIP Journal on Embedded Systems 2008 (November 2008)
6. Shelburne, M., Patterson, C., Athanas, P., Jones, M., Martin, B., Fong, R.: Metawire: Using FPGA configuration circuitry to emulate a Network-on-Chip. In: Proceedings of FPL 2008, Heidelberg, Germany, pp. 257–262 (2008)
7. Bomel, P., Crenne, J., Ye, L., Diguet, J.-P., Gogniat, G.: Ultra-Fast Downloading of Partial Bitstreams through Ethernet. In: Sirisuk, P., et al. (eds.) ARC 2010. LNCS, vol. 5992, pp. 72–83. Springer, Heidelberg (2010)

8. Liu, M., Kuehn, W., Lu, Z., Jantsch, A.: Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration. In: Proceedings of FPL 2009, Prague, Czech Republic (2009)
9. Claus, C., Laika, A., Jia, L., Stechele, W.: High performance FPGA based optical flow calculation using the census transformation. In: Proceedings of IV 2009, Xi'an, China (2009)
10. Claus, C., Zhang, B., Stechele, W., Braun, L., Hübner, M., Becker, J.: A multi-platform controller allowing for maximum Dynamic Partial Reconfiguration throughput. In: Proceedings of FPL 2008, Heidelberg, Germany, pp. 535–538 (2008)
11. Hübner, M., Braun, L., Becker, J., Claus, C., Stechele, W.: Physical Configuration On-Line Visualization of Xilinx Virtex-II FPGAs. In: Proceedings of ISVLSI 2007, Porto Alegre, Brazil, pp. 41–46 (2007)

Parametric Encryption Hardware Design

Adrien Le Masle[1], Wayne Luk[1], Jared Eldredge[2], and Kris Carver[2]

[1] Department of Computing, Imperial College London, UK
{al1108,wl}@ic.ac.uk
[2] BlueRISC, Inc, Amherst, MA, USA
{jared,kris}@bluerisc.com

Abstract. We present new scalable hardware designs of modular multiplication, modular exponentiation and primality test. These operations are at the core of most public-key crypto-systems. All the modules are based on an original Montgomery modular multiplier. Our multiplier is the first Montgomery multiplier design with variable pipeline stages and variable serial replications. It is 8 times faster than the best existing hardware implementation and 30 times faster than an optimised software implementation on an Intel Core 2 Duo running at 2.8 GHz. Our exponentiator is 2.4 times faster than an optimised software implementation. It reaches the performance of a more complex FPGA design using DSP blocks which is the fastest in the literature. Our prime tester is 2.2 times faster than the software implementation and is 85 times faster than hardware implementations of the same algorithm with only 60% area overhead.

1 Introduction

Most public-key cryptographic algorithms consist of two main stages: the key generation which requires the ability to generate large prime numbers and the encryption/decryption part.

Modular exponentiation is a common operation used by several public-key crypto-systems, such as the Diffie-Hellman key exchange protocol and the Rivest, Shamir and Adleman (RSA) encryption scheme. It is also, together with modular multiplication, the core of common prime tests such as the Rabin-Miller strong pseudo-prime test.

As security is becoming increasingly important, algorithms such as RSA need more and more bits for the keys used to be secured. For data that need to be protected until 2030, a 2048 bit key is recommended whereas a 3072 bit key is recommended for beyond 2031 [2]. This creates a need for scalable designs working with any bit-width.

Many new algorithms and improvements of existing algorithms for modular multiplication have been presented during the last decade [10]. This led to many hardware implementations of modular multiplication [3,4,6,8,9,11], modular exponentiation [3,7,9,11,12,13], and primality testing [5]. Most implementations target Field Programmable Gate Arrays (FPGAs) which offer rapid-prototyping platforms to compare different designs and can be reprogrammed as needed.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 68–79, 2010.

As FPGAs are quickly increasing in size, it is becoming more of a challenge to fully cover the design space available for a given budget. This introduces the need for parametric designs capable of exploring the entire design space, especially in terms of the speed-area trade-off.

This paper presents parametric hardware designs of modular multiplication, modular exponentiation and primality testing. Our main contributions include:

- A new parametric Montgomery multiplier design with variable pipeline stages and variable serial replications
- A modular exponentiator design based on our Montgomery multiplier
- A Rabin-Miller prime tester design using both our multiplier and our exponentiator
- An implementation of the proposed designs on Xilinx Virtex-5 FPGAs and a comparison of their performance in terms of speed and area with the main existing implementations.

Our Montgomery multiplier implementation is 8 times faster than the best existing hardware implementation [13] and 30 times faster than an optimised software implementation on an Intel Core 2 Duo running at 2.8 GHz. Our exponentiator is 2.4 times faster than an optimised software implementation and reaches the performance of a more complex FPGA design using DSP blocks [12] which is also the fastest design in the literature. Our prime tester is 2.2 times faster than the software implementation and is 85 times faster than hardware implementations of the same algorithm [5] with only 60% area overhead.

The rest of the paper is organised as follows. Section 2 explains the background relevant to our work. In section 3, we present the main challenges of our designs and how we address them. In section 4, we compare our FPGA implementations to the best existing implementations and highlight the scalability of our hardware architectures. Finally, section 5 concludes the paper.

2 Background

Most modular exponentiation algorithms require the ability to perform fast and area efficient modular multiplications. In [4], different algorithms for modular multiplication are compared in terms of the area-time product (AT). The Montgomery modular multiplication algorithm turns out to be the best with an AT complexity of $O(n^2)$.

Another important feature of a crypto-system is the ability to generate large prime numbers. Probabilistic methods for prime testing, determining whether or not a number is prime with a certain probability of error, are often used.

Modular Exponentiation. A simple but common algorithm for modular exponentiation is given in Alg. 1. To compute $X^E \bmod N$, the algorithm iterates on the bits of E from the least significant bit (LSB) to the most significant bit (MSB). At each iteration i, the variable $P_i = X^{2^i} \bmod N$ is squared modulo N to obtain $P_{i+1} = X^{2^{i+1}} \bmod N$. If $e_i = 1$, the accumulated product Z_i is multiplied by P_i modulo N, otherwise it remains the same. After n iterations, n being the bit-width of E, Z_n contains $X^E \bmod N$.

Algorithm 1. Exponentiation algorithm

Input: X,E,N with $E = \sum_{i=0}^{n-1} e_i 2^i$, $e_i \in \{0, 1\}$
Output: $Z_n = X^E \bmod N$

```
1 Z_0 = 1, P_0 = X
2 for i = 0 to n − 1 do
3     P_{i+1} = P_i^2 mod N
4     if e_i = 1 then
5         Z_{i+1} = Z_i.P_i mod N
6     else
7         Z_{i+1} = Z_i
8 end
```

Algorithm 2. Simple Montgomery algorithm for modular multiplication

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $(a_i, b_i, n_i) \in \{0, 1\}^3$, $n_0 = 1$, $0 \leq A, B \leq N$
Output: $P = A.B.2^{-n} \bmod N$

```
1 P = 0
2 for i = 0 to n − 1 do
3     P = P + a_i.B
4     P = P + p_0.N
5     P = P div 2
6 end
7 if P ≥ N then  P = P − N
```

Algorithm 3. Rabin-Miller strong pseudo-prime test

Input: $p = 2^r d + 1$ odd integer, set P of $|P|$ first primes
Output: *composite* if p is composite, *prime* if p is probably prime

```
1 for i = 0 to |P| − 1 do
2     a = P[i]
3     if a^d = 1 mod p or a^{2^j d} = −1 mod p for some 0 ≤ j ≤ r − 1 then
4         continue
5     else
6         return composite
7 end
8 return prime
```

Montgomery Modular Multiplication. A simple version of Montgomery modular multiplication is presented in Alg. 2. This algorithm iterates on the bits of A from the LSB to the MSB. At iteration i, $a_i.B$ is added to the accumulated product P. If P is odd, N is added to P. This does not change the result as the calculation is done modulo N. As N is odd (required by the algorithm), P becomes even and can be divided by 2 without remainder.

The drawback of the Montgomery algorithm is that it actually computes $A.B.2^{-n} \bmod N$, introducing an extra 2^{-n} factor which has to be eliminated. The common method to remove this factor is to convert the inputs in N-residue [7], to perform the modular multiplication and to convert back the result to a normal representation by Montgomery multiplying it by one.

Rabin-Miller Primality Test. In this paper, we consider a variant of the Rabin-Miller strong pseudo-prime test using a few number of primes as witnesses. This probabilistic test has a low probability of error. Algorithm 3 shows that the Rabin-Miller test relies on the ability to perform modular multiplications and exponentiations and is therefore a relevant application for our designs.

3 Design Flow

Our goal is to create parametric designs of Montgomery multiplication, Montgomery exponentiation and Rabin-Miller primality test. The benefits of such designs are that they can:

- adapt to the present and future needs in terms of key width
- explore a large design space in terms of speed and area, adapting to the accelerating advancement in FPGA development
- meet the requirements of various hardware encryption projects

Parametric Montgomery Multiplier Design. We choose a one-carry save adder (CSA) based Montgomery multiplier as the basic block of our design. A CSA is faster than a ripple-carry adder with no area overhead. Algorithm 4 from [4] is used. The diagram of a multiplier cell is given in Fig. 1. The bit-width of this cell is set as a design parameter.

We apply two techniques to our design: pipelining and serial replication. Pipelining improves the throughput of the design and serial replication improves

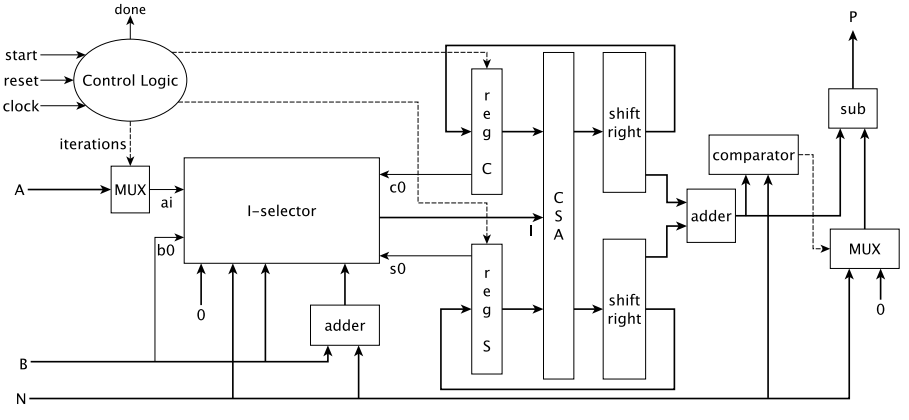


Fig. 1. Diagram of a Montgomery multiplier cell

Algorithm 4. Fast Montgomery algorithm for modular multiplication

Input: $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, $N = \sum_{i=0}^{n-1} n_i 2^i$, $(a_i, b_i, n_i) \in \{0,1\}^3$
Output: $P = A.B.2^{-n} \bmod N$

```
1  S = 0
2  C = 0
3  for i = 0 to n − 1 do
4      if (s0 = c0) and ai = 0 then  I = 0
5      if (s0 ≠ c0) and ai = 0 then  I = N
6      if (s0 ⊕ c0 ⊕ b0) = 0 and ai = 1 then  I = B
7      if (s0 ⊕ c0 ⊕ b0) = 1 and ai = 1 then  I = B + N
8      S, C = S + C + I
9      S = S div 2
10     C = C div 2
11 end
12 P = S + C
13 if P ≥ N then  P = P − N
```

its latency. Three main challenges have to be addressed to design a parametric Montgomery multiplier using these techniques:

Challenge 1. Algorithm 4 cannot be easily parallelised due to the data dependencies between the consecutive values of S and C in the main loop. At iteration $i+1$, the values of S and C from iteration i are needed to compute the new value of I and the new values of S and C.

Challenge 2. To explore as much design space as possible, the bit-width, the number of replications and the number of pipeline stages should be parameters which can take any value.

Challenge 3. The control should adapt to the values of these parameters.

Consider Challenge 1. Figure 2 shows the basic structure of our pipelined design. Each Montgomery cell is a modified version of the basic cell presented earlier. We cope with Challenge 1 by allowing each basic cell to perform a consecutive part of the iterations. For this principle to work, the final addition and subtraction blocks are removed from this cell and the number of iterations performed by each cell becomes a parameter. The basic cell is added with the ability to load the S and C registers from the inputs. The current values of S and C are also available at the output of each cell.

Inside each pipeline block, the CSA can be replicated as many times as needed. The data dependencies problem prevents us from simply duplicating the CSA and perform several iterations in parallel. Instead, several CSAs along with the shift logic are put in series. This is equivalent to unrolling the loop of Alg. 4 $r-1$ times. The I-selector is also replicated as the values of I differ for each CSA and at each iteration. At equal frequencies, replication decreases the latency of the design by a factor of r, the total number of CSAs. The area overhead is less than r because only a part of the basic cell is replicated. In practice, replication

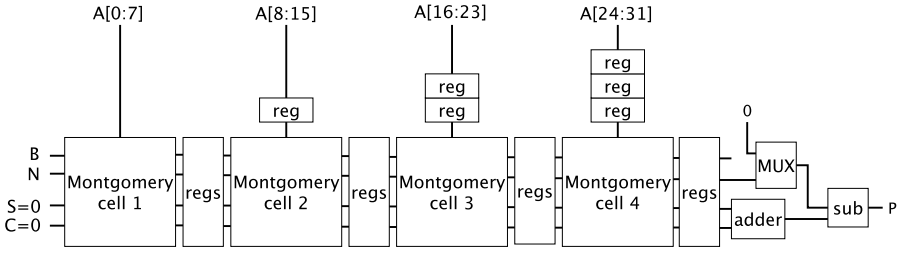


Fig. 2. Structure of a 32 bit pipelined Montgomery multiplier with 4 pipeline stages

also increases the critical path, reducing the maximum clock frequency at which the multiplier can run.

Consider Challenge 2. Allowing the bit-width (n) and the pipeline depth (p) to take any value makes it more difficult to divide the number of iterations between blocks. When n is not a multiple of p, each block cannot perform the same number of iterations. We address this challenge by adding an extra iteration to the first n mod p pipeline blocks. This leads to n mod p blocks computing $\lfloor n/p \rfloor + 1$ iterations, and $(n - n \bmod p)$ blocks computing $\lfloor n/p \rfloor$ iterations.

Inside a pipeline block, in order to allow the number of replications (r) to take any value, the result can be extracted from any CSA. This solution deals with the case when the number of iterations the cell has to perform is not a multiple of the number of replications.

Consider Challenge 3. A flexible pipeline control is implemented. This control deals with the updates of two types of registers: the registers between blocks and the triangular register array. The register triangular structure consists of arrays of registers controlled as FIFO queues. The inputs enter all the FIFOs at the same time when the `done` signal of the very first cell is triggered. The element at the head of the FIFO of a given cell leaves the queue when the corresponding cell has finished using it, that is when its `done` signal is triggered. In practice, extra registers store the position of the first empty slot in each FIFO, acting as pointers. When an element leaves the FIFO, the FIFO registers are updated accordingly (register i takes the value of register $i + 1$) and the corresponding pointer is decremented by 1. When an element enters the FIFO, the register indexed by the pointer is updated with the value of this element and the pointer incremented by 1.

Inside a pipeline block, the control logic manages the extraction of the result from the correct CSA, depending on the number of replications chosen and the number of iterations this particular block has to perform.

Let us consider an example summarizing this section with $n = 1024$, $p = 5$ and $r = 7$. As $\lfloor n/p \rfloor = 204$ and $n \bmod p = 4$, the first 4 pipeline blocks compute $204 + 1 = 205$ iterations and the last one computes 204 iterations. Our flexible pipeline control deals with this issue. Inside each block, we want 7 replications. In the first 4 pipeline blocks, we loop through the 7 CSAs 30 times ($\lceil 205/7 \rceil$). The result is extracted from CSA number 2 (205 mod 7) after the last iteration.

In the last pipeline block, we loop through the 7 CSAs 30 times ($\lceil 204/7 \rceil$). The result is extracted from CSA number 1 (204 mod 7) after the last iteration. This complex behaviour is managed by the control logic of each pipeline block.

Application to Modular Exponentiation. We use our modular multiplier to design a parametric hardware implementation of Alg. 1. The exponentiator uses the pipelining and replication capabilities of the multiplier description.

Two main problems have to be solved for this design to be efficient in terms of speed and area. First, the multiplier has to be optimally pipelined. We can show that the number of pipeline stages of the multiplier giving best performance for use with the exponentiator is equal to two. This is due to the fact that in Alg 1, P_{i+1} depends on P_i and Z_{i+1} depends on both P_i and Z_i. If we use more than two pipeline stages, the multiplier's pipeline cannot be kept full due to these data dependencies.

Second, integrating our multiplier in a bigger design can reduce its running frequency due to critical path problems. The latency of the adders and subtractors used in the multiplier would become a bottleneck for large bit-widths. To reduce the critical path, all the ripple-carry adders and the subtractors can be pipelined with any depth.

The design of our exponentiator is represented in Fig. 3. It has three main parameters: the number of multiplier's pipeline stages, the number of replications for the multiplier's pipeline cells, and the pipeline depth of the adders/subtractors. At each iteration, the current values of P and Z are stored in a RAM. The control logic manages the inputs to give to the multiplier and the data to write back. It also controls the multiplier.

Application to Primality Testing. We design a parametric Rabin-Miller prime tester based on both our multiplier and our exponentiator. The challenge is to use these two modules optimally, while keeping the design simple.

To save area, one single multiplier is shared by the exponentiator (to perform the multiplications needed to compute $a^d \bmod p$) and the prime tester (to perform the consecutive modular multiplications needed to compute the $a^{2^j d} \bmod p$). To improve the speed of the prime test, the exponentiator takes full advantage of the pipelining and replication features of the multiplier. However, it is not worth using the pipeline of the multiplier for the calculation of the $a^{2^j d} \bmod p$. It can be shown that the mean value of r in Alg. 3 is equal to two and that on average the multiplier is only used once directly by the prime tester at each iteration. Pipelining the computation of the $a^{2^j d} \bmod p$ would therefore make the design more complex with almost no performance benefit.

A diagram containing the important blocks, signals and connections of the prime tester is presented in Fig. 4. The values of the first prime numbers are stored in a ROM. At each iteration, the control logic selects the prime to use for the test and the inputs to give to the multiplier, to the comparator and to other intermediate registers. The control logic contains the state machine of the prime tester which controls the multiplier and the exponentiator.

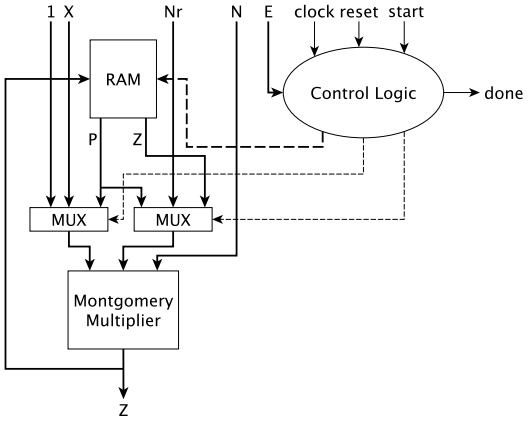


Fig. 3. Montgomery exponentiator

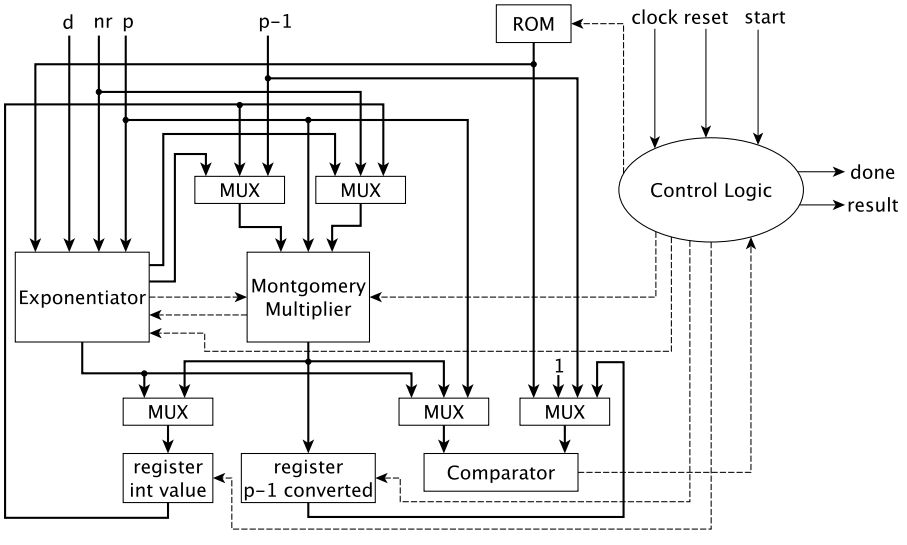


Fig. 4. Rabin-Miller prime tester

4 Results

Our three designs are implemented in Verilog. We synthesize our designs with Xilinx ISE 11.1 for Xilinx Virtex-5 FPGAs with "speed" as the optimisation mode and "normal" as the optimisation level. The results for the area and the maximum clock frequency are those given by the synthesis operation. When comparing the reported performance with other implementations, one should take into account that they do not all target the same FPGA. These results only give an idea of how our implementations perform compared to the best

implementations in the literature. However, the scalability results are made accurate by implementing our designs on the same FPGA for most values of r and p.

Multiplier. Table 1 compares the execution time of our multiplier with other implementations for $n = 1024$ bits. For the software version, we report the mean and the standard deviation (σ) of the execution time for one million multiplications of random numbers. The execution time of our hardware multiplier only depends on p, r and the clock frequency. Our multiplier without any pipelining and replication is faster than most existing implementations with a runtime of 4.28 μs. It is also faster than the software implementation of modular multiplication using the very optimised GMP library on an Intel Core 2 Duo E7400 running at 2.8 GHz. For $p = 8$ and $r = 8$, our multiplier is 8 times faster than the best existing hardware implementation and 30 times faster than the optimised software implementation. We can still get better performance by increasing r and p if we target an FPGA with enough available area. Our design scales as the device scales and can therefore adapt to future FPGA families.

The results of Tab. 1 also show how our multiplier scales with p and r. Keeping r constant, when p doubles (from 1 to 2), the execution time is halved with 85% area overhead. This area overhead is less than 100% as the area of the adder and subtractor required at the output of the pipeline is not negligible, especially for small values of p and r. Keeping p constant, by increasing r from 1 to 4 a speedup of 2.5 times is achieved with less than 80% area overhead. Increasing the number of replications is less area-consuming than increasing the number of pipeline stages as a smaller part of the basic cell is duplicated. However, the

Table 1. Performance comparison of 1024 bit multipliers

Design	Device	Clock (MHz)	Area (LUT-FF pairs)	Ex. Time (μs)
Our design ($p = 8$, $r = 8$)	XC5VLX330T-2	99.13	206 982	0.18
Our design ($p = 2$, $r = 8$)	XC5VLX110T-3	110.98	59 337	0.59
Our design ($p = 2$, $r = 4$)	XC5VLX110T-3	149.55	42 429	0.87
Our design ($p = 1$, $r = 8$)	XC5VLX110T-3	102.63	31 925	1.27
Tang [13]	XC2V3000-6	90.11	N/A	1.49
Our design ($p = 1$, $r = 4$)	XC5VLX110T-3	150.23	20 593	1.72
Our design ($p = 2$, $r = 1$)	XC5VLX110T-3	226.31	23 436	2.28
Our design ($p = 1$, $r = 1$)	XC5VLX110T-3	239.74	13 671	4.28
GMP 4.2.4 [1] `mpz_mul/mpz_mod`	Intel Core 2 Duo E7400	2800	N/A	mean: 5.45 σ: 0.53
Oksuzoglu [11] (1020 bit)	XC3S500E-4FG320C	119	6 906	7.62
McIvor [8]	XC2V3000	75.23	23 234	13.45
Daly [6]	XCV1000	55	10 116	18.67
Amanor [9]	XVC2000E-6	49	8 064	21.00

Table 2. Time-Area products normalised to our design for 1024 bit multiplication

Design	Area (LUT-FF pairs)	Clock Cycles	Time × Area
McIvor [8]	23 234	1025	6.08
Daly [6]	10 116	1027	2.65
Amanor [9]	8 064	1027	2.11
Oksuzoglu [11] (1020 bit)	6 906	907	1.60
Our design ($p = 2$, $r = 8$)	59 337	66	1.00

increase in speed is less substantial due to the negative effect of replication on the maximum frequency.

Table 2 compares implementations in terms of the Time × Area product. For $p = 2$ and $r = 8$ our multiplier ranks first. It is interesting to see that $(p, r) = (2, 8)$ is a design point favouring speed over area. Our design benefits mainly applications with high speed requirements and large area available.

Exponentiator. Table 3 compares the execution time of our exponentiator with other implementations for $n = 1024$ bits. The pipeline depth of all ripple-carry adders and all subtractors is fixed to 8 in order to reduce the critical path. For the software version, we also report the mean and standard deviation for one million exponentiations of random numbers. For $p = 2$ and $r = 8$, our implementation running at 97.9 MHz is 2.4 times faster than the optimised software implementation using the GMP library on an Intel Core 2 Duo E7400 running at 2.8 GHz. Our exponentiator can also reach the speed of the best Montgomery modular exponentiator in the literature which uses DSP operations.

Prime tester. Table 4 compares the execution time of our prime tester with other implementations for $n = 1024$ bits. The pipeline depth of all ripple-carry

Table 3. Performance comparison of 1024 bit exponentiators

Design	Device	Clock (MHz)	Area	Ex. Time (ms)
Suzuki [12]	XC4VFX12-10SF363	200/400	7 874 LUT-FF + 17 DSP48	1.71
Our design ($p = 2$, $r = 8$)	XC5VLX110T-3	97.9	65 200 LUT-FF	1.74
Tang [13]	XC2V3000-6	90.11	14 334 slices + 62 multipliers	2.33
Our design ($p = 2$, $r = 2$)	XC5VLX110T-3	145.66	28 008 LUT-FF	3.88
GMP 4.2.4 [1] `mpz_powm`	Intel Core 2 Duo E7400	2800	N/A	mean: 4.23 σ: 0.12
Oksuzoglu [11] (1020 bit)	XC3S500E-4FG320C	119	6 906 LUT-FF + 20 multipliers	7.95
Our design ($p = 1$, $r = 2$)	XC5VLX110T-3	135.9	17 414 LUT-FF	8.14
Blum [3]	XC4000	45.7	13 266 LUT-FF	11.95

Table 4. Performance comparison of 1024 bit prime testers

Design	Device	Clock (MHz)	Area (LUT-FF)	Ex. Time mean/σ (ms)
Ours ($p = 2$, $r = 8$)	XC5VLX110T-3	86.7	64 817	2.01/0.741
Ours ($p = 2$, $r = 4$)	XC5VLX110T-3	87.1	41 970	3.54/1.31
GMP 4.2.4 [1] `mpz_millerrabin`	Intel Core 2 Duo E7400	2800	N/A	4.33/1.82
Ours ($p = 1$, $r = 4$)	XC5VLX110T-3	87.1	31 538	6.81/2.51
Ours ($p = 2$, $r = 2$)	XC5VLX110T-3	87.0	30 892	6.64/2.45
Ours ($p = 1$, $r = 1$)	XC5VLX110T-3	87.1	22 346	25.3/9.32
Cheung [5] (non-scalable design)	XC3S2000	6.1	40 262	171.45/_
Cheung [5] (scalable design 32 PE)	XC3S2000	25.6	18 566	2235.08/_
Cheung [5] (scalable design 8 PE)	XC3S2000	26.5	5 684	6338.95/_

adders and all subtractors is also set to 8. Unlike our other designs, the execution time of the prime tester depends on the number under test. We choose 10 000 random numbers and use them for all the experiments. We report the mean and the standard deviation. For $p = 1$ and $r = 1$, our prime tester is 6.8 times faster than Cheung's non scalable design and takes 1.8 times less area. It is 88 times faster than the fastest scalable design from [5] with only 20% area overhead. For $p = 2$ and $r = 8$, our design running at 86.7 MHz is 2.2 times faster than the GMP implementation on an Intel Core 2 Duo running at 2.8 GHz. It is 85 times faster than Cheung's non scalable design and only takes 1.6 times more area.

Our multiplier, our exponentiator and prime tester descriptions cover a huge design space. The exponentiator and the prime tester scale with p and r the same way as the multiplier except from one point: their speed cannot be increased by using more than two multiplier's pipeline stages as shown before. However, once this threshold is reached, increasing the number of replications remains relevant in order to increase the speed of these two designs.

5 Conclusion and Future Work

This paper presents a new parametric Montgomery multiplier design with variable pipeline stages and variable serial replications. It is 8 times faster than the best existing hardware implementation and 30 times faster than an optimised software implementation on an Intel Core 2 Duo running at 2.8 GHz. We design a modular exponentiation module based on our multiplier. It is 2.4 times faster than the optimised software implementation and reaches the performance of a more complex FPGA implementation using DSP blocks. A Rabin-Miller prime tester gathering the strengths of our modular multiplication and exponentiation modules is presented. It is 2.2 times faster than the software implementation and is 85 times faster than hardware implementations of the same algorithm with only 60% area overhead. Our three designs are scalable and their performance are only limited by the device used. As FPGAs are growing steadily, the

parametric nature of our modules enables them to fully explore the design space available in any current and future project.

Current and future work includes extending our replication and pipelining methods to the exponentiator and the prime tester, making our parametric multiplier capable of reaching the very low-area end of the design space and developing tools that automate our replication and pipelining approaches.

Acknowledgments. The support of UK EPSRC and BlueRISC is gratefully acknowledged.

References

1. Gmp library manual, `http://gmplib.org/manual/`
2. RSA Labs article on RSA security, `http://www.rsa.com/rsalabs/node.asp?id=2004`
3. Blum, T., Paar, C.: High-radix Montgomery modular exponentiation on reconfigurable hardware. IEEE Trans. Comput. 50(7), 759–764 (2001)
4. Bunimov, V., Schimmler, M., Tolg, B.: A complexity-effective version of Montgomery's algorithm. In: Workshop on Complexity Effective Designs (2002)
5. Cheung, R., Brown, A., Luk, W., Cheung, P.: A scalable hardware architecture for prime number validation. In: IEEE Int. Conf. on Field-Programmable Technology, pp. 177–184 (2004)
6. Daly, A., Marnane, W.: Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic. In: ACM Symp. on FPGAs, pp. 40–49 (2002)
7. Fry, J., Langhammer, M.: RSA & Public key cryptography in FPGAs. CDC (2003)
8. Mclvor, C., McLoone, M., McCanny, J.: Fast Montgomery modular multiplication and RSA cryptographic processor architectures. In: 37th Asilomar Conf. on Signals, Systems and Computers, vol. 1, pp. 379–384 (2003)
9. Narh Amanor, D., Paar, C., Pelzl, J., Bunimov, V., Schimmler, M.: Efficient hardware architectures for modular multiplication on FPGAs. In: Int. Conf. on Field Programmable Logic and Applications, pp. 539–542 (2005)
10. Nedjah, N., de Macedo Mourelle, L.: A review of modular multiplication methods and respective hardware implementation. Informatica 30(1), 111–129 (2006)
11. Oksuzoglu, E., Savas, E.: Parametric, secure and compact implementation of RSA on FPGA. In: Int. Conf. on Reconfigurable Computing and FPGAs, pp. 391–396 (2008)
12. Suzuki, D.: How to maximize the potential of FPGA resources for modular exponentiation. In: Workshop on Crypt. Hardware and Emb. Sys., pp. 272–288 (2007)
13. Tang, S., Tsui, K., Leong, P.: Modular exponentiation using parallel multipliers. In: IEEE Int. Conf. on Field-Programmable Technology (FPT), pp. 52–59 (2003)

A Reconfigurable Implementation of the Tate Pairing Computation over $GF(2^m)^\star$

Weibo Pan and William Marnane

Dept. of Electrical and Electronic Engineering
University College Cork, Cork City, Ireland
{weibop,liam}@rennes.ucc.ie

Abstract. In this paper the performance of a closed formula implemented in reconfigurable hardware for the Tate pairing Algorithm over the binary field of $GF(2^m)$ is studied. Using the algorithm improvement of Soonhak Kwon [2], the schedule for performing the Tate pairing without a square root operation is explored along with the area and time consumption trade-offs involved in the hardware implementation of the target algorithm.

Keywords: Tate pairing, FPGA implementation.

1 Introduction

Elliptic curve cryptography (ECC) is a popular cryptographic scheme which achieves a high level of security using small key sizes. With a 163 bit key size, ECC can ensure the same security level as a 1024 bit key RSA cryptography[11].

Based on ECC operations, pairing is a new type of public-key cryptographic scheme. Many cryptographic schemes based on the bilinear pairings [4] have been developed to exploit the pairing algorithm of Miller [3]. The most popular implementation of the Miller algorithm are the Tate pairing and the Weil Pairing. For key sizes that are likely to be used in practice, the Tate pairing is proved to be more efficient than the Weil Pairing in all fields [1,5]. There have been many algorithm developments and improvements to speed up the computation of the Tate pairing [4,6,7,8]. The notion of the squared Tate pairing was introduced by Eisentrager [9] and Barreto et al. [4] to eliminate the need for division as the denominator becomes one after the final exponentiation.

Implementations of these algorithms in reconfigurage hardware such as FPGAs depends on the underlying field ($GF(p)$, $GF(3^m)$ and $GF(2^m)$) and a survey of the previous works done by Dormale and Quisquater [10] summarized these FPGA-based implementations. In this paper, we show the FPGA implementation of the refined algorithm of Tate pairing introduced by Soonhak Kwon [2]. This algorithm showed that an efficient closed formula can be obtained for the computation of the Tate Pairing for supersingular elliptic curves over a binary

* This material is based upon works supported by the Science Foundation Ireland under Grant No. [SFI/ 08/RFP/ENE1643].

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 80–91, 2010.

field $GF(2^m)$ with odd dimension m. The implementation of the algorithm using different digit sizes and different numbers of multipliers are examined.

2 Arithmetics over Binary Fields

Since the finite field arithmetic underpins the pairing based cryptosystems, the efficiency of the reconfigurable design will depend on the efficiency of the basic operations over such field. In this section, hardware architectures for arithmetic operations in field of characteristic 2 are discussed.

2.1 Architecture for Computation over $GF(2^m)$

The basic arithmetic operations over binary field are the addition, multiplication, squaring and inversion over $GF(2^m)$which are the building blocks for the Tate pairing.

Addition. As a basic operation, addition of two elements on $GF(2^m)$ is performed as per $c(x) = a(x) + b(x)$. Since $GF(2)$ addition is performed using an XOR gate, the addition in $GF(2^m)$ simply requires an XOR chain. It is noted that in the field of $GF(2^m)$, the relationship $a + a = 0$ exists. Thus addition and subtraction are equivalent here. An addition over $GF(2^m)$ is represented as **A** in this paper.

Multiplication. Multiplication of two elements $c(x) = (a(x) * b(x)) \bmod f(x)$ is shown as: $c(x) = (a(x) * b(x)) = (\sum_{i=0}^{m-1} a_i x^i * \sum_{j=0}^{m-1} b_j x^j) \bmod f(x)$. This equation comprises of two parts. The first part is a multiplication with two inputs of degree $m-1$, and an output of degree $2m-2$. And the second part is a reduction, represented as 'mod $f(x)$', which reduces the result of the first part to degree $m-1$ or less, through the irreducible polynomial $f(x)$. The cost of this reduction is small because the fixed irreducible polynomial in the base field used for the pairings has a small hamming weight, such as trinomial or pentanomial.

A Digit-Serial Multiplication (DSM) was introduced by Hankerson et al. [12]. The DSM Algorithm takes elements $a(x)$ and $b(x)$ and outputs the products modulo the fixed irreducible polynomial $f(x)$. This algorithm computes d bits of the input $b(x)$ in an iteration and it takes $n = [\frac{m}{d}]$ iterations to finish the multiplication. As can be estimated, the larger the digit size d used, the larger the area of the multiplier and at the same time, the less time arequired to complete a multiplication. There is a trade-off between area and computation time a this performance will be discussed in the coming sections. A multiplication over $GF(2^m)$ is represented as **M**.

Squaring. A bit-parallel squaring architecture introduced in [13] is used in this work. Instead of performing a multiplication, this squaring architecture simply interleaves the input with zeroes, with a $2m-1$ to m bit reduction block following it. A Squaring over $GF(2^m)$ is represented as **S**.

Division. Division in $GF(2^m)$ computes $c(x) = a(x)/b(x) \bmod f(x)$. It is equivalent to an inversion by simply setting the input $a(x)$ of the division to $1 \in GF(2^m)$. A field division can be performed using an architecture based on the Extended Euclidean Algorithm introduced in [14]. The architecture computes the result in $2m$ clock cycles and requires a large area in practice. In comparison to the time taken for multiplication using the DSM, the time taken for division of $2m$ clock cycles is quite time consuming. Since the division only exists in the final exponentiation, only one divider will be used in the design of this algorithm to keep the area small.

2.2 Architecture for Computation over $GF(2^{4m})$

The tate pairing requires Multiplication, Squaring and Division over the extension field $GF(2^{4m})$ and the field is defined using the polynomial $p(x) = x^4+x+1$.

***GF*(2^4*m*) Multiplication.** A multiplication in this extension field can be implemented using an architecture designed by [16] and presented by Keller [17] as shown in Fig.1. This architecture used in this work requires 9 **M** and 16 **A**.

***GF*(2^{4m}) Squaring.** Similar to the $GF(2^m)$ squarer, the dedicated $GF(2^{4m})$ squarer is shown in fig. 2. This architecture computes $c(x) = a^2(x) \bmod p(x)$ where $p(x) = x^4 + x + 1$, $a(x) \in GF(2^{4m})$. As can be seen clearly from its architecture, the squarer is much more efficient in timing and area than the $GF(2^{4m})$ multiplier.

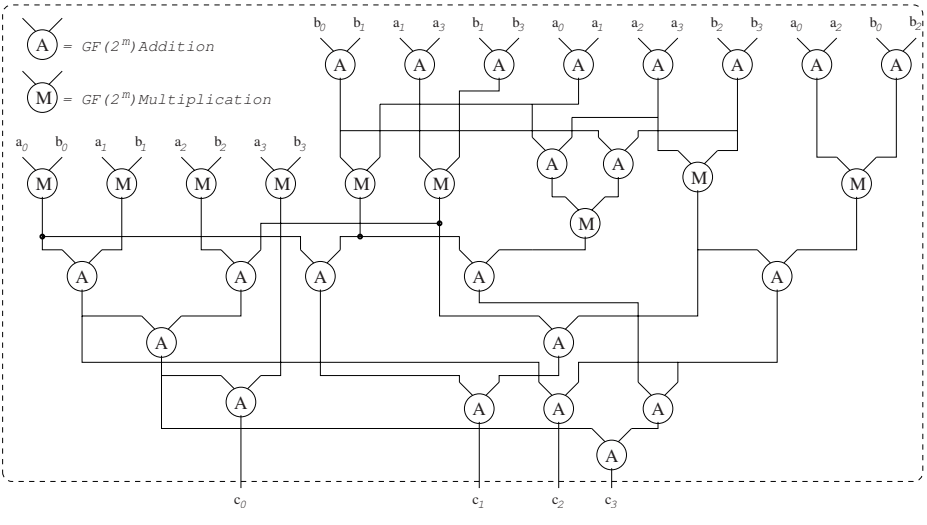


Fig. 1. $GF(2^{4m})$ Multiplier

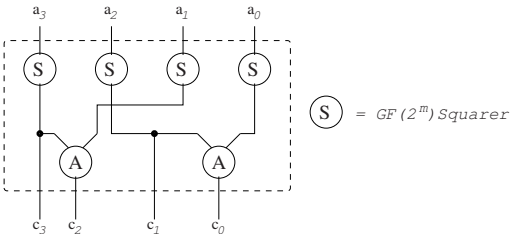


Fig. 2. $GF(2^{4m})$ Squarer

***GF*(2^{4m}) Inversion.** An algorithm for inversion over $GF(p^n)$ was presented in [18], is shown as Algorithm 1.

Algorithm 1. $GF(p^n)$ Inversion Algorithm

Input: $a(x)$, $p(x)$ such that $deg(a(x)) < deg(p(x))$
Output: $c(x) = a(x)^{-1} \bmod p(x)$
Initialize: $b = 0; c = 1; p = p(x); g = a(x)$
while $deg(p) \neq 0$ do
  if $deg(p) < deg(g)$ then
    exchange p, b with g, c respectively
  end
  $j = deg(p) - deg(g)$
  $\alpha = g^2_{deg(g)}$
  $\beta = p_{deg(p)} \cdot g_{deg(g)}$
  $\gamma = g_{deg(g)} \cdot p_{deg(p)-1} - p_{deg(p)} \cdot g_{deg(g)-1}$
  $p = \alpha.p - (\beta.x^j + \gamma.x^{j-1}).g$
  $b = \alpha.b - (\beta.x^j + \gamma.x^{j-1}).c$
  if $deg(p) = deg(g)$ then
    $p = g_{deg(p)}.p - p_{deg(p)}.g$
    $b = g_{deg(p)}.b - p_{deg(p)}.c$
  end
end
return $c(x) = a(x)^{-1} = b = p_0^{-1}.b$

In the case of our work, for the field $GF(p^n)$, p is set to 2^m, and $n = 4$. In Algorithm 1, function $deg(p)$ returns the degree of the polynomial p. The input polynomial $p(x)$ is set to $x^4 + x + 1$ initially of degree 4. $a(x)$ in this algorithm is considered as a degree 3 polynomial. The while loop in this algorithm reduces the degree of p by one or more in every iteration until the degree of p reaches zero. It is obvious that only additions, multiplications, and substractions, which are treated the same as additions, are necessary in the for loop. At the end of this algorithm, in the step of final exponentiation, a division and 4 multiplications

are performed to get the result $c(x)$ such that $(a(x) * c(x)) \bmod p(x) = 1$, where $a(x) = \sum_{i=0}^{3} a_i x^i$.

3 Tate Pairing

3.1 Elliptic Curve over $GF(2^m)$

A supersingular elliptic curve on the binary finite field $GF(2^m)$ is defined by : $E(GF(2^m)) : Y^2 + Y = X^3 + X + b$, where b ∈ {0,1}. Point $P = (x, y)$ on the curve is defined as a pair of elements $(x, y) \in GF(2^m)$ which satisfy the curve equation. In the field of $GF(2^m)$, an element is represented as a polynomial which is made up of a binary sequence of degree $m - 1$. Therefore a point (x, y) on the curve $E(GF(2^m))$ is represented by two degree $m-1$ binary polynomials, i.e., $2m$ bits.

3.2 Algorithm of Tate Pairing

For cryptographic operations, the Tate pairing is generally represented by $e_l(P; Q)$, where P and Q are points of order l on the elliptic curve $E(GF(2^m))$. This function evaluates to a point in the extension field $GF(2^{4m})$.

The most important property of the Tate pairing is that of bilinearity shown in eq(1), which forms the basis of a wide range of pairing protocols.

$$e_l(aP; bQ) = e_l(P; Q)ab \tag{1}$$

An efficient method of computing the Tate pairing is given in Miller's Algorithm in [3]. A modified scheme of Miller's Algorithm is presented by Duursma-Lee in [8]. The algorithm improved by Kwon [2] improved the Duursma-Lee's algorithm by avoiding the square root computation. Three curves are applicable for this algorithm. Among then, the curve $Eb : Y^2 + Y = X^3 + X$ has the embedding degree 4. For better FPGA implementation, Shu and Kwon improved the algorithm in [15], as shown in Algorithm 2. We chose it for our implementation.

In Algorithm 2, the Tate pairing for a supersingular elliptic curve over $GF(2^m)$ is computed. With the m-bit input sequences P and Q on the elliptic curve Eb, we get the output $C(x)$ from this algorithm. The output $C(x) = \{c_3, c_2, c_1, c_0\}$ is the pairing result of this algorithm. c_3, c_2, c_1 and c_0 are m-bit binary sequences. From the algorithm above, we can see that the underlying field operations needed in the Tate pairing computation are **S** and **M**.

Moreover, in the final exponentiation step, $C^{2^{2m}-1}$ can be written in the form of $(C^{2^m})^2 * C^{-1}$. Since there is a well known property of $GF(2^m)$ that $a^{2^m} = a$ where $a \in GF(2^m)$, computing $(C^{2^m})^2$ needs only several additions and squarings. Together with the inversion for the computation of C^{-1}, the final exponentiation can be easily performed.

Algorithm 2. Improved algorithm for computing Tate pairing

Input: $P = (\alpha, \beta), Q = (x, y)$ Output: $C = e_l(P; Q)$
1. $C \leftarrow 1$
2. $u \leftarrow x^2 + y^2 + b + \frac{m-1}{2}, v \leftarrow u, y \leftarrow y^2$
3. $\alpha \leftarrow \alpha^4, \beta \leftarrow \beta^4, \gamma \leftarrow \alpha(v+1)$
4. for $(i = 1$ to $m; i++)$
5. $A(t) \leftarrow \gamma + u + \beta + (\alpha + v)t + (\alpha + v + 1)t^2$
6. $C \leftarrow C^2$
7. $C \leftarrow C * A(t)$
8. if $i < m$ then
9. $u \leftarrow u + v + 1, v \leftarrow v + 1$
10. $\alpha \leftarrow \alpha^4, \beta \leftarrow \beta^4, \gamma \leftarrow \alpha(v+1)$
11. end if
12. end for
13. $C(x) = C(x)^{2^{2m}-1}$
14. return $C(x)$

3.3 Reconfiguration of the $GF(2^{4m})$ Multiplication

Different from normal $GF(2^{4m})$ Multiplication, the operation of $C \leftarrow C * A(t)$ can be simplified. Let

$A(t) = w + zt + (z+1)t^2$, where $w = \gamma + u + \beta, z = \alpha + v$.

$C(x)$ can be written in the form of $c_0 + c_1 t + c_2 t^2 + c_3 t^3$, where $c_i \in GF(2^m)$. Thus for the step $C \leftarrow C * A(t)$, we have $C(t) * (w + zt + (z+1)t^2) = c'_0 + c'_1 t + c'_2 t^2 + c'_3 t^3$, where

$c'_0 = c_0 w + (c_2 + c_3)(z+1) + c_3,$

$c'_1 = c_0 w + (c_1 + c_2 + c_3)w + (c_0 + c_2 + c_3)(w + z + 1) + c_3(z+1) + c_0 + c_3,$

$c'_2 = c_0 w + (c_1 + c_2 + c_3)w + (c_0 + c_2 + c_3)(w + z + 1) + (c_1 + c_2)(w + z + 1) + c_1,$

$c'_3 = (c_1 + c_2 + c_3)w + (c_1 + c_2)(w + z + 1) + c_2.$

As can be easily seen from the operations above, to update $A(t)$, $5\mathbf{A}+4\mathbf{S}+1\mathbf{M}$ are required in every iteration. And to update $C(t)$ in one iteration, $19\mathbf{A} + 4\mathbf{S} + 6\mathbf{M}$ are required. In a parallel architecture, the update of $A(t)$ and $C(t)$ can be done at the same time. By using different numbers of multipliers, we have different schedules for computing step 5-11 in Algorithm 2, shown in Fig.3.

In Fig.3, blocks PRE and POST represent the pre- and post-computations of the $7\mathbf{M}$. The pre-computations consist of $14\mathbf{A} + 8\mathbf{S}$ and takes up 44 clock cycles. The post-computations consist of $12\mathbf{A}$ and some memory writting operations, taking up 53 clock cycles. The operation for a single $\mathbf{M}$ takes $n = \frac{m}{d}$ clock cycles, with one additional clock cycle reading the product from the output into memory. In this work, the processor will always spend 104 clock cycles on the operations excluding multiplicaiton because they are performed in serial. In this case, only 1 Adder and 1 Squarer are needed in the architecture.

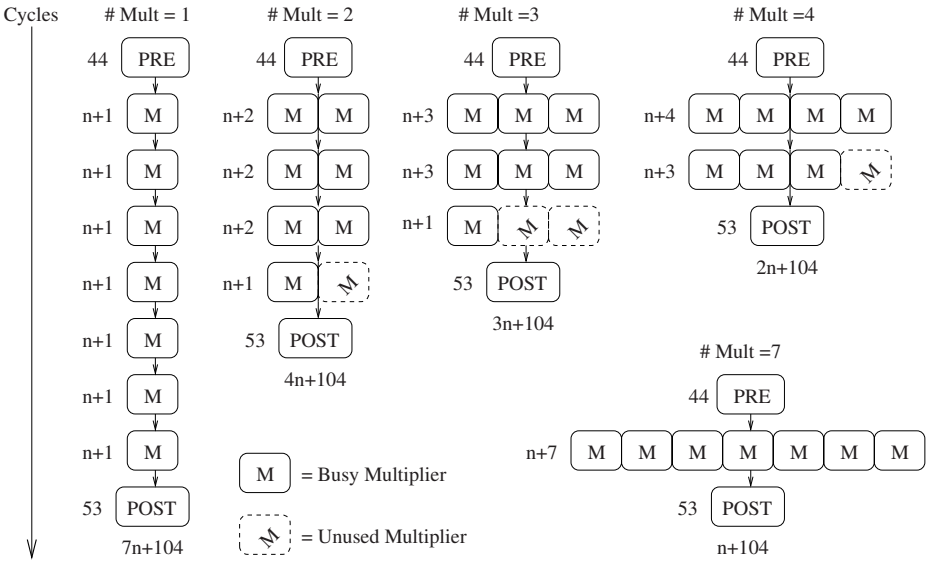


Fig. 3. Schedules for different numbers of Multipliers

With different number $\#Mult$ of multipliers, the design can deal with at most $\#Mult$ multiplications at the same time. We use $\#Mult$ =1, 2, 3, 4 and 7 in this paper. Using 5 and 6 multipliers are too inefficient due to the unused multipliers. Since only 7 multiplications are needed in each iteration, putting more multipliers in the design will not speed up the operation time. As can be seen in Fig.3, adding multipliers reduces a multiple of n clock cycles. And using only one Multiplier, which gives a minimum area design, takes a long operation time of $7n + 104$ clock cycles.

3.4 Architecture Design of Tate Pairing

Fig.4 shows the top architecture for implementing the Tate pairing using Algorithm 2. As can be seen in Fig.4, the top level of the architecture is based on the $GF(2^m)$ modules. A memory bank is also used to store the inputs and intermediate variables required in the algorithm. The $GF(2^m)$ blocks and memory are controlled by a Finite State Machine (FSM) controller, which iterates through an instruction set contained in ROM. No $GF(2^{4m})$ modules can be seen on the top level because the $GF(2^m)$ function blocks are reused for each of the $GF(2^{4m})$ operations. The instructions of the $GF(2^{4m})$ operations are stored in the ROM, and all the operations have accesses to all the $GF(2^m)$ blocks so that the design will reach a high utilization of all its hardware.

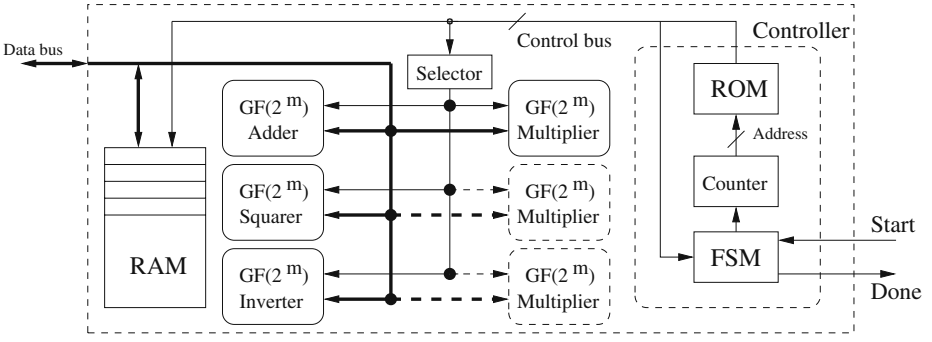


Fig. 4. FPGA architecture of Tate pairing

4 Implementation Results

In this implementation, the Tate pairing is written in VHDL. The design is simulated using ModelSim XE III 6.3c, and the target FPGA technology is Xilinx Virtex II xc2v6000-4ff1152. We designed different numbers (1, 2, 3, 4 and 7) of multipliers, and different digit sizes d (1, 4, 8 and 16) to implement the target algorithm at the field size of $m = 251$ and $m = 163$. The designs can reach a minimum 108 MHz frequency for $m = 251$ and 111 MHz for $m = 163$. For better comparison, we are using 108 MHz in all the cases of $m = 251$, and 111 MHz in all the cases of $m = 163$.

The post Place & Route results of the designs are shown in Table 1 and Table 2. As can be seen from the tables, the area increases when the digit size

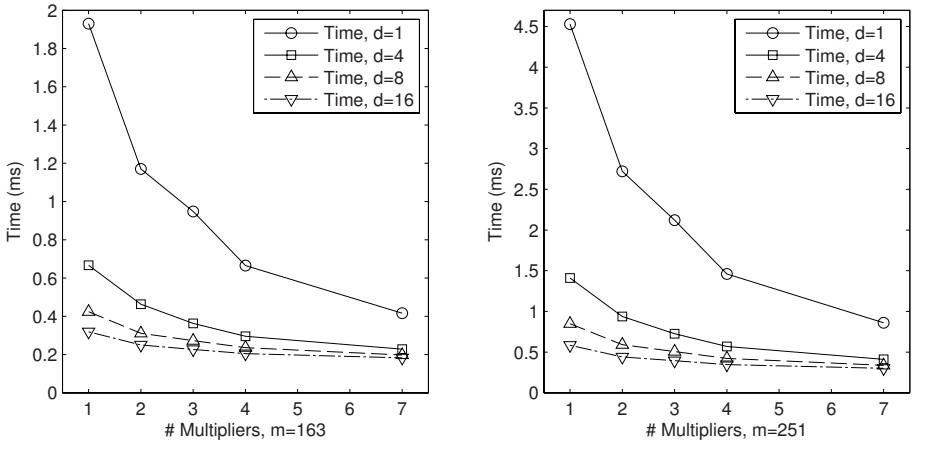


Fig. 5. Calculation time of the Tate pairing design

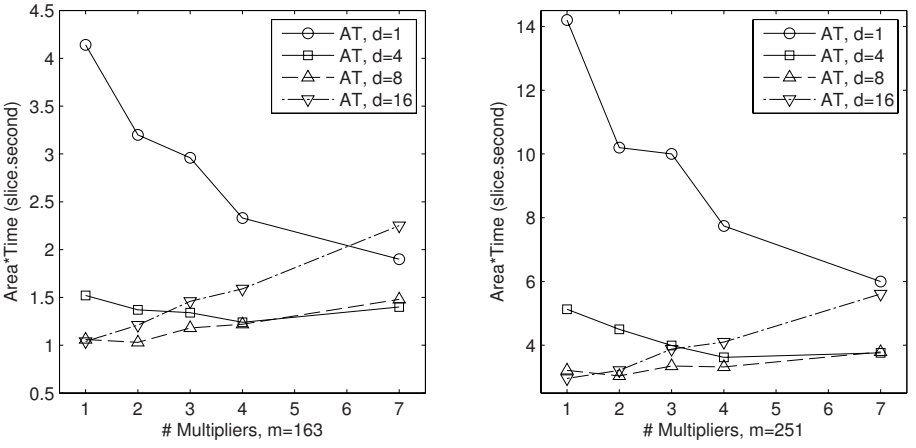


Fig. 6. Area*Time performance of the Tate pairing design

Table 1. Implementation results, Xilinx xc2v6000-4ff1152, 111MHz, m=163

Mult	d	Slices	Time(ms)	A.T(s.Slice)	d	Slices	Time(ms)	A.T(s.Slice)
1	1	2147(6%)	1.93	4.14	4	2286(6%)	0.667	1.52
	8	2498(7%)	0.424	1.06	16	3268(9%)	0.319	1.04
2	1	2736(8%)	1.17	3.20	4	2958(9%)	0.463	1.37
	8	3326(9%)	0.311	1.03	16	4825(14%)	0.250	1.21
3	1	3125(9%)	0.948	2.96	4	3684(10%)	0.363	1.34
	8	4316(12%)	0.273	1.18	16	6452(19%)	0.227	1.46
4	1	3503(10%)	0.665	2.33	4	4192(12%)	0.296	1.24
	8	5176(15%)	0.236	1.22	16	7761(22%)	0.205	1.59
7	1	4563(13%)	0.416	1.90	4	6155(18%)	0.228	1.40
	8	7490(22%)	0.198	1.48	16	12277(36%)	0.183	2.25

and the number of multipliers goes up but the time it takes to complete the operations reduces. So we take the Area*Time (A.T) product as a parameter.

Fig.5 and Fig.6 show the time for computing the Tate pairing algorithm and the A.T products of the designs for both $m = 251$ and $m = 163$. As can be seen from the timing curves in Fig.5, adding more multipliers or increasing the digit size d both speed up the computation. The A.T product curves in Fig.6 show that increasing the digit size d and the number of multipliers do not always give improvement to the performance of the design. Since a lower A.T product is always wanted for a design, the design with 2 multiplier and digit size $d = 8$ makes the best performance for and $m = 163$. And using 1 $GF(2^m)$ multiplier and $d = 16$ has the lowest A.T product for the $m = 251$ designs.

Comparing the results with other FPGA implementations of the Tate pairing implemented in $GF(2^m)$ is shown in Table 3. As can be seen, Keller implemented a design on the same field of $GF(2^{4*251})$ [17], on the same technology of Xilinx

Table 2. Implementation results, Xilinx xc2v6000-4ff1152, 108MHz, m=251

Mult	d	Slices	Time(ms)	A.T(s.Slice)	d	Slices	Time(ms)	A.T(s.Slice)
1	1	3135(9%)	4.53	14.2	4	3635(10%)	1.41	5.13
	8	3790(11%)	0.849	3.21	16	5072(15%)	0.584	2.96
2	1	3793(11%)	2.72	10.2	4	4799(14%)	0.937	4.50
	8	5132(15%)	0.592	3.04	16	7275(21%)	0.441	3.21
3	1	4736(14%)	2.12	10.0	4	5499(16%)	0.726	3.99
	8	6599(19%)	0.507	3.35	16	9838(29%)	0.394	3.88
4	1	5302(15%)	1.46	7.74	4	6343(18%)	0.570	3.62
	8	7856(23%)	0.423	3.32	16	11822(24%)	0.347	4.10
7	1	6965(20%)	0.861	6.00	4	9130(27%)	0.412	3.76
	8	11221(33%)	0.338	3.79	16	18693(55%)	0.300	5.61

Table 3. Results Comparison with other work of Tate pairing over $GF(2^m)$

Name, Ref.	Target Field	Design Size	Area	Time(ms)	Freq.(Mhz)
Keller,[17]	$GF(2^{251})$	$1Mult, d = 1$	3370	18.6	40
Keller,[17]	$GF(2^{251})$	$9Mults, d = 6$	26132	1.07	43
Shu,[15]	$GF(2^{239})$	$6Mults, d = 16$	18202	0.047	100
Shu,[15]	$GF(2^{283})$	$6Mults, d = 16$	22726	0.076	84
Li,[19]	$GF(2^{283})$	$12Mults, d = 32$	55,844	0.59	159
This work	$GF(2^{251})$	$1Mult, d = 1$	3135	4.53	111
This work	$GF(2^{251})$	$7Mults, d = 16$	18693	0.30	108

Virtex II XC2V6000-4ff1152. He got the calculation time of at least 1.07 ms, with an area of 26132(77%) slices. H. Li et al. implemented their design on Xilinx Virtex 4 XC4VFX140-FF1517-11 [19] where the calculation time is 0.59 ms, almost twice our calculation time, but at the cost of 55,844 slices, which is much larger than our design. The design done by Shu et al. [15] was implemented on Xilinx XC2VP1000-6FF-1702. Their designs reached a smaller calculation time. They expanded all the Additions and Squarings by adding more adders and squarers. The designs operate with the additions and squarings in parallel. This method improves the calculation speed when $\#Mult$ and digit size d both are very big, but increases the design area significantly.

5 Conclusion

The implementation of a closed formula for the Tate pairing over the binary field of $GF(2^m)$ has been presented. In this work, we improved the algorithm by putting all the 7 $GF(2^m)$ multiplication in parallel, this reduces the calculation time significantly when using a maximum of 7 $GF(2^m)$ multipliiers. In the top level design, any $GF(2^{4m})$ operation can call and use all the basic $GF(2^m)$ function blocks. This makes the design of high efficiency and flexibility, and reduces its area significantly. As mentioned above, using 2 $GF(2^m)$ multipliers and setting the digit size $d = 8$ reaches the best trade-off between time and area

for $m = 163$. And with 1 $GF(2^m)$ multiplier and $d = 16$, the $m = 251$ design will reach its lowest A.T product. However, in an area constrained application, one should choose the best suitable number of multipliers and digit size for a good performance. While if no area constraint is applied to the designs, for the benifit of a higher speed, using maximum multipliers with a larger digit size d is always suggested.

References

1. Beuchat, J.-L., Brisebarre, N., Detrey, J., Okamoto, E., Rodríguez-Henríquez, F.: A Comparison between Hardware Accelerators for the Modified Tate Pairing over F_{2^m} and F_{3^m}. In: Galbraith, S.D., Paterson, K.G. (eds.) Pairing 2008. LNCS, vol. 5209, pp. 297–315. Springer, Heidelberg (2008)
2. Kwon, S.: Efficient Tate Pairing Computation for Elliptic Curves over Binary Fields. In: Boyd, C., González Nieto, J.M. (eds.) ACISP 2005. LNCS, vol. 3574, pp. 134–145. Springer, Heidelberg (2005)
3. Miller, V.S.: Short Programs for functions on Curves (1986) (unpublished manuscript)
4. Barreto, P.S.L.M., Kim, H.Y., Lynn, B., Scott, M.: Efficient algorithms for pairing-based cryptosystems. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 354–369. Springer, Heidelberg (2002)
5. Granger, R., Page, D.L., Smart, N.P.: High Security Pairing-Based Cryptography Revisited. In: Hess, F., Pauli, S., Pohst, M. (eds.) ANTS 2006. LNCS, vol. 4076, pp. 480–494. Springer, Heidelberg (2006)
6. Granger, R., et al.: Hardware and Software Normal Basis Arithmetic for Pairing Based Cryptography in Characteristic Three. IEEE Transactions on Computers 54, 852–860 (2005)
7. Granger, R., et al.: On Small Characteristic Algebraic Tori in Pairing-Based Cryptography. LMS Journal of Computation and Mathematics 9, 64–85 (2006)
8. Duursma, I.M., Lee, H.-S.: Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 111–123. Springer, Heidelberg (2003)
9. Eisenträger, K., Lauter, K., Montgomery, P.L.: Improved weil and tate pairings for elliptic and hyperelliptic curves. In: Buell, D.A. (ed.) ANTS 2004. LNCS, vol. 3076, pp. 169–183. Springer, Heidelberg (2004)
10. Dormale, G.M., et al.: High-speed hardware implementations of Elliptic Curve Cryptography: A survey. Journal of Systems Architecture 53(2-3), 72–84 (2007)
11. Gupta, V., et al.: Performance analysis of elliptic curve cryptography for SSL. In: Proceedings of the 1st ACM workshop on Wireless security, pp. 87–94. ACM Press, New York (2002)
12. Hankerson, D., et al.: Guide to Elliptic Curve Cryptography. Springer, Heidelberg (2004)
13. Mastrovito, E.D.: VLSI Architectures for Computation in Galois Fields. PhD thesis, Dept. Electrical Engineering, Linkoping University, Linkoping, Sweden (1991)
14. Shantz, S.C.: From Euclid's GCD to Montgomery Multiplication to the Great Divide. Tech. Rep. SMLI TR-2001-95, Sun Microsystems, pp. 1–10 (2001)
15. Shu, C., et al.: FPGA Accelerated Tate Pairing Based Cryptosystems over Binary Fields. In: Proceedings of the IEEE International Conference on Field Programmable Technology 2006, pp. 173–180. IEEE, Los Alamitos (2006)

16. Shantz, S.C., Karatsuba, A., Ofman, Y.: Multiplication on many-digital numbers by automatic computers. Translation in Physics-Doklady 7, 595–596
17. Keller, M., Kerins, T., Crowe, F., Marnane, W.P.: FPGA implementation of a gF(2^m) tate pairing architecture. In: Bertels, K., Cardoso, J.M.P., Vassiliadis, S. (eds.) ARC 2006. LNCS, vol. 3985, pp. 358–369. Springer, Heidelberg (2006)
18. Lim, C.H., Hwang, H.S.: Fast implementation of elliptic curve arithmetic in $GF(p^n)$. In: Imai, H., Zheng, Y. (eds.) PKC 2000. LNCS, vol. 1751, pp. 405–421. Springer, Heidelberg (2000)
19. Li, H., et al.: FPGA implementations of elliptic curve cryptography and Tate pairing over a binary field. Journal of Systems Architecture: the EUROMICRO Journal 54(12), 1077–1088 (2008)

Application Specific FPGA Using Heterogeneous Logic Blocks

Husain Parvez, Zied Marrakchi, and Habib Mehrez

LIP6, Université Pierre et Marie Curie
4, Place Jussieu, 75005 Paris, France
{parvez.husain,zied.marrakchi,habib.mehrez}@lip6.fr

Abstract. An Application Specific Inflexible FPGA (ASIF) [12] is an FPGA with reduced flexibility that can implement a set of application circuits which will operate at different times. Application circuits are efficiently placed and routed on an FPGA in such a way that total routing switches used in the FPGA architecture are minimized. Later all unused routing resources are removed from the FPGA to generate an ASIF. An ASIF which is reduced from a heterogeneous FPGA (i.e. containing hard-blocks such as Multipliers, Adders and RAMS etc) is called as a Heterogeneous-ASIF. This work shows that a standard-cell based Heterogeneous-ASIF using Multipliers, Adders and Look-Up-Tables for a set of 10 opencores application circuits is 85% smaller in area than a single driver FPGA using the same blocks, and only 24% larger than the sum of areas of their standard-cell based ASIC version. If the Look-Up-Tables are replaced with a set of repeatedly used hard logic gates (such as AND gate, OR gate, flip-flops etc), the ASIF becomes 89% smaller than the Look-Up-Table based FPGA and 3% smaller than the sum of ASICs. The area gap between ASIF and sum of ASICs can be further reduced if repeatedly used groups of standard-cell logic gates in an ASIF are designed in full-custom. One of the major advantages of an ASIF is that just like an FPGA, an ASIF can also be reprogrammed to execute new or modified circuits, but at a very limited scale. A new CAD flow is presented to map application circuits on an ASIF.

1 Introduction

Field Programmable Gate Arrays (FPGAs) are reconfigurable devices that can be easily re-programmed to execute a large variety of applications. These devices are therefore very effective and economical for a product requiring low-volume production. However, the flexibility of FPGAs makes them much larger, slower, and more power consuming than their counterpart ASICs (Application Specific Integrated Circuits) [8]. Consequently, FPGAs are not suitable for applications requiring high volume production, high performance or low power consumption. An ASIC has area, speed and power advantages over an FPGA; but at the expense of higher non-recurring engineering (NRE) cost and higher time-to-market. The NRE cost and time-to-market are reduced with the advent of a new breed of ASICs known as Structured-ASIC. Structured-ASIC consist of array of

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 92–109, 2010.

optimized elements which implement a desired functionality by making changes on few upper mask layers.

FPGA vendors have also started giving provision to migrate FPGA based application to Structured-ASIC [1]. The main idea is to prototype, test and even ship initial few designs on an FPGA; later it can be migrated to Structured-ASIC for high volume production [6]. In this regard, Altera has proposed a clean migration methodology [13] that ensures equivalence verification between FPGA and its Structured-ASIC (known as HardCopy [1]). However, migration of an FPGA based application to HardCopy can execute only a single circuit. HardCopy totally loses the quality of an FPGA which uses the same hardware for executing multiple applications at different times. So, we propose a new reduced FPGA that can execute a set of circuits at different times. This reduced FPGA is called as an Application Specific Inflexible FPGA (ASIF). When an FPGA-based product is in the final phase of its development cycle, and if the set of circuits to be mapped on the FPGA are known, it can be reduced for all the given set of circuits. This work extends the idea of an ASIF [12] in two main directions. (i) A Heterogeneous-ASIF is proposed; it is an ASIF which uses hard-blocks such as Multipliers, Adders and RAMS etc. A Heterogeneous-ASIF for a given set of application circuits is also compared with sum of areas of their standard-cell based ASICs. (ii) A major advantage of an ASIF over sum of areas of ASICs is achieved by proposing a new CAD flow which can map application circuits on an ASIF. Just like an FPGA, an ASIF can also be reprogrammed to execute new or modified circuits, but at a limited scale.

The concept of an ASIF is similar to configurable ASIC cores (cASIC [5]); cASIC is a reconfigurable device that can implement a limited set of circuits which operate at mutually exclusive times. However, cASIC and ASIF have several major differences. cASIC is intended as an accelerator in a domain-specific systems-on-a-chip, and is not designed to replace the entire ASIC-only chip. For that reason, cASIC supports only full-word logic blocks (such as 16-bit wide multipliers, adders, RAMs etc) to implement data-path circuits. However, heterogeneous-ASIF presented in this work supports both fine- and coarse-grained logic blocks. The routing network used by cASIC and ASIF are totally different. cASIC uses 1-D segmented bus interconnect, whereas ASIF uses 2-D mesh interconnect. Another major difference between cASIC and ASIF is the approach with which their routing networks are optimized. cASIC is generated using a constructive bottom-up "insertion" approach with reconfigurability inserted through the addition of multiplexers and demultiplexers. On the contrary, an ASIF is generated using an iterative top-down "removal" technique in which different circuits are mapped onto an FPGA; flexibility is removed from it to support only the given set of circuits. The benefit of a "removal" approach over an "insertion" approach is that any existing FPGA architecture can be reduced to an ASIF using this "removal" technique.

This paper considers only the area optimization of an ASIF. Delay and power parameters are not explored in this work. Section 2 describes a reference FPGA architecture that is reduced to an ASIF; it also presents a software flow to map

a circuit on the reference architecture. Section 3 discusses floor-planning techniques for different heterogeneous blocks. Section 4 describes an ASIF generation technique. Section 5 presents the experimental results using a set of opencores application circuits. A new placement algorithm for mapping application circuits on an ASIF is presented in section 6. Possible layout generation methodologies for an ASIF are presented in section 7. Finally section 8 presents a conclusion and future work.

2 Reference FPGA Architecture and Software Flow

This section describes a reference FPGA architecture that will be reduced to an ASIF. This reference FPGA is a mesh-based VPR-style (Versatile Place & Route) [4] heterogeneous architecture, as shown in Figure 1. It contains Configurable Logic Blocks (CLBs) and Hard Blocks (HBs) arranged in columns. The input and output pads are arranged at the periphery. Each CLB contains a Look-Up-Table(LUT) and a flip-flop (FF). A block (referred to a CLB or HB) is surrounded by a uniform length, single driver, unidirectional routing network [10]. The input and output pins of a block connect with the neighboring routing channel. An FPGA tile showing the detailed connection of a CLB with its neighboring routing network is also shown in Figure 1. A unidirectional disjoint switch box connects different routing tracks (or wire segments) together. The connectivity of the routing channel with the input and output pins of a block, abbreviated as Fc(in) and Fc(out), is set to be maximum at 1. The channel width is varied according to the netlist requirement but remains in multiples of 2 [10].

A software flow (shown in Figure 2) maps a netlist on the heterogeneous FPGA. Input to this software flow is a structural netlist in VST (structured VHDL) format. This netlist is composed of traditional standard cell library instances and hard block (HB) instances. VST2BLIF tool is modified to convert

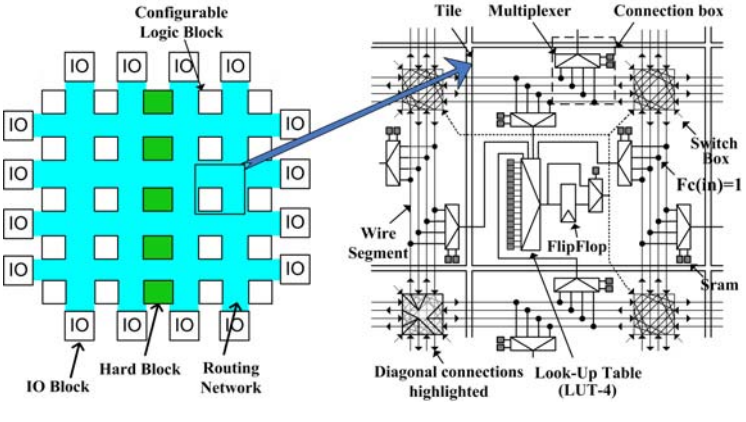


Fig. 1. Reference FPGA Architecture

Fig. 2. Software Flow

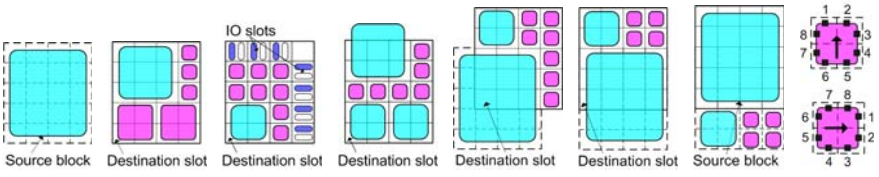


Fig. 3. Block Movement Cases

the VST file with hard blocks to BLIF [3] file format. Later PARSER-1 removes all instances of hard blocks and passes the remaining netlist to SIS [14] for synthesis into Look-Up-Table format. Dependence between HBs and remaining netlist is preserved by adding temporary input and output pins to the main netlist. After SIS, and later after packaging and conversion of the netlist to NET format through T-VPACK [2], PARSER-2 adds all the removed HBs into the netlist. It also removes all previously added temporary inputs and outputs.

A netlist file (in .NET format) includes CLBs (LUTs and/or FF), HBs (Hard Blocks) and IO (Inputs and Outputs) instances which are interconnected through signals called NETS. A software module named PLACER uses the simulated annealing algorithm [4] [7] to place the CLBs/HBs/IO instances on their respective blocks of FPGA. The bounding box (BBX) of a NET is a minimum rectangle that contains the driver instance and all receiving instances of a NET. The PLACER attempts to achieve a placement with a minimum sum of half-perimeters of the bounding boxes of all NETS. It moves an instance randomly from one block position to another. After each move operation, the BBX cost is updated incrementally. Depending on cost value and annealing temperature, the operation is accepted or rejected. After placement, a software module named ROUTER routes the netlist on the architecture. The router uses a pathfinder algorithm [11] to route the netlists using FPGA routing resources.

3 Floor-Planning Techniques

If a heterogeneous FPGA is to be reduced to an ASIF, the position of different blocks on the architecture can be optimized to achieve better area gains. For this purpose, the PLACER is modified to find out the positions of different blocks on the FPGA architecture for a given set of netlists. Initially, an FPGA architecture is defined using an architecture description file. BLOCKS of different sizes are

defined, and later mapped on a grid of equally sized SLOTS, called as a SLOT-GRID. Each BLOCK occupies one or more SLOTS. The type of the BLOCK and its input and output PINS are used to find the size of a BLOCK. In a LUT-4 based FPGA, a CLB occupies 1 slot, and a 16x16 multiplier occupies 5x5 slots. Input and output PINS of a BLOCK are defined, and CLASS numbers are assigned to them. PINS with the same CLASS number are considered equivalent; thus a NET targeting a receiver PIN of a BLOCK can be routed to any of the PINS of the BLOCK belonging to the same CLASS. The PLACER moves an instance of a netlist from one BLOCK to another, moves a BLOCK from one SLOT position to another, or rotates a BLOCK around its own axis. After each operation, the placement cost is updated for all disturbed NETS. Depending on cost value and annealing temperature, the operation is accepted or rejected. Multiple netlists are placed together to get a single architecture floor-planning for all netlists. With multiple netlist placements, each BLOCK can allow multiple instances to be mapped onto it; but multiple instances of the same netlist cannot be mapped onto a single BLOCK.

The PLACER performs an operation on a "source" and a "destination". The "source" is randomly selected to be either an instance from any of the input netlists or a BLOCK from the architecture. If the source is an instance, any random matching BLOCK is selected as its destination. If the source is a BLOCK to be rotated, the same source position becomes the destination as well.

If the source BLOCK is to be moved from one SLOT position to another, a random SLOT is selected as its destination. The rectangular window starting from this destination slot, having the same size and shape as the source BLOCK is called the destination window, whereas the window occupied by the source BLOCK is called a source window. The dashed line in Figure 3(a) is the source window, and the solid line in Figure 3(b) depicts a valid destination window. The source window always contains a single BLOCK, whereas destination window can contain multiple BLOCKS. The destination SLOT is rejected if (i) destination window exceeds the boundaries of the SLOT-GRID (as shown in Figure 3(c)), (ii) destination window contains at least one such BLOCK which exceeds the limits of destination window (as shown in Figure 3(d)) and (iii) source window overlaps destination window diagonally i.e. partial horizontal, partial vertical overlap (as shown in Figure 3(e)). The procedure is repeated until a valid destination slot is found. Once both the source and destination are selected, the PLACER performs an operation. Different operations performed by the PLACER are described as follows:

Instance Move: An instance move operation is applied on a source instance and a destination block. If the destination block is not occupied by any instance, the source instance is simply moved to the destination block. If the destination block is occupied by an instance, then a swap operation is performed.

Block Jump: If source window does not overlap destination window, a block JUMP operation is performed. All blocks in the destination window are moved to the source window, and the source block is moved into the destination window.

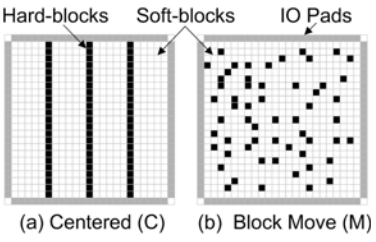


Fig. 4. Floor-planning Techniques

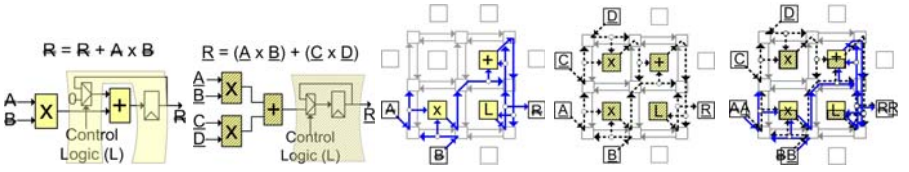
Block Translate: If source window overlaps destination window, then a translation operation is performed. Only horizontal and vertical translation is supported in this work. No diagonal translation is performed. Figure 3(f) and 3(g) show a case of vertical translation. The 5 blocks found on the upper 2 rows of the destination window (as shown in Figure 3f) are moved to the lower 2 rows of the source window (as shown in Figure 3g). The source block is then moved to the destination window.

Block Rotate: Rotation of BLOCKS is important when the classes of each of its pins are different. In such a case, the size of the bounding box varies depending on pin positions and their directions. Multiples of 90° rotation are allowed for all BLOCKS with a square shape, whereas only a 180° rotation is allowed for rectangular (non-square) BLOCKS. A 90° rotation for rectangular BLOCKS requires both rotation and move operation; which is left for future work. The orientation of BLOCK is used by the bounding box cost function to calculate the exact position and direction of each of its PINS. Figure 3(f) depicts a 90° clock-wise rotation.

Experiments in section 5 are performed using 3 different types of floor-plannings. (i) The blocks of same type are placed in columns positioned at equal distance with one another as shown in Figure 4(a) (referred as Column floor-planning, 'C'). (ii) Blocks perform a 'Block Jump' and 'Block Translate' as shown in Figure 4(b) (referred as Block Move floor-planning, 'M'). (iii) Blocks are both moved and rotated (referred as Block Move/Rotate floor-planning, 'MR').

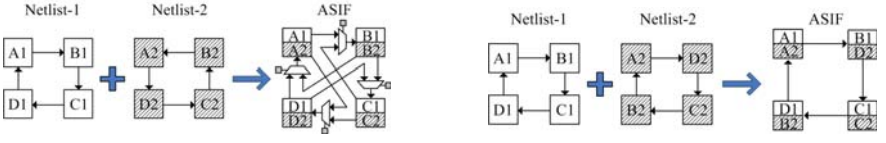
4 ASIF Generation

A simplistic ASIF generation technique is explained with the help of test circuits shown in Figure 5. Initially a target FPGA architecture is generated with maximum number of Blocks required by the two netlists shown in Figure 5(a) and (b). Both netlists are individually placed and then routed on the target architecture with minimum channel width. Figure 5(c) and 5(d) show the two netlists placed and routed individually on the target FPGA. Wires used by netlist-1 are continuous/blue, whereas wires used by netlist-2 are dotted/black. Grey wires are the unused wires in an FPGA. For simplification, Figure 5 does not show the



(a) Netlist-1 (b) Netlist-2 (c) Netlist-1 on FPGA (d) Netlist-2 on FPGA (e) ASIF for Netlist-1,2

Fig. 5. Test Circuits



(a) Normal placement (b) Efficient placement

Fig. 6. Efficient Placement

inner connection details of switch box and connection box. Once placement and routing of both the netlists is finalized on FPGA, ASIF is generated by removing all the routing resources that remain unused by both the netlists. Figure 5(e) shows an ASIF for the two netlists; grey switches and corresponding wires are removed to generate an ASIF. The area of an ASIF can further decrease if the netlists are efficiently placed and routed on the FPGA. Efficient placement tries to place the instances of different netlists in such a way that minimum routing switches are required in an FPGA. Later, efficient routing encourages different netlists to route their NETS on an FPGA with maximum common routing paths. After all netlists are efficiently placed and routed on FPGA, unused switches are removed from the architecture to generate an ASIF. Efficient placement and routing techniques are discussed in detail below.

4.1 Efficient Placement

Efficient placement tries to place driver instances of different netlists on a common block position, and their receiver instances on another common block. Later, efficient routing increases the probability to connect the driver and receiver instances of these netlists by using the same routing wires. Efficient placement can be understood with the help of an example shown in Figure 6. Figure 6(a) shows two simple, placed netlists; having minimum possible BBX placement cost. The ASIF for these two netlists requires 4 multiplexers (mux-2). Figure 6(b) shows the same two netlists that are placed efficiently with the same BBX cost just as for netlists in Figure 6(a); the ASIF for these netlists requires no switch at all.

$$\begin{aligned} \text{Cost} &= ((\text{W} * \text{BBX}) + ((100 - \text{W}) * \text{DC} * \text{NormalizationFactor})) / 100 \\ \text{Where } 0 \leq \text{W} \leq 100&, \text{NormalizationFactor} = \text{Initial BBX} / \text{Initial DC} \\ \text{BBX} &= \text{Bounding Box Cost, DC} = \text{Driver Count Cost} \end{aligned} \quad (1)$$

A simulated annealing algorithm with bounding box (BBX) cost function is commonly used to place each netlist individually on an FPGA. This type of placement is called here as an Intra-netlist placement. A new "Driver Count" (DC) cost function is proposed here to perform efficient placement. This cost function calculates the sum of driver blocks targeting the receiver blocks of the architecture over all netlists. A placement based upon "Driver Count" cost function is called here as Inter-netlist placement. Efficient placement considers a set of netlists simultaneously and aims at optimizing both intra-netlist and inter-netlist instance placements. The aim is to minimize the BBX cost of each netlist and the DC cost over all the netlists. In Figure 6(a) the ASIF has a "Driver Count" cost of 8, where each block has 2 different drivers. In Figure 6(b) the "Driver Count" cost is 4; each block has only one driver. Efficient placement uses a combination of both cost functions (i.e. the bounding box (BBX) cost function and the Driver Count (DC) cost function). Since BBX cost and DC cost are not of the same magnitude, initially both costs are adjusted by multiplying one of them by a normalization factor. This normalization factor is determined from the initial BBX and DC costs. Weighting coefficients are attributed to both cost functions and a new weighted cost is computed as shown in Equation 1. The simulated annealing algorithm later minimizes this new weighted cost. It should be noted that as the weight for DC function is increased the DC cost decreases, but the BBX cost increases, and vice-versa. With an increase in the BBX cost, more routing switches are required to route a netlist, which in turn means that more area is required. A trade-off needs to be searched to obtain a good solution. Experiments performed on 20 MCNC benchmark circuits [12] show that a BBX:DC weighting ratio of 4:1 gives the best area results.

Placement of multiple netlists is supported in the same way as used by cA-SIC [5]. With multiple netlist placements, each block of the architecture can allow mapping of multiple instances belonging to different netlists. All netlists are placed simultaneously; the placer chooses an instance randomly from any input netlist, and changes its position. The placer is also modified to support "Driver Count" cost function. The differences in the BBX Cost and the DC cost are updated incrementally. New weighted cost is calculated with the given weights; the simulated annealing algorithm uses this new cost to decide if the movement is accepted or rejected.

4.2 Efficient Routing

Efficient routing tries to minimize the total routing switches required in an ASIF. This is done by maximizing the shared switches required for routing all netlists on the FPGA. Efficient wire routing allows different netlists to route their NETS on an FPGA with maximum common routing paths. After all netlists are efficiently routed on FPGA, unused switches are removed from the architecture to generate an ASIF.

The pathfinder routing algorithm is modified to support efficient wire routing. Before we describe these changes in detail, a short introduction of the routing algorithm is presented here. An FPGA routing network is represented by a graph with nodes connecting each other through edges; each routing wire of the architecture is represented by a node, and connection between two wires is represented by an edge. When a netlist is routed on the FPGA routing graph, each NET (i.e. connection of a driver instance to its receivers) is routed using a congestion driven "Shortest Path" algorithm. A conflict is said to occur if two or more NETS of the same netlist share the same node (i.e. routed on the same wire). Once all NETS in a netlist are routed, one routing iteration is said to be completed. At the end of an iteration, there can be conflicts between different NETS sharing the same node. The congestion parameters are updated and iteration is repeated until routing converges to a feasible solution (i.e. no conflicts are found) or routing fails (i.e. "maximum" iteration count is reached). Multiple netlists can be routed on the FPGA by allowing nodes to be shared by multiple NETS belonging to different netlists; however the NETS belonging to the same netlist cannot share the same node (conflict).

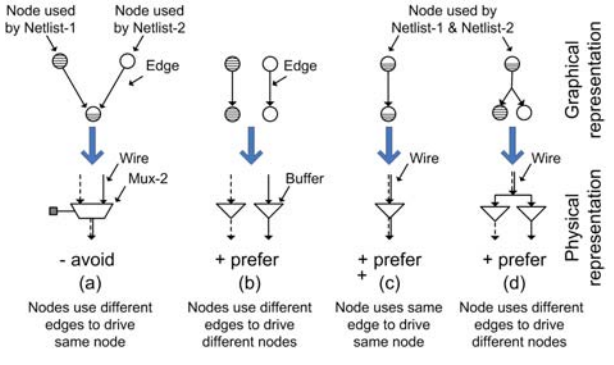


Fig. 7. Efficient Routing

Figure 7 explains different cases through which efficient wire routing mechanism can be implemented in the pathfinder algorithm. Graphical representation in Figure 7(a) shows a case in which two nodes occupied by NETS of 2 different netlists drive the same node. Figure 7(b) shows a case in which nodes occupied by NETS of different netlists use different edges to drive different nodes. In Figure 7(c), both netlists share the same node and edge to drive a single node. Finally, Figure 7(d) shows a node shared by both netlists which targets different nodes. In order to reduce the total number of switches, the physical representation in Figure 7 suggests that case (a) must be avoided because it increases the number of switches (here a mux-2), whereas case (b), (c) & (d) should be preferred. If more routing resources exist in FPGA architecture, NETS of different netlists can easily find enough free nodes to exploit case (b). For this reason,

in order to create more routing resources, experiments are performed in section 5 with several channel widths.

$$\begin{array}{ll} \text{(Normal)} & \text{Cost(n)} = \text{CongestionCost(n)} \\ \text{(Avoid)} & \text{Cost(n)} = (1 + \text{Factor}) * \text{CongestionCost(n)} \\ \text{(Prefer)} & \text{Cost(n)} = (1 - \text{Factor}) * \text{CongestionCost(n)} \\ & \text{Where } 0 \leq \text{Factor} \leq 0.99 \end{array} \quad (2)$$

The efficient routing scenarios shown in Figure 7 must be integrated into the pathfinder algorithm. For this purpose the cost function of a node is modified in a way similar to the time-driven cost function [4]. A particular routing is avoided or preferred by increasing or decreasing the cost of a node. If a NET is routed from current node to next node, the cost of next node can be calculated with the formulas shown in Equation 2. The cost of a node normally depends on its congestion cost. The increase or decrease in the node cost is controlled by a constant "Factor". The value of this factor ranges between 0 and 0.99. If an FPGA architecture has limited routing resources, a maximum value of "Factor" might not allow the routing algorithm to resolve all congestion problems. Consequently, the value of "Factor" is gradually decreased if the routing solution does not converge after a few routing iterations. All experimental results shown in section 5 use the maximum value of "Factor". The maximum routing iteration count is set to 30, and the value of "Factor" decreases if routing does not converge within the first 15 iterations.

Multiple netlists can be routed on an FPGA architecture, either in sequence or in parallel. Sequential routing of netlists in a particular order is called here "Netlist by Netlist" routing. Each routed netlist saves information about used nodes and edges. Later, next netlist uses this information to perform efficient routing by giving preference to some nodes over others. Simulation tests are performed with netlists sequenced in different orderings (i.e. netlists ordered in ascending or descending order, according to their size, channel width, wire utilization and few random orders). It has been noted that, with limited routing resources, the total area of an ASIF varies when netlists are arranged in different sequences. However this difference in area becomes negligible as routing resources increase.

In order to avoid the dependence on a particular order of netlists, netlists are routed in parallel. Two parallel techniques are tried; "Iteration by Iteration" (routing-iterations of different netlists are routed in a sequence) and "Net by Net" (NETS of different netlists are routed in a sequence). But both techniques give worse results than the best "Netlist by Netlist" ordering. This is because in parallel routing techniques, routing of all netlists remains incomplete simultaneously. To avoid congestion, NETS keep on changing their path in different iterations. A path that is chosen by netlist-2 because it is used by netlist-1, might not eventually be used by netlist-1, but still remains in use by netlist-2. Thus, due to inaccurate routing information, both parallel methods end up taking more switches than the best "Netlist by Netlist" results.

Table 1. Netlist Version-lut4,-lut3,-lut2

Index	Netlist Name	Mult 16x16 (Common	Add 20x20 Elements)	In	Out	LUT-4 Version-lut4	LUT-3 Version-lut3	LUT-2 Version-lut2
1.	cf_fir_24_16_16	25	48	418	37	2149	2149	2149
2.	rect2polar	0	52	34	40	1328	1497	3657
3.	polar2rect	0	43	18	32	803	803	2280
4.	cfft16x8	0	26	20	40	1511	1937	3282
5.	fm_receiver	1	20	10	12	910	1004	1266
6.	fm_receiver2	1	19	9	12	1308	1508	1987
7.	cf_fir_7_16_16	8	14	146	35	638	638	638
8.	lms	10	11	18	16	940	965	970
9.	rs_encoder	16	16	138	128	537	537	537
10.	cf_fir_3_8_8	4	3	42	18	159	159	159
	Maximum	25	52	418	128	2149	2149	3657

Table 2. Netlist Version-Gates

Index	Netlist Name	Mult 16x16	Add 20x20	In	Out	mux2	inv	an12	nor2	buf	and2	nmux2	nand2	zero	one	flip-flop
1.	cf_fir_24_16_16	25	48	418	37	0	0	400	0	0	0	0	0	25	10	2116
2.	rect2polar	0	52	34	40	962	151	68	202	40	94	0	356	1	18	1113
3.	polar2rect	0	43	18	32	736	17	5	2	32	0	1	5	31	14	741
4.	cfft16x8	0	26	20	40	165	587	47	883	26	75	256	735	37	10	774
5.	fm_receiver	1	20	10	12	0	124	374	363	0	35	10	408	22	17	374
6.	fm_receiver2	1	19	9	12	0	205	42	868	234	40	1	566	21	19	248
7.	cf_fir_7_16_16	8	14	146	35	0	0	128	0	0	0	0	0	8	11	619
8.	lms	10	11	18	16	0	40	65	5	16	13	10	40	22	10	912
9.	rs_encoder	16	16	138	128	0	136	0	8	128	136	0	8	33	16	128
10.	cf_fir_3_8_8	4	3	42	18	0	0	32	0	0	0	0	0	8	3	148
	Maximum	25	52	418	128	962	587	400	883	234	136	256	735	37	19	2116

5 Experimentation and Analysis

5.1 Area Model

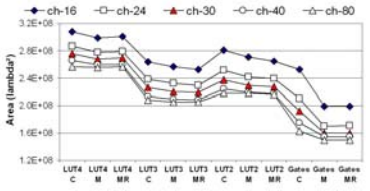
A generic area model is used to calculate the area of FPGA and various ASIFs. The area model is based on a reference FPGA architecture shown in Figure 1. Areas of SRAMS, multiplexers, buffers, flip-flops and other gates are taken from a symbolic standard cell library which works on unit Lambda. The total area is computed as the sum of area of switch boxes, connection boxes, buffers and logic blocks (CLBs, HBs). The area model also reports the total number of routing wires used for routing all netlists. When an ASIF is generated, all unused routing resources are removed. With the removal of switches, wires are connected with one another to form long wires. Appropriate amount of buffers are added to reduce these long wires. For example, in case of a LUT-4 based ASIF, a buffer is added for every wire of length 16 (spanning 16 CLBs) and for every output wire of a block driving 8 wires (i.e. having a fanout of 8).

5.2 Benchmark Circuits

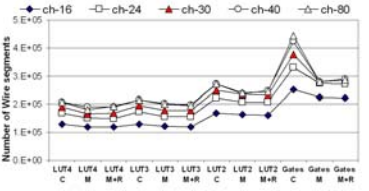
A set of opencores circuits are used for experimentations. The synthesized netlists are passed through the software flow (described in section 2) and converted into .NET format. Four different versions of netlists are generated (as shown in Table 1 & 2). The number of multipliers, adders and IOs remain the same in all the four netlists; however they differ by the type of logic blocks (either by the size of LUTs used or by the use of gates). Finding the right mix of gates is not always an easy task. This work attempts three different set of gates; the best set of gates is selected. The three set of gates are (i) All the netlists are synthesized using the minimum type of gates (which include only nand gates, zero gates, buffers and flip-flops). (ii) All the netlists are synthesized using maximum type of gates i.e. any gate required from the standard-cell library (iii) All the netlists are synthesized using only commonly used gates required by the netlists; this set of commonly used gates are manually selected from the synthesis of netlists done in (ii). The ASIFs generated using the commonly used gates gives the minimum area results as compared to the ASIFs generated using minimum and maximum type of gates. Three versions of the 10 opencores circuits using LUT-2, LUT-3 and LUT-4 (along with multipliers, adders and IOs) are presented in Table 1. Table 2 shows the 4th version of the same netlists with the commonly used gates. Although the netlists use various sizes of multipliers and adders, their common maximum size is selected for an FPGA/ASIF. It can also be noticed that due to the nature of the netlists (due to the number of flip-flops), there is no major difference between the LUT-4 version and LUT-3 version.

5.3 Results

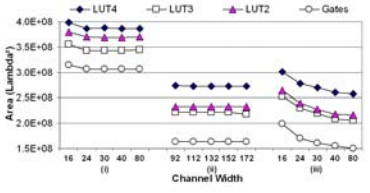
ASIF generation techniques are applied on 4 different versions of opencores benchmark circuits shown in Table 1 and 2. Experiments are performed with different floor-planning techniques, and with various routing resources. Figure 8(a) shows area comparison of different ASIFs, and Figure 8(b) shows the number of wire segments used by different ASIFs. The X-axis in both of these figures shows ASIFs generated using 4 versions of the netlist, with different floor-planning techniques. The letters 'C', 'M' and 'MR' represent the type of floor-planning used (as described in section 3); 'C' stands for Column placement; 'M' stands for Move operation, where blocks can be jumped or translated; 'MR' stands for both Move and Rotate operation. Y-axis in Figure 8(a) shows the total area of ASIFs in Lambda square. Y-axis in Figure 8(b) shows the total number of wire segments (routing tracks) used by different ASIFs. It can be noted in Figure 8(a) that the area of an ASIF decreases as the channel width (routing resource) increases. This is because, the availability of more routing resources increase the probability to prefer case 7(b), (c) and (d), which in turn increase the probability to avoid 7(a); thus less number of switches are used. However as the channel width increases, the number of wires also increases (shown in Figure 8(b)). The number of wires can play a pivotal role in the area of ASIF if it dominates the logic area. This can happen if an ASIF is generated for a large number of netlists, the layout of an ASIF is generated in a smaller process technology, or different blocks of an



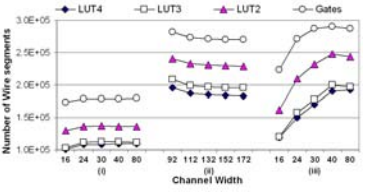
(a) Area comparison for different ASIFs with varying channel widths



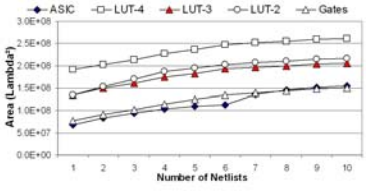
(b) Wire count comparison for different ASIFs with varying channel widths



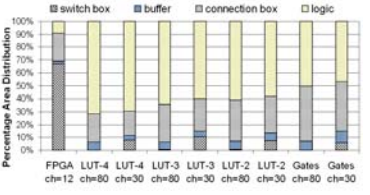
(c) Area comparison for ASIFs with different Placement/Routing(P/R) (i) Normal P/R (ii) No wire sharing (iii) Efficient P/R



(d) Wire count comparison for ASIF with different Placement/Routing(P/R) (i) Normal P/R (ii) No wire sharing (iii) Efficient P/R



(e) ASIC vs. ASIFs with varying LUT sizes and number of netlists



(f) Percentage area distribution for FPGAs and ASIFs

Fig. 8. Experimental Results

ASIF are designed in full-custom. In such a case, an ASIF with smaller channel width can give a trade-off solution.

From Figure 8(a), it can be noticed that ASIF areas come in the following order : ASIFArea(LUT-4) > ASIFArea(LUT-2) > ASIFArea(LUT-3) > ASIFArea(Gates). This effect is mainly due to the area of logic blocks in ASIFs. Because of the nature of netlists (shown in Table 1), the maximum number of LUT-4 in version-lut4 are same as that of LUT-3 in version-lut3; thus ASIF using version-lut3 is smaller than ASIF using version-lut4. On the other hand, the number of LUT-2 in version-lut2 are so much greater than number of LUT-3 in version-lut3 that (despite LUT-2 being smaller in size than LUT-3) the total logic area in an ASIF using version-lut2 bypasses the logic area occupied by the ASIF using version-lut3. The total logic area of ASIF in version-gates is much smaller than logic area in version-lut3; however a gates based ASIF uses more routing resources than used by LUT-3 based ASIF. But, eventually gates based ASIF gives the best result in terms of area.

The effect of floor-planning can also be noticed in Figure 8(a). ASIFs for LUT-4, LUT-3 and LUT-2 based netlists show only a slight area gain (between 0.5% and 6%) with Move ('M') and Move/Rotate ('MR') floor-planning over Column floor-planning ('C'). However, in case of gates version, a considerable gain (between 8% to 21%) is achieved with Move floor-planning 'M'. This is because the gates version of netlists uses 13 different types of blocks, whereas LUT based version uses only 3 types of logic blocks. The floor-planning of 13 types of blocks in columns greatly restricts the placements of different instances of a netlist, which in turn requires more routing resources to connect with one another. The Block Move/Rotate floor-planning did not show much better area results over Move floor-planning.

Figure 8(c) and (d) compare three different ASIF generation techniques. The comparison is done for 10 netlists with FPGA floor-planning attained through block Move/Rotate operation. These techniques differ in the placement and routing of netlists on FPGA, before being reduced to ASIF. The section (i) in Figure 8(c)(d) uses normal placement and routing; (iii) uses efficient placement and routing. Section (ii) uses normal placement; whereas routing of each netlist is performed using different channel tracks (i.e. no switch boxes and routing tracks are shared amongst different netlists; only logic blocks and connection boxes are shared). Consequently, the routing channel width used by (ii) is the sum of channel widths used by all the netlists; whereas (i) and (iii) share the routing channels for all netlists. The area of an ASIF shown in (iii) is 33 to 51 percent better than (i), and 6 to 9 percent better than (ii) for four different netlist versions at maximum channel widths. The slight improvement of (iii) over (ii) is due to efficient placement, which places several instances of different netlists on the same blocks of the architecture; later efficient routing facilitates the use of common switches and wires to drive the same blocks; thus reducing the number of switches. Figure 8(d) shows that the number of wires used in (i) are very less as compared to (ii) and (iii); the wires used in (ii) remains relatively constant; the wires used in (iii) change with different channel widths. In an ASIF for large number of netlists, the wires used by (ii) can become so huge that the overall area of chip becomes wire dominant. In such a case, an ASIF shown in (iii) can give a trade-off solution using smaller channel widths.

Figure 8(e) compares ASIC and ASIF with various number of netlists (the order of netlists in Table 1 is respected). The best achieved ASIFs (generated with maximum channel width and Block Move/Rotate floor-planning) are compared here with the sum of areas of ASICs of the netlists. The X-axis represents the number of netlists used in the experiment; 1 means that only "cf_fir_24_16_16" is used, 2 means that "cf_fir_24_16_16" and "rect2polar" are used, and so on. The Y-axis presents the area in symbolic units (Lambda square). If area occupied by routing wires is not dominant, a LUT-3 based ASIF for 10 netlists is only 24% larger than the sum of areas of ASICs, whereas a gates based ASIF is 3% smaller than the sum of areas of ASICs for 10 netlists. The sum of ASIC areas is slightly smaller than for the gates based ASIF with 7 or less netlists. Comparison with an FPGA (not shown in Figure) shows that a LUT-3 based ASIF is 85% and

a gates based ASIF is 89% smaller than a Heterogeneous LUT-3 based, single driver, uni-directional FPGA using a channel width of 12.

Figure 8(f) shows the percentage area distribution in FPGA and ASIFs. In a LUT-3 based FPGA, using a channel width of 12, only 9.3% of the area is taken by logic area (CLBs, Multipliers and Adders), whereas the remaining area is taken by the routing area (switch boxes, connection boxes and buffers). In ASIFs, the routing area is decreased down to such an extent that the logic area occupies a very important percentage of the total area; in gates version of ASIF, logic area takes 50% of area for a channel width of 80; in LUT-3 version of ASIF, logic area takes 64% of area for a channel width of 80.

6 Mapping Application Circuits on an ASIF

An ASIF retains limited routing resources of an FPGA to execute only a known set of circuits. However, these limited resources in an ASIF might be exploited to execute any new circuit. Though an ASIF is unable to map the same variety of circuits as an FPGA does; it might be used to map some elementary circuits, or slightly modified versions of the same circuits used in the generation of an ASIF. One of the problems to map a new netlist on an ASIF is to find an appropriate placement with sufficient routing paths to route all signals. It is not sure if such a placement solution exists in an ASIF or not. Even if few routable placement solutions exist, the currently used heuristic based placement algorithms do not guarantee to find a placement solution from a solution space that contains few or maybe only one placement solution. In this regard, we attempted the simulated annealing based placement algorithm to re-place the same set of netlists on an ASIF which are used in its generation. The placement algorithm needs to find a placement of the new netlist without increasing the "Driver Count" cost (see section 4.1) of the ASIF. However, the algorithm is unable to find a routable placement solution in limited time.

This section presents an elementary version of a branch-and-bound [9] based placement algorithm which is able to re-place the same set of circuits on their own ASIF. The same algorithm is later used to map a simple new netlist on an ASIF. The main idea is to attempt placement of all instances of a netlist on all possible BLOCK positions of the ASIF. Although, limited routing resources decrease inter-BLOCK connectivity in an ASIF; attempting all possible combinations can however be an extremely time consuming task. For that reason, the placement combinations for instances of a netlist are systematically decreased. The algorithm also ensures that the placement of a driver and its receiver instances are only attempted on interconnected BLOCKS.

The algorithm initially extracts BLOCK connectivity from an ASIF (i.e. which driver PINS of a BLOCK are routable to which receiver PINS of a BLOCK). Connectivity information is also extracted for the netlist that is attempted to be mapped on the ASIF. The connectivity information of the ASIF and the netlist are used to generate an exhaustive list of BLOCKS (called here as maplist) on which each instance of a netlist can be mapped. For each instance 'i', the

Table 3. Placement Execution Time (in seconds)

Index	Netlist Name	ASIF (LUT-3)	ASIF (Gates)	FPGA (LUT-3)
1.	cf_fir_24_16_16	180	151	445
2.	rect2polar	11589	173	263
3.	polar2rect	182	117	105
4.	cfft16x8	∞	122	340
5.	fm_receiver	922	81	103
6.	fm_receiver2	1083	51	183
7.	cf_fir_7_16_16	66	60	57
8.	lms	∞	357	69
9.	rs_encoder	193	69	58
10.	cf_fir_3_8_8	39	34	11
11.	scan_path(900 (42%) CLBs)	202	-	-

maplist is further reduced by recursively attempting to place its driver and receiver instances on the BLOCKS found in their own maplist and validating if routing connection exists between the placed instance 'i' and its placed driver and receiver instances. A BLOCK is removed from the maplist of an instance 'i' if any of its N^{th} level driver and receiver is unable to find any routable BLOCK position. Once the maplist is finalized for all the netlist instances, a recursive, tree-pruning based algorithm attempts placement of all instances in a netlist over all their possible BLOCK positions on an ASIF.

The proposed algorithm is used to find a routable placement solution for 10 opencores circuits which are used to generate the ASIF. Execution time for mapping a LUT-3 and gates version netlists on their respective ASIFs is shown in Table 3. For the sake of comparison, the execution time of simulated annealing based placement algorithm to map netlists on a LUT-3 based FPGA is also shown in the table. The execution time of this algorithm mainly depends on the number of possible BLOCK positions on which each instance can be mapped. A generalized BLOCK such as a Look-Up-Table tends to give more BLOCK positions to a netlist instance than a specialized BLOCK such as MULT or AND gate etc. For this reason, a LUT-3 based ASIF takes more execution time compared to a gate based ASIF. Due to the same reason, the algorithm is unable to find a routable solution for 2 LUT-3 version netlist in limited time. However, an ASIF with a generalized BLOCK is more suitable to map new circuits. A simple scan-path circuit passing through 42% of the CLBs has also been successfully mapped on an LUT-3 version ASIF for 10 opencores circuits. A scan-path circuit passing through all CLBs is not possible due to the limited inter-CLB connections. The mapping of more complex circuits is still in progress. However, the target netlist modification appears to be inevitable to map new netlists on an ASIF. The netlist modification may include addition of buffers and instance replication to better adapt the new netlist according to ASIF characteristics. Besides, there still remains a lot of room for optimization at the algorithmic and implementation level. These optimizations can further decrease the execution time to map new or modified circuit on an ASIF.

7 Propositions for Layout Generation

No physical layout has been performed yet for an ASIF. All area comparisons are done using an area model. However, few layout generation methodologies are proposed here. After the definition of an ASIF, a behavioral VHDL model of an ASIF can be generated; this model can be placed and routed using any ASIC CAD flow tool. The tool can be restricted to use the same floor-planing as performed earlier by the PLACER. Besides, hardware description of an ASIF can also be integrated as an embedded module into larger designs.

The layout of an ASIF can also be generated using a tile-based layout methodology. However due to irregular routing network of ASIFs, there can be large number of tiles with different sizes and characteristics. Consequently, the width of column of tiles can be reduced only to the maximum width of any tile in that column; and the height of row of tiles can be reduced only to the maximum height of any tile in that row. In such a case, instead of wasting free area in smaller tiles, some routing connections can be wisely added to increase the flexibility of an ASIF. This added flexibility in ASIFs might increase the probability of mapping a new circuit on it.

The layout of an ASIF can be facilitated by uniformly reducing an FPGA to an ASIF (i.e. all tiles in an FPGA or a group of FPGA tiles are reduced in a uniform manner). Uniform reduction of an FPGA means that if a switch is removed from a tile, the same switch is removed from all the tiles in that group. In this way, all reduced FPGA tiles in a group can be easily abutted together to generate an ASIF layout. Such an ASIF might not be as area efficient as presented in section 5; however this ASIF may lead to easier layout generation, easier full-custom layout of the limited number of tiles, and maybe (due to its uniformity) easy mapping of new application circuits. The amount of area losses incurred with such a layout generation technique, and its effects on the flexibility of an ASIF needs to be explored.

8 Conclusion

This paper presents an Application Specific Inflexible FPGA (ASIF) using heterogeneous logic blocks. The floor-planning of different hard blocks is performed to achieve better area gains. A Heterogeneous-ASIF for a group of 10 opencores circuits is compared with an FPGA and with the sum of areas of ASICs for the given set of 10 netlists. It has been shown that a Heterogeneous ASIF containing multipliers, adders and LUT-3 is 85% smaller than a LUT-3 based FPGA, and only 24% larger than the sum of areas of standard-cell based ASICs for 10 opencores netlists. The Heterogeneous ASIF containing multipliers, adders and hard logic gates is 89% smaller than a LUT-3 based FPGA, and 3% smaller than the sum of ASICs for the 10 opencores netlists. A new CAD flow is also presented which can map new application circuits on an ASIF. In future we intend to improve this CAD flow to support mapping of more complex new circuits on an ASIF. We would like to find area-delay tradeoffs for ASIFs. We also intend to explore different automatic layout generation techniques for ASIFs.

References

1. Altera, http://www.altera.com
2. Marquart, A., Betz, V., Rose, J.: Using cluster-based logic block and timing-driven packing to improve FPGA speed and density. In: International symposium on FPGA, Monterey, pp. 37–46 (1999)
3. Berkeley Logic Synthesis and Verification Group, University of California, Berkeley. Berkeley Logic Interchange Format (BLIF), http://vlsi.colorado.edu/~vis/blif.ps
4. Betz, V., Marquardt, A., Rose, J.: Architecture and CAD for Deep-Submicron FPGAs (January 1999)
5. Compton, K., Hauck, S.: Automatic Design of Area-Efficient Configurable ASIC Cores. IEEE Transaction on Computers 56(5), 662–672 (2007)
6. Hutton, M., Yuan, R., Schleicher, J., Baeckler, G., Cheung, S., Chua, K., Phoon, H.: A Methodology for FPGA to Structured-ASIC Synthesis and Verification. In: DATE, March 2006, vol. 2, pp. 64–69 (2006)
7. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimisation by Simulated Annealing. Science 220(4598), 671–680 (1983)
8. Kuon, I., Rose, J.: Measuring the Gap Between FPGAs and ASICs. In: FPGA 2006, pp. 21–30 (Februray 2006)
9. Lawler, E.L., Wood, D.E.: Branch-and-bound methods: A survey. Operations Research 14, 699–719 (1966)
10. Lemieux, G., Lee, E., Tom, M., Yu, A.: Directional and Single-Driver Wires in FPGA Interconnect. In: ICFPT (2004)
11. McMurchie, L., Ebeling, C.: Pathfinder: A Negotiation-Based Performance-Driven Router for FPGAs. In: Proc. FPGA 1995 (1995)
12. Parvez, H., Marrakchi, Z., Mehrez, H.: ASIF: Application Specific Inflexible FPGA. In: ICFPT 2009 (2009)
13. Pistorius, J., Hutton, M., Schleicher, J., Iotov, M., Julias, E., Tharmalignam, K.: Equivalence Verification of FPGA and Structured ASIC Implementations. In: FPL 2007, pp. 423–428 (August 2007)
14. Sentovich, E.M., et al.: Sis: A system for sequential circuit analysis. Tech. Report No. UCB/ERL M92/41, University of California, Berkeley (1992)

Reconfigurable Communication Networks in a Parametric SIMD Parallel System on Chip

Mouna Baklouti, Philippe Marquet, Jean Luc Dekeyser, and Mohamed Abid

INRIA, University of Lille, France
CES, University of Sfax, Tunisia
{mouna.baklouti,philippe.marquet,jean-luc.dekeyser}@lifl.fr,
mohamed.abid@enis.rnu.tn

Abstract. The SIMD parallel systems play a crucial role in the field of intensive signal processing. For most the parallel systems, communication networks are considered as one of the challenges facing researchers. This work describes the FPGA implementation of two reconfigurable and flexible communication networks integrated into mppSoC. An mppSoC system is an SIMD massively parallel processing System on Chip designed for data-parallel applications. Its most distinguished features are its parameterization and the reconfigurability of its interconnection networks. This reconfigurability allows to establish one configuration with a network topology well mapped to the algorithm communication graph so that higher efficiency can be achieved. Experimental results for mppSoC with different communication configurations demonstrate the performance of the used reconfigurable networks and the effectiveness of algorithm mapping through reconfiguration.

Keywords: Reconfigurable architectures, communication networks, SIMD processors, parallel architectures, FPGA.

1 Introduction

Embedded image or signal processing applications require high performance systems and highly integrated implementation solutions. They are mostly developed on embedded systems with high performance processing units like DSP or Single Instruction Multiple Data (SIMD) processors. While SIMD systems may have been out of fashion in the 1990s, they are now developed to make effective use of the millions of transistors available and to be based on the new design methodologies such as IP (Intellectual Property) reuse. Nowadays we have a great variety of high capacity programmable chips, also called reconfigurable devices (FPGAs) where we can easily integrate complete SoCs architectures for many different applications. Due to the inherent flexibility of these devices, designers are able to quickly develop and test several hardware(HW)/software(SW) architectures. In this paper, we used Altera [16] reconfigurable devices to implement the mppSoC (massively parallel processing System on Chip) architecture and get experimental results. Our contribution to SIMD on-chip design domain

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 110–121, 2010.

consists in the implementation at an RTL abstraction level of a parameterized system with flexible reconfigurable networks: one dedicated to neighboring communications and one to assure point to point communications. Reconfiguration is accomplished through instructions. The designer can choose the appropriate mppSoC configuration to execute a given application.

This paper is structured as follows. Section 2 presents several SIMD architectures and focuses on the implementation of their interconnection networks. Section 3 briefly introduces the mppSoC platform. Section 4 details the integration of reconfigurable communication networks in the mppSoC design. Section 5 discusses some algorithms varying the used interconnection network. Finally, Section 6 summarizes the contribution with a brief outlook on future work.

2 Related Works

Due to rapid advancement in VLSI technology, it has become feasible to construct massively parallel systems, most of them are based on static interconnection structures as meshes, trees and hypercubes. Typically, an SIMD implementation [3] consists of a Control Unit, a number of processing elements (PE) communicating through an interconnection network (ICN), which often are custom-made for the type of application it is intended for. If the ICN does not provide direct connection between a given pair of processors, then this pair can exchange data via an intermediate processor. The ILLIAC IV [5] used such an interconnection scheme. The ICN in the ILLIAC IV allowed each PE to communicate directly with 4 neighboring PEs in an 8x8 matrix pattern. So, to move data between two PEs, that are not directly connected, the data must be passed through intermediary PEs by executing a programmed sequence of data transfers [4]. This can lead to excessive execution time if more irregular communications are needed. The same problem of communication bottleneck is encountered with other architectures like [10] and [13], which may cause a significant increase in the cycle count of the program. Other new SIMD architectures have been proposed [7] [14] but they don't perform irregular communications since they integrate only a neighborhood ICN. A nearest neighbor ICN is good for applications where the communications are restricted to neighboring PEs. However, there are several problems which require non local communications. Some massively parallel machines [15] [8] have a scheme to cover such communication patterns. But, they integrate a static ICN. The problem is that different applications might have different demands for the architecture. Several reconfigurable SIMD architectures have appeared. The Morphosys [12] proposed dynamically reconfigurable SoC architecture. It contains a sophisticated programmable tri-level ICN. This gives efficient regular applications, but unfortunately non neighbours communications seem to be tedious and time consuming. RC-SIMD [2] is a reconfigurable SIMD architecture based on two segmented unidirectional communication busses. It is powerful in term of neighboring communications even between distant-PEs, however not efficient for irregular communications. A dynamically reconfigurable SIMD processor array is also described in [1]. It includes a two-dimensional array

of 64x64 identical PEs. It is dedicated to compute programmable mask-based image processing with only support of small local neighboring operations.

Previous proposals appear incomplete from an application perspective. While some architectures are powerful in term of inter-PE communication and some of them are reconfigurable, they can not perform non local operations efficiently. These parallel architectures are not flexible nor scalable to support the requirements of different data parallel applications. The proposed system extends these works by using flexible reconfigurable communication networks based on parametric architecture. In the following, we propose a model of a parallel SIMD system for SoC named mppSoC and we describe its different components.

3 MppSoC Design

MppSoC is an SIMD massively parallel processing System on Chip built within nowadays processors. It is composed of a number of 32-bit PEs, each one attached to a local memory and potentially connected to its neighbours via a regular network. Furthermore, a massively parallel Network on Chip, mpNoC, is able to perform irregular communications. The whole system is controlled synchronously by an Array Controller Unit (ACU). The ACU and PEs are built from the same processor IP (the miniMIPS [17] in this work). The ACU is a complete processor, having 5 stages of pipelining, whereas the PE is a reduced one having only the 3 last execution units. This processor building methodology has a significant gain allowing the integration of a large number of PE on a chip. The ACU transfers parallel arithmetic and data processing instructions to the processor array, and handles any control flow or serial computation that cannot be parallelized. The overall structure of the mppSoC architecture and pipeline is shown in Fig. 1. The pipelined mppSoC architecture has been described in previous papers [11]. Since mppSoC is designed as a parametric architecture, the designer has to set some parameters in order to generate one configuration on FPGA such as the number of PEs, the memory size and the topology of the neighborhood network if it exists.

The mppSoC system is programmed by a single instruction stream partitioned into sequential and parallel instructions. The mppSoC instruction set is derived from the IP processor instruction set used in the design which is modified by adding parallel instructions. Some specific instructions control the two networks, allowing data transfer. Below, we will detail the mppSoC networks.

4 MppSoC Communication Networks

In order to improve the parallel system performances, and to satisfy the requirements of different data parallel applications we propose flexible and reconfigurable communication networks. Availability of such communication is critical to achieve high performance. MppSoC networks are partitioned into two types: regular and irregular networks. The designer can use none, one or both routers to construct the needed mppSoC configuration.

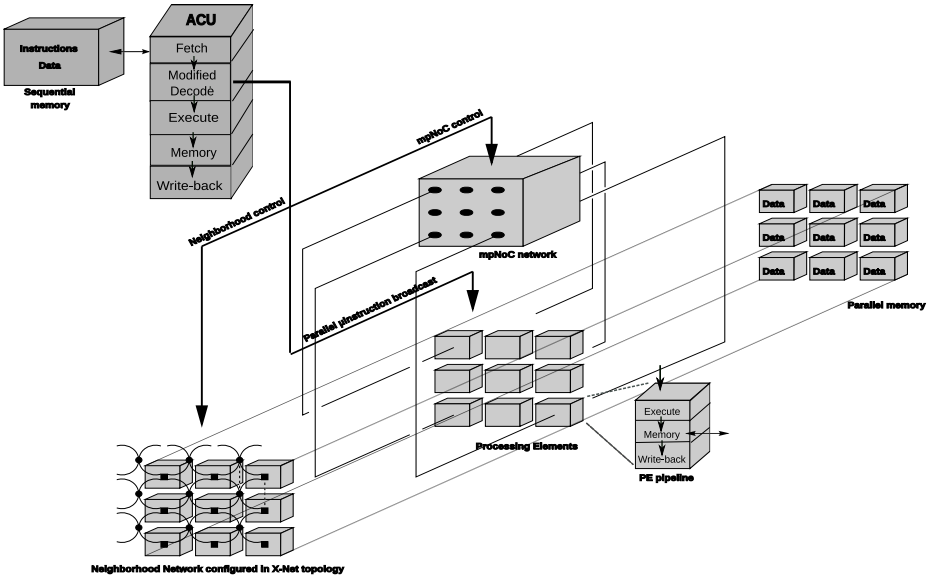


Fig. 1. MppSoC Design

4.1 Reconfigurable Massively Parallel Network on Chip

The mpNoC IP is an irregular network performing point to point communications. It accomplishes three main functions in the mppSoC system. Firstly, the mpNoC is able to connect, in parallel, any PE with another one. Secondly, the mpNoC could connect the PEs to the mppSoC devices. Thirdly, it is able to connect the ACU to any PE. The mpNoC allows parallel I/O transfers solving the need of a high bandwidth required by data parallel applications. It consists mainly of a Mode Manager responsible of establishing the needed communication mode and an interconnection network assuring data transfer. MpNoC input and output ports are connected to switches controlled by the ACU, as shown in Fig. 2. Theses switches allow to connect either the PEs or the I/O devices and the ACU to the mpNoC, depending on the chosen communication mode issued from the ACU to the Mode Manager. In fact, the communication mode could be set at runtime through a mode instruction, when executed the corresponding connections are activated. The mpNoC reconfiguration is performed in two levels. The first level is at compile time where the designer chooses to integrate mpNoC with a selected interconnection network. The second level is during real-time where the communication protocol between the different devices can be altered based on the mode instruction. The proposed mpNoC is scalable according to the number of PEs connected to the network. It integrates an interconnect, responsible of transferring data from sources to destinations, which may be of different types (bus, crossbar, multi-stages, etc). Different networks are provided in a library needed when designing mppSoC. Allowing the designer

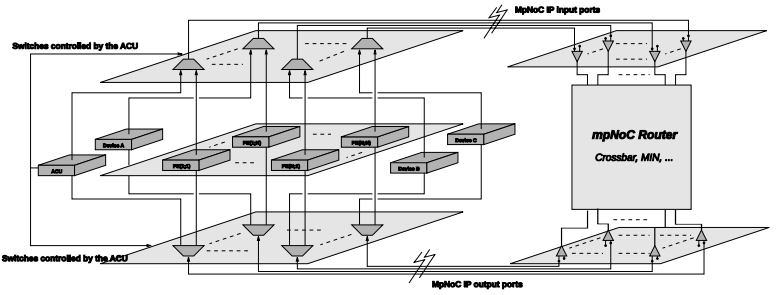


Fig. 2. mpNoC integration into mppSoC

to choose the internal network increases run-time performances. The interconnect interface is generic in order to support a configurable size (4x4, 32x32 for example). While targeting an mpNoC integration into mppSoC, the number of mpNoC sources and destinations is equal to the number of PEs. When using mpNoC, we integrate also a controller to ensure synchronization between PEs since it is the most SIMD important feature. The mpNoC controller verifies if data transferred by the sender is received by the corresponding receiver. At the end of the transmission, the mpNoC controller sends an acknowledgment to the ACU in order to continue executing instructions. The ACU does not issue a new instruction until the communication occurs.

4.2 Reconfigurable Neighbourhood Network

In most data parallel applications, processors work on neighboring data and need to communicate fast among themselves for high performance. Thus a neighbourhood network is also integrated in the mppSoC system. Most common data parallel algorithms need a broad range of processor connectivities for efficient execution. Each of these connectivities may perform well for some tasks and badly for others. Therefore, using a network with a selective broadcast capability, various configurations can be achieved, and consequently, optimal performance can be achieved. We propose different regular network topologies: linear array, ring, mesh, torus, and xnet (a two dimensional toroidal with extra diagonal links). To change from one topology to another the programmer has to use the mode instruction with the appropriate topology value in order to assure the appropriate connections. Five values are defined to specify the provided 5 topologies. In fact, if selected, the neighborhood network with a given topology is generated at compile time. Then the different neighboring links could be changed at run-time. To achieve a high reusability and reconfigurability, the neighborhood network consists of routing elements or switches that are connected to the PEs. Their interface is equipped with 9 ports or interfaces: north, east, south, west, north east, north west, south east, south west and local. The local one is the port that communicates to its attached PE. The switcher activates the appropriate port to transfer data to the needed destination. The way it forwards the data depends on

the executed communication instruction. The network is controlled by the ACU through mode instruction. At every mode instruction, the switches determine a new network topology for the system. In a sense, this is an extension of the SIMD paradigm because for each instruction, the data manipulating the connectivity are controlled in exactly the same way as the data for computing. Circuit switching was adopted to establish the connection, and as a result, a very long path can be established in a large system. In the regular communication, we can specify the distance between PEs on the same row or column or diagonal (in the case of Xnet). The distance defines the number of paths needed to achieve the communication between the PE sender and the other receiver. Consequently, one PE can communicate, not only to his direct neighbour, but also to more distant PE. The nearest neighbourhood network is different from the mpNoC, since it is faster with a less significant communication overhead. In this case, all PE communications take place in the same direction at the same time. Since each interconnection function is a bijection, this transfer of data occurs without conflicts. Sending and receiving data through networks are managed by different communication instructions that will be described in the following subsection.

4.3 Communication Instruction Set

We identify different instructions to program an mppSoC system: processor instructions, micro instructions and specific instructions which are encoded from the processor instructions. Communication instructions, MODE, SEND and RECEIVE, are examples of specific ones. They may be used in different ways to ensure various functions.

MODE instruction serves to establish the needed communication mode in the case of mpNoC or the network topology in the case of neighborhood communication. It relies on the store SW instruction: SW cst, @ModeManager, where:

- @ModeManager = "0x00009003" for mpNoC and "0x00009004" for the neighborhood network.
- cst is the chosen defined value that corresponds to the mpNoC communication mode or the topology of the neighborhood network.

The mode values are defined in the mppSoC configuration file. After setting the required interconnection, data transfers will occur through SEND and RECEIVE instructions.

SEND instruction serves to send data from the sender to the corresponding receiver, based on the SW instruction: SW data, address. The 32bits address can be partitioned in different fields depending on the established mpNoC mode. It contains in case of:

- PE-PE Mode: the identity of the PE sender, the identity of the PE receiver and the PE memory address;
- PE-ACU Mode: the identity of the PE sender and the ACU memory address;
- ACU-PE Mode: the identity of the PE receiver and the PE memory address;
- PE-Device Mode: the identity of the PE sender and the device address;

- ACU-Device Mode: the device address;
- Device-PE Mode: the PE memory address;
- Device-ACU Mode: the ACU memory address.

In the case of regular communication, address contains the distance, the direction and the memory address. There are eight constant direction values, defined in the mppSoC configuration file, that the programmer can specify to denote the eight possible router directions.

RECEIVE instruction serves to obtain the received data, relying on the load memory instruction: LW data, address. It analogously takes the same address field as SEND instruction.

According to his application, the programmer can use all instruction types to satisfy his needs. In the next section, we will present experimental results to validate the proposed mppSoC design.

5 Experiments

In this work, we have executed 3 algorithms: Matrix Multiplication (MM), reduction and picture rotation algorithms, written in MIPS assembly code. In fact, the GNU MIPS assembler has been modified to generate a binary which can be directly integrated in the bit stream of the FPGA mppSoC implementation. The assembly code can be then executed by mppSoC. Each configuration, operating at 50 MHz frequency, is generated in VHDL code and prototyped on the Altera Stratix 2S180 FPGA with 179k logic elements. The proposed system can be efficiently implemented also in any other FPGA family. Different interconnection networks are evaluated and compared with the implemented algorithms. Experimental results were obtained using the ModelSim Altera simulator to simulate and debug the implemented design and the Quartus II which is a synthesis and implementation tool [16] used also to download the compiled program file onto the chip.

5.1 Matrix Multiplication

One of the basic computational kernels in many data parallel codes is the multiplication of two matrices (C=AxB). For this application we have implemented a 64PE mppSoC with mpNoC using two ICN fixed at compile time: a shared bus and a crossbar. As the space of the FPGA HW is limited, 64 is the highest number of PEs that we could integrate on the Stratix 2S180 when integrating the two mppSoC networks. They are arranged in 8x8 grid. The matrices A and B are of size 128x128, partitioned into 8 submatrices A(i,j) and B(i,j), each of size 16x16. Each PE is attached to a 1 Kbyte local data memory. To perform multiplication, all-to-all row and column broadcasts are performed by the PEs. The following code for PE(i,j) is executed by all PEs simultaneously:

```
for k=1 to 7 do
send A(i,j) to PE(i,(j+k) mod 8)  /* East and West data transfer */
for k=1 to 7 do
send B(i,j) to PE((i+k) mod 8,j)  /* North and South data transfer */
for k=0 to 7 do
C(i,j)=C(i,j)+A(i,k)*B(k,j)  /* Local multiplication of submatrices */
```

Fig. 3 depicts the FPGA resource occupation and the execution time results. We validated that the architecture based on the crossbar interconnect IP is more efficient but occupies a large area on the chip. In fact, the full crossbar has the particularity to perform all permutations. However, its space on a chip is quadratic depending on the number of inputs and outputs. On the other hand, busses are relatively simple and the HW cost is small. However, we see that the execution time when using a bus is over two times higher than when using a crossbar. This is due to the fact that in a single bus architecture, one interconnection path is shared by all the connected PEs so that only one PE can transmit at a time. We have also tested the use of the neighborhood network (2D mesh selected at compile time) compared to mpNoC. In the mppSoC program we use the mode instruction with the needed topology value in order to change from one topology to another. Fig. 4 shows a comparison between 4 different ICN. As expected, the architecture based on regular network is the most effective for MM application. These tests show also that the torus network is the most appropriate neighbourhood network.

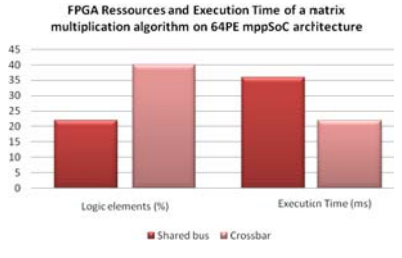


Fig. 3. Experimental results of running a MM algorithm on 64-PE mppSoC

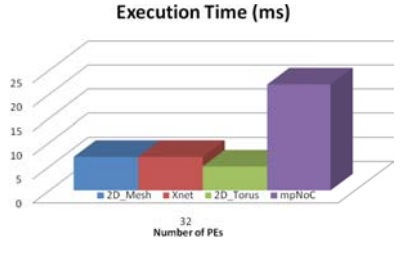


Fig. 4. Execution time results using different communication networks

5.2 Reduction Algorithm

The reduction algorithm [8] presents one basic image processing operations. When reduction computation is conducted in parallel, it is known that the computation can be completed with the minimum number of steps using a binary-tree representation as shown in Fig. 5. To implement the reduction algorithm we use the recursive doubling procedure, sometimes also called tree summing. This algorithm combines a set of operands distributed across PEs [9]. Consider the example of finding the sum of M numbers. Sequentially, this requires one load and M-1 additions, or approximately M additions. However, if these M numbers are distributed across N = M PEs, the parallel summing procedure requires $log_2(N)$ transfer-add steps, where a transfer-add is composed of the transfer of a partial sum to the PE and the addition of that partial sum to the PE's local sum. The described algorithm (sum of 16384 integers) is executed on mppSoC configurations with 64 PEs (2D and linear) and with topologically distinct interconnection networks (mesh/array neighbourhood network and a crossbar based mpNoC). Execution performances are then compared (Fig. 6). We note that the architecture of the used parallel system has also a great impact on the speedup of a given algorithm. We notice that the mppSoC based on the regular network is the most effective for this type of application. In the case of a completely-connected topology, the speedup is six times lower than when using a mesh inter-PE network. The two regular topologies, mesh as well as linear array, give approximately the same execution time. Indeed, the time obtained

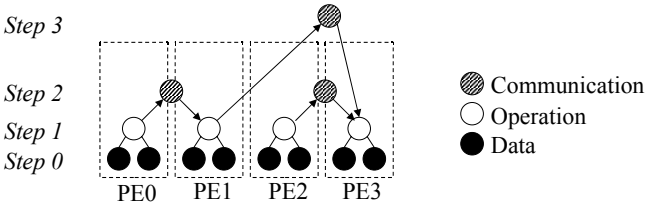


Fig. 5. Parallel reduction computation using four processing elements

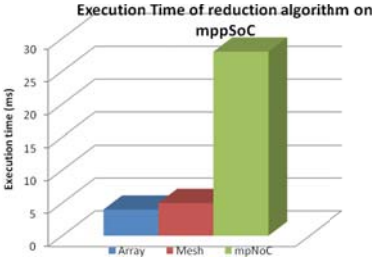


Fig. 6. Execution time on different mppSoC configurations

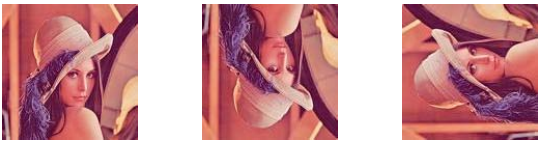


Fig. 7. Picture rotation

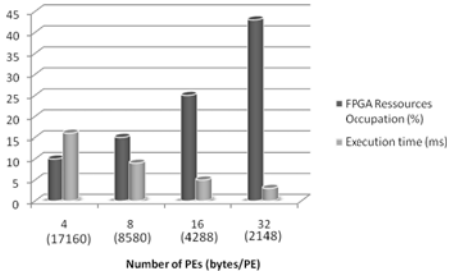


Fig. 8. System scalability/performance running a picture rotation algorithm on a Stratix 2S180

with a linear router is slightly lower than with a mesh router. This is due to the additional communication overhead introduced by the mesh router. So, the linear neighborhood network is the most effective of the reduction algorithm. These different results show also the flexibility of the mppSoC architecture and the high efficiency achieved by establishing a well mapped network topology to one algorithm.

5.3 Picture Rotation

In this algorithm, we realize 17161-pixel picture rotations. The resulting Lena pictures of Fig. 7 were provided by an execution of binary programs on our mppSoC FPGA implementation. The image rotation requires a non homogeneous data movement and I/O throughput requirements. That's why, we have used a crossbar based mpNoC to assure communications and to perform parallel I/O for reading and displaying the resultant image on a VGA screen. So the interconned network in this case is the mpNoC. Selective broadcast capability in this network is enabled by the mode instruction used in the program. We have also tested different number of PEs with variable memory size. Fig. 8 presents synthesis and execution results on various mppSoC designs. We notice a compromise between area and execution time. The results prove the performance of the proposed design. Indeed, when increasing the number of PEs (multiplying by 8) the speedup increases (5 times higher) and the FPGA area is multiplied by a factor of 4 which is an acceptable rate. The results show that the FPGA based

implementation is inexpensive and can easily be reconfigured as new variations on the algorithm are developed.

From all previous experiments we demonstrate the effectiveness of reconfigurable and parametrical networks in a massively parallel system. These networks can perform neighboring as well as irregular communications to satisfy a wide range of algorithm communication graphs. The designer has to make the right choice between the two networks, depending on the application, in order to optimize the whole system performances. In fact, the flexibility and configurability of the mppSoC architecture, in particular its interconnection networks, allow the designer to generate the most appropriate architecture satisfying his needs. It is vital to have a flexible interconnection scheme that can be applied to the system design. The parameterization of the mppSoC is also a key aspect to easily tailor the architecture according to HW as well as SW requirements. The performances of the described system is found better than other architectures. Compared to the ASC processor [10] for example, our mppSoC achieves higher performances (40.48 Mhz with 64 PEs compared to 26.9 Mhz with 50 PEs in the ASC). The mppSoC PEs are 32bits instead of 8bits and contain 3 pipeline stages instead of 4. Compared to the SIMD architecture described in [13], mppSoC is more powerful since it includes a reconfigurable neighboring network rather than a static 2D torus network and it can respond to the irregular communications. In [13], data transfers are based on a global bus which may cause some excessive time since the bus has a limited bandwidth. In term of speed, compared to the H-SIMD architecture [6] mppSoC shows powerful results. In fact, for matrices of size less than 512, the H-SIMD machine is not fully exploited and does not sustain high performance. However, mppSoC is parametric and can be fitted in small as well as large quantity in one FPGA. With a matrix size of 200 the H-SIMD makes 7ms to achieve the computation compared to 5ms obtained when executing multiplication of matrices of size 128 on mppSoC. This comparison prove the high performance and the efficiency of mppSoC.

6 Conclusion

This paper presents a configurable SIMD massively parallel processing system prototyped on FPGA devices. MppSoC can be parameterized to contain several PEs with variable memory size. It is characterized by its reconfigurable communication networks: a neighborhood network and an mpNoC dedicated to irregular communications. Including or not an mpNoC in a given mppSoC design is a trade-off between the cost in term of silicon and the advantage in term of performance and flexibility. To evaluate the mppSoC system we have implemented different sized architectures with various configurations for three representative algorithms. The flexibility of the architecture allows to match the design with the application and to improve the performances and satisfy its requirements. Future work deal with the choice of the processor IP. The ACU for example is not reconfigurable in itself but can be replaceable by an equivalent soft processor or a self designed ACU. The PE could be also obtained by reducing the ACU.

Our aim is to test other processors with the mppSoC design and assure their reconfigurability. The ultimate goal is to develop a complete tool chain facilitating the mppSoC implementation. The nature of the targeted applications may be the decisive element in the design choice.

References

1. Ginhac, D., Dubois, J., Paindavoine, M., Heyrman, B.: An SIMD Programmable Vision Chip with High-Speed Focal Plane Image Processing. Eurasip J. on Embedded Systems 2008 (2008)
2. Fatemi, H., Mesman, B., Corporaal, H., Basten, T., Kleihorst, R.: RC-SIMD: Reconfigurable communication SIMD architecture for image processing applications. J. Embedded Computing 2, 167–179 (2006)
3. Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Trans. Comput. 21, 948–960 (1972)
4. Parhami, B.: Introduction to Parallel Processing: Algorithms and Architectures. Kluwer Academic Publishers, Dordrecht (1999)
5. Michael Hord, R.: The Illiac IV the first supercomputer. Computer Science Press, Rockville (1982)
6. Xu, X., Ziavras, S.G., Chang, T.G.: An FPGA-Based Parallel Accelerator for Matrix Multiplications in the Newton-Raphson Method. In: Yang, L.T., Amamiya, M., Liu, Z., Guo, M., Rammig, F.J. (eds.) EUC 2005. LNCS, vol. 3824, pp. 458–468. Springer, Heidelberg (2005)
7. Schurz, F., Fey, D.: A programmable parallel processor architecture in FPGAs for image processing sensors. In: Proc. IDPT 2007 (2007)
8. Siege, H.J., Wang, L., So, J.E., Maheswaran, M.: Data parallel algorithms. ECE Technical Reports (1994)
9. Stone, H.: Parallel computers. In: Stone, H. (ed.) Introduction to Computer Architecture, Chicago, pp. 327–355 (1975)
10. Wang, H., Walker, R.A.: Implementing a Scalable ASC Processor. In: Proc. International Symposium on Parallel and Distributed Processing, IPDPS. IEEE Computer Society, Los Alamitos (2003)
11. Baklouti, M., Marquet, P., Abid, M., Dekeyser, J.L.: A design and an implementation of a parallel based SIMD architecture for SoC on FPGA. In: Proc. DASIP, Bruxelles, Belgium (2008)
12. Lee, M.-H., Singh, H., Lu, G., Bagherzadeh, N., Kurdahi, F.J., Filho, E.M.C., Alves, V.C.: Design and Implementation of the MorphoSys Reconfigurable Computing Processor. In: VLSI Signal Processing, pp. 147–164 (2000)
13. Kumar, P.: An FPGA Based SIMD Architecture Implemented with 2D Systolic Architecture for Image Processing (2006), SSRN, `http://ssrn.com/abstract=944733`
14. Eklund, S.E.: A Massively Parallel Architecture for Linear Machine Code Genetic Programming. In: Liu, Y., Tanaka, K., Iwata, M., Higuchi, T., Yasunaga, M. (eds.) ICES 2001. LNCS, vol. 2210, pp. 216–224. Springer, Heidelberg (2001)
15. Blank, T.: The MasPar MP-1 architecture. In: Proc. IEEE Compcon Spring 1990, pp. 20–24. IEEE Society Press, San Francisco (1990)
16. Altera, `http://www.altera.com`
17. OpenCores, miniMIPS overview, `http://www.opencores.org/projects.cgi/web/minimips`

A Dedicated Reconfigurable Architecture for Finite State Machines

Johann Glaser, Markus Damm, Jan Haase, and Christoph Grimm

Institute of Computer Technology, Vienna University of Technology, Austria
{glaser,damm,haase,grimm}@ict.tuwien.ac.at

Abstract. For ultra-low-power sensor networks, finite state machines are used for simple tasks where the system's microprocessor would be overqualified. This allows the microprocessor to remain in a sleep state, thus saving energy. This paper presents a new architecture that is specifically optimized for implementing reconfigurable Finite State Machines: Transition-based Reconfigurable FSM (TR-FSM). The proposed architecture combines low use of resources with a (nearly) FPGA-like reconfigurability.[1]

1 Introduction

Wireless sensor networks are applied in numerous fields, including building automation, automotive systems, container tracking, and geological surveillance. To ensure autonomous operation over a long period of time (up to several years), the power consumption of such a node must be very low (some μW). A possible strategy to reduce power consumption is to keep the microprocessor (CPU) in an inactive low power mode as long as possible, and to use a separate, hard-wired Finite State Machines (FSMs) for simple, periodic tasks such as:

- Power management by switching on (or off) peripheral units like sensors, A/D converters and the microprocessor itself.
- Controlling the periodic measurement of sensor values, A/D conversion and decision upon further processing.
- Handling communication at physical and even MAC layer in wake-up receivers.

Although a hard-wired FSM permits significant reduction of power consumption, its application is restricted by the fact that it is not programmable as the firmware is. This restricts the applicability of FSMs to tasks that don't have to be changed later on. The exertion of programmable logic such as an embedded FPGA would permit a tradeoff, but unfortunately introduces a large area and power overhead [1].

In this paper, a new architecture for implementing re-configurable FSMs is presented: Transition-based Reconfigurable FSM (TR-FSM). TR-FSMs have a low logic, wiring and configuration effort and are thereby applicable in ultra-low power applications while offering re-configurability.

[1] This work is conducted as part of the Sensor Network Optimization through Power Simulation (SNOPS) project which is funded by the Austrian government via FIT-IT (grant number 815069/13511) within the European ITEA2 project GEODES (grant number 07013).

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 122–133, 2010.

The rest of the paper is organized as follows: The following section surveys various hardware implementations of FSMs. Then the new approach is introduced and implementation details are described. This is followed by an analysis of the architecture and its advantages compared to FPGAs, and a conclusion.

2 Related Work

For the implementation of an FSM in an ASIC, synthesis tools use logic minimization algorithms to find optimal combinational circuits which realize the state transfer function and the output function. An alternative implementation is to read from a ROM to retrieve the results of the two functions. Both, the input signals and the state signals are used as address inputs. The data output of the ROM is split into the FSM output signals and the next state signals, where the latter are fed back to the inputs through a clocked register [2].

A RAM in read mode is a simple approach to realize a *reconfigurable* FSM. By writing the RAM content to a specific set of data the FSM functions are specified. The disadvantage of the memory approach is the required size. For n_I input signals, n_S state bits and n_O output signals, the memory has to offer $2^{n_I+n_S}$ words with a width of $n_S + n_O$ bits.

The first reconfigurable logic structures were implemented as sum-of-product term structures in PALs and PLAs [3]. Another widely used circuit design for reconfigurable logic are FPGAs. These comprise look-up tables, flip-flops and rich routing resources to universally implement any logic function [3].

The synthesis results for FSMs in an FPGA are suboptimal, according to [4]. Therefore, new synthesis methods using multi-level architectural decomposition and utilization of Block RAMs to reduce the amount of logic resources are introduced. However, this approach still relies on FPGAs and their disadvantages for low-power applications as outlined in [1].

In [5], an unbalanced and unsymmetrical architecture for reconfigurable FSMs is introduced. The FSM is split into a sequential section which implements the state transition function and a logic section which implements the output signals, both connected

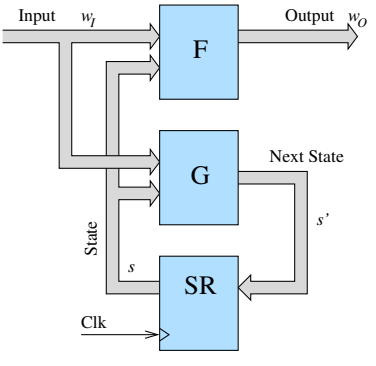


Fig. 1. Generic FSM (SR: state register)

via routing resources. Although this approach achieves an area reduction of 43 % and a power consumption decrease of 82 %, it has overhead due to its concentration on the logic functions for the next state and output signals.

While the previously mentioned approaches implement the full FSM at once, [6] only implements the logic required to calculate the next state (and output signals) for the current state. After every state transition, the internal logic is reconfigured to realize the next state logic for the current state. While this approach greatly reduces the total amount of logic elements by increasing their utilization, the permanent reconfiguration effort is not a low power approach.

3 Transition-Based Reconfigurable FSM

We consider FSMs with n_S state bits, n_I input signals, and n_O output signals, with the respective signals being Boolean. For $w_I \in \{0,1\}^{n_I}$, $s, s' \in \{0,1\}^{n_S}$ and $w_O \in \{0,1\}^{n_O}$, the state transition function and the output function is defined by $G(s, w_I) = s'$ and $F(s, w_I) = w_O$, respectively (see Figure 1). That is, we generally assume Mealy automata, and denote a transition by $s \overset{w_I/w_O}{\longrightarrow} s'$.

The number of possible transitions per state equals 2^{n_I}. Since there are 2^{n_S} possible states, we get an upper bound of $n_T \leq 2^{n_S+n_I}$ for the total number of transitions for such an FSM. However, FSMs in concrete applications have a number of transitions which is considerably lower than this bound. This is especially true for FSMs with many inputs signals, where certain states are only inspecting a subset of these signals. An example is shown in Figure 2, which shows the state transition graph (with outputs omitted) of the FSM "opus" from the LGSynth93 benchmark [7]. Only the state "IOwait" inspects all 5 input signals, while all other states consider at most 2. As a result, the number of transitions in the example is considerably lower (30 transitions) than the possible number regarding the 10 states (320 transitions) or regarding the 4 state bits necessary for state encoding ($2^{n_S+n_I} = 2^{5+4} = 512$ transitions).

For FSMs with such low transition-per-state-ratios, we propose an approach which focuses on the *transitions* rather than on the state transition function G, and call this

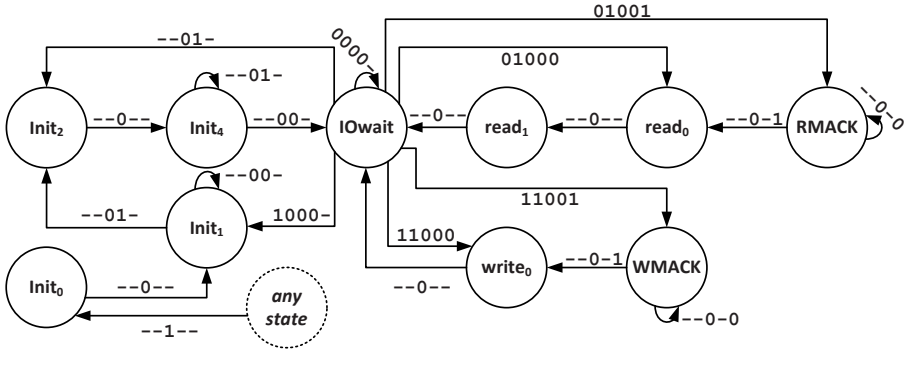


Fig. 2. Example finite state machine

Transition-based Reconfigurable FSM (TR-FSM). Instead of providing a big reconfigurable block for implementing the whole state transition function, we provide several smaller reconfigurable blocks for implementing each transition.

Applying TR-FSM for ASIC or SoC-design is a two-stage process. In the first stage, the necessary resources for the TR-FSM are specified. This can either be an estimation based on FSM prototypes for a certain application class, or it is based on a collection of FSMs which the resulting TR-FSM must be able to be configured to. This can be compared to choosing a sufficiently large FPGA (in terms of slices and BRAM) for the FSMs. However, in TR-FSMs there are more size parameters to be considered, but these parameters depend *directly* on the FSMs in question and not on the state encoding or the quality of logic minimization and synthesis algorithms.

From this specification, the semiconductor chip containing the TR-FSM is produced. In the second stage, this TR-FSM embedded in the chip now can be configured with an actual FSM analogous to configuring an FPGA. However, since the TR-FSM is already structured like an FSM, the synthesis process is considerably less complex.

3.1 Architecture

Note that for the following considerations we omit the FSM outputs. That is, for our purposes a transition is a starting state s, a target state s' and the collection of all input patterns which cause the FSM to change from state s to s', even if the associated outputs are different for each input pattern. The output function F is computed independently, e.g. with a dedicated reconfigurable block. In Section 3.2, we extend the TR-FSM to an architecture which also computes the output in line.

The basis of the proposed architecture is the so called *transition row* (see Figure 3). A state selection gate (SSG) compares the current state to its configured value and enables the transition row. A reconfigurable input witching matrix (ISM) selects a subset of n_T out of the n_I input signals which then serve as the inputs for the input pattern gate (IPG), which is a reconfigurable combinational logic block, e.g. a look-up table (LUT). If activated, the IPG outputs a "1" iff the input pattern matches a certain transition from the recognized state. We also refer to n_T as the *size* of the transition row. Consider for example an FSM which changes from state s to s' on the input patterns `10---` and `0-1--`.

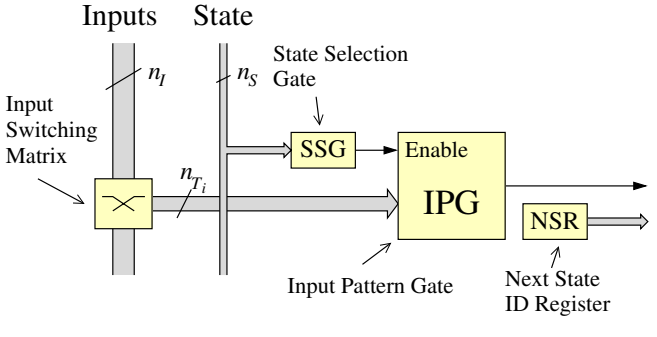


Fig. 3. A transition row

The ISM of the transition row for this transition would select the leftmost 3 signals out of the 5 input signals, and the SSG would check for the encoding of the state s. If s is the current state, the IPG would output a "1" on the input patterns `10-` and `0-1`, indicating that the transition $s \longrightarrow s'$ is active, and would output "0" otherwise.

The reconfigurable *next state ID register* (NSR) of the corresponding transition row is then selected by a multiplexer ("Select") and fed back to the *state register* (SR) as the new state of the FSM.

The overall architecture contains many such transition rows with varying input width (see Figure 4). The number of transition rows and their input widths are derived during the specification stage mentioned above. The output signals are computed in a separate configurable logic block ("Outputs" in Figure 4), which for example can be a LUT or an embedded FPGA IP. Note that the state encoding can be chosen freely with respect to the transition logic, since both, the SSG and the NSR are fully configurable. That is, a different state encoding won't yield any benefits like reducing the number of transition rows needed.

As depicted in Figure 4, the current state is also fed into the next state selection logic block ("Select"). This allows for a very simple treatment of loops, i.e. transitions

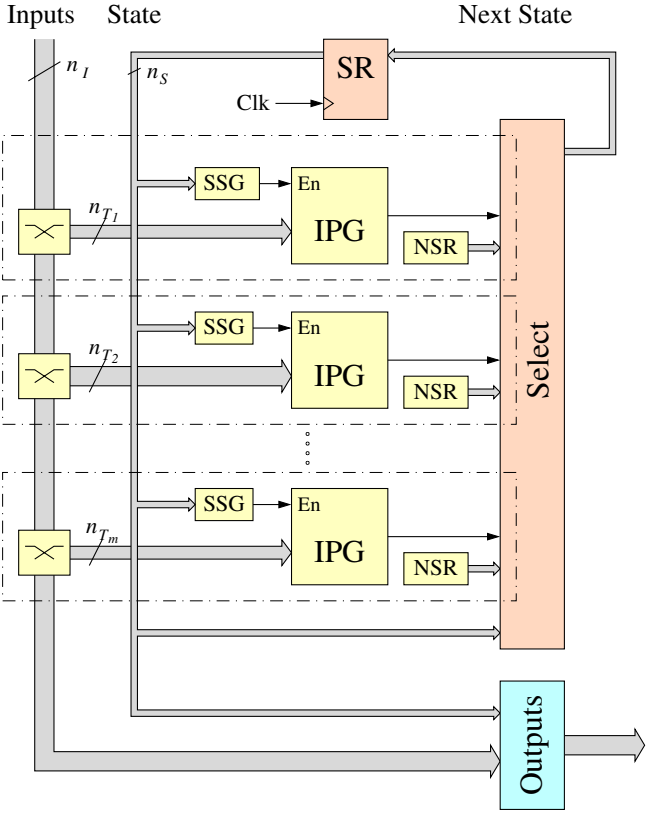


Fig. 4. The overall TR-FSM architecture

where the FSM remains in the same state. In the case that none of the transition rows are activated, the select logic chooses the current state as new state. Therefore, no transition rows have to be used for loops.

In some cases, one of the input signals of an FSM serves as a reset signal (e.g. the third input signal of the "opus" example in Figure 2). If it is set, the FSM resets to its starting state, which we can assume to be encoded by the `0`-vector. For such cases, we propose a special transition row whose ISM selects this signal and directly feeds it to the state register reset input. Since this reset overrides all transition rows, we can exclude the reset signal from all other transition rows.

To summarize, the reconfigurable parts of the architecture are the state selection gate (SSG), the input switching matrix (ISM), the input pattern gate (IPG), and the next state ID register (NSR).

3.2 Including Output Computation

Instead of computing the output function F with a separate reconfigurable combinational logic block, the output associated to a transition can also be included in the transition row as output pattern register (OPR), similar to the NSR (see Figure 5).

The disadvantage is that more transition rows have to be included for this approach. Consider for example two transitions $s \xrightarrow{10/1} s'$ and $s \xrightarrow{11/0} s'$ in an FSM. With the architecture presented in Section 3.1, these two transitions could be covered by one transition row, since the starting state and the target state are identical. Also, the transition row would only need to inspect one input bit by checking for the input pattern `1-`.

If the outputs are considered, two transition rows have to be used here, and both transition rows have to inspect both input signals. Also, loops can't be treated as a default case anymore, since each loop will in general produce a different output. Therefore, each loop now needs to be implemented with a transition row, too.

The possible output functions F computable with this TR-FSM variant are restricted. However, if an FSM rarely outputs different values when changing between two specific states, this approach saves the overly generic reconfigurable block computing F. The overall TR-FSM architecture in this case would look very similar to Figure 4, except that the current state is not fed to the next state selection block. Additionally a second selection block for the output signal replaces reconfigurable combinational logic block for F.

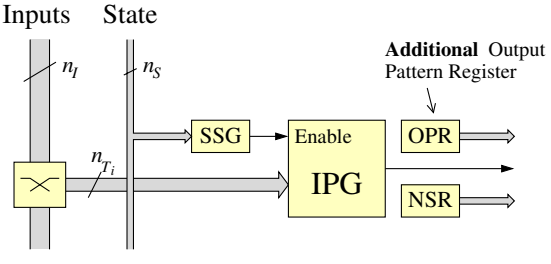


Fig. 5. A transition row with output pattern register (OPR)

3.3 Implementation Details

State Selection Gate (SSG). Since the SSG has to recognize one specific state, it can be implemented with an n_S bit wide AND gate preceded by configurable input inverters. However, similar to the reset signal discussed in Section 3.1, an FSM might have a common error state which is reached from several (but not all) states if an error input is active. For such multi-source transitions, an SSG implemented as an n_S input LUT instead of an AND gate can be used. Now, the respective transition row can be enabled for multiple states.

Input Pattern Gate (IPG). The IPG can be implemented in different ways. The first implementation is similar to that of the SSG, i.e., a n_T wide AND gate with configurable input inverters. This gate can recognize only one specific input pattern.

The second implementation is to use a LUT, such that it is possible to activate the transition for multiple different input patterns. This is beneficial either if the transitions from one state have mixed don't care conditions or if an IPG with more inputs than necessary is used (e.g., because all rows with less inputs are used by other transitions).

Input Switch Matrix (ISM). Consider an ISM with n_I input signals of which $n_T < n_i$ are connected to the IPG, requiring n_I switches per connected signal. Since only one (or none) of the n_I input signals is selected, there are $n_I + 1$ possible switch combinations, which can be encoded as binary numbers with $\lceil \log_2(n_I + 1) \rceil$ bits. Altogether, this yields $n_T \cdot \lceil \log_2(n_I + 1) \rceil$ configuration bits per transition row. If we exclude the possibility to select no signal (e.g. if the IPG is realized as a LUT such that it can implement don't cares), this number reduces to $n_T \cdot \lceil \log_2(n_I) \rceil$.

Another variant to encode the configuration of an ISM exploits that the order of the input signals for the IPG is not important. Therefore, a bit vector of size n_I of which a maximum of n_T bits are set to "1" is sufficient. Every bit represents a primary input which is connected to the IPG if its respective bit is "1". However, this encoding is only more concise than the previous one if $n_I + 1 < n_T \cdot \lceil \log_2(n_I + 1) \rceil$. That is, if a TR-FSM on average has transition rows of small sizes, the previous encoding is more beneficial.

The number $n_I \cdot n_T$ of switches per ISM can also be optimized due to the insignificance of the order of the selected inputs (see Figure 6): If the leftmost input signal is selected, it can be selected as first IPG input signal, and since it can be selected only

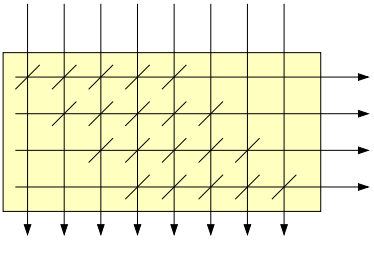


Fig. 6. Optimizing the Input Switch Matrix

once, no further switches are required for the leftmost input signal. The same holds for the second input signal on the left and the second IPG input signal, and so on. Also, the rightmost input signal can always be handled by the last IPG input signal, therefore the respective switch can be removed from the previous IPG input signals, and so on. Altogether, from every selection signal, $n_T - 1$ switches can be removed, such that the total number of switches is $[n_I - (n_T - 1)]n_T$, which gives a reduction of $n_T(n_T - 1)$ switches.

Next State Selection. Every transition row holds the configurable next state ID register (NSR). A multiplexer selects the next state ID associated with the transition row which outputs a logical "1". The configuration applied by the bit stream must ensure, that for every combination of inputs and states at most one of the IPG emits a logic "1" and all others emit a logic "0". The case that all outputs are "0" is legal because in this case the current state is maintained (note the connection of the current state to the "Select" box in Figure 4).

We expect the total number of transitions to be rather high (several tens to several hundreds, see Table 1). For such huge multiplexers the heterogeneous tree approach [8] was proposed. An implementation with pass transistors or transmission gates provides a purely combinational circuit [9].

4 Analysis

We analyzed the proposed approach using several benchmark FSMs taken from the LGSynth93 Benchmark [7]. We chose FSMs which need at least 4 state bits and had relatively few transitions per state. These FSMs were grouped into 3 sets with roughly similar values regarding state vector width, input signals, and output signals. For each group, a TR-FSM was specified by the following algorithm:

Let $\Gamma = \{A_1, \ldots, A_k\}$ be the FSM-group, $n_I(A_i)$ the number of input signals of an FSM A_i, and $N = \max\{n_I(A_1), \ldots, n_I(A_k)\}$. Denote by $m_{i,j}$ the number of transitions in A_i which need to inspect exactly j input signals (that is, they can be implemented with a transition row of size at least j). This results in a $k \times N$ matrix $(m_{i,j})$, where each row effectively specifies an optimal TR-FSM for the respective FSM.

The TR-FSM for the group Γ is now specified by a vector $(M_N, \ldots, M_0)$, where M_j is the number of transition rows of size j needed. This vector must satisfy the equations

$$M_j = \max_{i=1,\ldots,k} \left\{ m_{i,j} - \sum_{j<l\leq N} M_l - m_{i,l} \right\}$$

for each entry. That is, $M_N = \max_{i=1,\ldots,k}\{m_{i,N}\}$, and the rest of the M_j can be computed iteratively. This formula takes into account that a transition row of size n_T can also implement a transition which needs to inspect less than n_T input signals.

This algorithm was applied to the three groups to specify TR-FSMs with a stand-alone output logic (Table 1) as well as TR-FSMs which include output signal generation as described in Section 3.2 (Table 2). In the second case, the FSMs where taken "as is"

Table 1. Example implementations without outputs

Name	#states	#state-bits	#inputs	#outputs	#transitions	#non-loop transitions	input signals to inspect per bitwidth									configuration bits	utilization (in %)
							8	7	6	5	4	3	2	1	0		
planet	48	6	7	19	71	70					6	7	9	17	31	↓	32
s208	18	5	11	2	35	34							31	2	1		20
s420	18	5	19	2	35	34							31	2	1		20
s510	47	6	19	7	75	52							10	18	24		20
s820	25	5	18	19	107	85	2	3	1	5	9	19	25	21			82
sand	32	5	11	9	90	60			3	14	11	8	4	19	1		68
scf	121	7	27	56	152	152	2				16	2		30	102		100
TR-FSM	n.a.	**7**	**27**	**56**	n.a.	**152**	**2**	**3**	**1**	**11**	**11**	**11**	**25**	**21**	**67**	**5087**	n.a.
ex1	20	5	9	19	73	57					11	12	31	1	2	↓	68
ex4	14	4	6	9	18	16						2	4	10			9
mark1	15	4	5	16	22	22						6	1	2	13		12
s1488	49	6	8	19	118	116				6	7	9	17	76	1		100
styr	30	5	9	10	92	73			3	24	17	10	3	7	9		81
TR-FSM	n.a.	**6**	**9**	**19**	n.a.	**116**			**3**	**24**	**17**	**10**	**3**	**58**	**1**	**4037**	n.a.
bbsse	16	4	7	7	42	35					7	5	19	4		↓	25
cse	16	4	7	7	55	39				14	11	10	3	1			80
keyb	19	5	7	2	46	45		8	8	8	2	6	7	4	2		85
opus	10	4	5	6	22	16					4	1		5	6		11
pma	25	5	8	8	38	38				8	18	7	2	3			82
s1	20	5	8	6	80	68		2	1	4	18	16	19	6	2		100
s386	14	4	7	7	40	32					7	6	18	1			27
TR-FSM	n.a.	**5**	**8**	**8**	n.a.	**68**		**8**	**8**	**8**	**2**	**15**	**19**	**6**	**2**	**3439**	n.a.

Table 2. Example implementations with outputs

Name	#states	#state-bits	#inputs	#outputs	#transitions	input signals to inspect per bitwidth									configuration bits	utilization (in %)
						8	7	6	5	4	3	2	1	0		
planet	48	6	7	19	111					14	11	34	33	19		60
s208	18	5	11	2	70					64	6					58
s420	18	5	19	2	70					64	6				↓	58
s510	47	6	19	7	75							10	41	24		39
s820	25	5	18	19	149	6	5	1	6	45	46	37	3			98
sand	32	5	11	9	102			5	17	13	15	8	44			71
scf	121	7	27	56	152	2				16	2		30	102		100
TR-FSM	n.a.	**7**	**27**	**56**	**152**	**6**	**5**	**1**	**10**	**42**	**45**	**37**	**3**	**3**	**17058**	n.a.
ex1	20	5	9	19	136			24	20	34	33	20	5		↓	72
ex4	14	4	6	9	20						2	10	8			7
mark1	15	4	5	16	22						6	1	2	13		7
s1488	49	6	8	19	225			24	23	13	62	63	39	1		100
styr	30	5	9	10	101			6	41	26	8	4	7	9		58
TR-FSM	n.a.	**6**	**9**	**19**	**225**			**24**	**23**	**31**	**44**	**63**	**39**	**1**	**13150**	n.a.
bbsse	16	4	7	7	44					8	13	18	5		↓	33
cse	16	4	7	7	58				19	18	15	5	1			88
keyb	19	5	7	2	46		8	8	8	3	6	7	4	2		74
opus	10	4	5	6	22				4	2		9	7			18
pma	25	5	8	8	40				8	18	7	3	4			46
s1	20	5	8	6	80		2	2	4	22	18	22	8	2		100
s386	14	4	7	7	44					11	14	18	1			34
TR-FSM	n.a.	**5**	**8**	**8**	**80**		**8**	**8**	**8**	**13**	**15**	**18**	**8**	**2**	**4507**	n.a.

without any preprocessing. In the first case, a simple optimization algorithm was applied to the FSMs in the groups which combined transitions with equal start- and target-state and different outputs to one transition, which yields less transitions rows. Also, the transition rows tend to be smaller, since this optimization process produces more don't cares. Also, loops didn't contribute to the entries of the matrix $(m_{i,j})$, since they are treated as default transitions and don't need to be implemented with a transition row. In both cases, if an FSM possessed a reset input signal, this signal was not considered and a special transition row as outlined in Section 3.1 was added to the TR-FSM.

The number of configuration bits needed for each TR-FSM was assessed as follows: Each transition row needs n_S configuration bits for the SSG and the NSR, respectively. Each transition row T of size n_T needs $\lceil \log_2 n_I \rceil \cdot n_T$ configuration bits for the ISM (binary encoding) and 2^{n_T} configuration bits for the IPG (implemented as LUT). If the output is computed in the transition rows, we have to add n_O bits for each transition row as well. Summarizing, if $T(A)$ denotes the transition rows of an RT-FSM A, the number of configuration bits is

$$T(A) \cdot 2 \cdot n_S + \sum_{T \in T(A)} (\lceil \log_2 n_I \rceil \cdot n_T + 2^{n_T})$$

if no outputs are considered. The leftmost summand changes to $T(A)(n_O + 2 \cdot n_S)$ if the output is also computed in the transition rows. Each of the TR-FSMs specified need a reset transition row, which requires $\lceil \log_2 n_I \rceil$ configuration bits for the ISM, and one bit to configure possible inversion of the reset bit.

Table 1 shows the result for TR-FSMs without output computation. Here, the column "#non-loop transitions" gives the number of transitions which actually need transition rows for implementation. Table 2 treats the case where the outputs are computed within the transition rows. In both Tables, below of each of the three groups, the data for the respective groups' RT-FSM is given. Note that only relatively few of the input signals have to be inspected, at most 8 in any case.

Each of the FSMs in each group was then synthesized to the respective group TR-FSM, which is a simple mapping process of transitions to transition rows with enough resources. The column to the right shows the resource utilization by comparing the number of configuration bits needed for the used transition rows to the total number of configuration bits of the respective group TR-FSM.

Since no silicon implementation of the TR-FSM approach has been done yet, we don't compare to an FPGA implementation regarding chip-space and power consumption. Therefore we take the size of the configuration bit stream of both approaches as a cost indicator. While a LUT in an FPGA only needs $2^4 = 16$ configuration bits, there are considerably more configuration bits needed to configure the whole slice and especially the routing between the CLBs. According to [10] the configuration bit stream of the Virtex family dedicates 864 bits per CLB and therefore 432 bits per slice.

For every FSM in our analysis we took the lower number of slices occupied by the VHDL or Verilog implementation according to Table 5.7 (p. 83) of [4]. Within each group, the FSM with the most used slices was picked (see Table 3). With this amount of slices any other FSM in the particular group can be implemented. The length of the configuration bit stream was calculated for the maximum FSM, and then compared

Table 3. Configuration bits needed for TR-FSM and FPGA

Group	Conf.Bits TR-FSM	max. FSM [4]	Slices [4]	Conf.Bits FPGA	Comparison
1	17,058	scf	168	145,152	11.8 %
2	13,150	styr	119	102,816	12.7 %
3	4,507	pma	71	61,344	7.3 %

to length of the configuration bit stream of the respective group's TR-FSM. The last column of Table 3 shows the size of the TR-FSM configuration bit stream compared to the FPGA configuration bit stream in percent.

This shows that TR-FSMs need considerably less configuration bits than FPGAs (7.3 to 12.7 %). While we see this as an indicator for a prospective improvement regarding area and power consumption, this is also beneficial by itself regarding run-time reconfigurability and for applications with limited memory resources like wireless sensor nodes. Note that we don't compare directly to the improvements of [4] since this approach involves BRAMs.

FPGAs achieve the high flexibility with an high amount of rich and flexible routing resources. On the other hand the involved short and long wires impose a high capacitive load and include numerous pass transistors along the path. Both result in increased power consumption and area overhead. TR-FSM, on the other hand, have a very low routing overhead, since the general FSM structure is already implemented. Also, the unused transition rows of an TR-FSM can be switched off by voltage gating to save power.

Other advantages are the constant latencies of the TR-FSM, since the routing is fixed, and the fact that the synthesis of an FSM to a specific TR-FSM is very easy, since arbitrary state encodings can be used and no sophisticated minimization problems are involved.

5 Conclusion and Future Work

This paper introduces a novel, fixed latency architecture for reconfigurable finite state machines, which is based on the state transitions instead of the state transfer function (transition-based reconfigurable FSMs, TR-FSM), which is beneficial if the FSMs in question have relatively few transitions.

Synthesizing an FSM onto a given TR-FSM is straightforward, and only few configuration bits are needed. Compared to an implementation of FSMs in an (embedded) FPGA, the approach also promises several advantages regarding power consumption and area. However, this has to be quantified yet by concrete implementations.

Further extensions of our approach range from multi-bit LUTs for the IPG, such that a single transition row can handle transitions to multiple different states and/or with varying output patterns, with only a little overhead in the LUT. For linear sequences simplified transition rows with zero or one input signal, or specialized time-out transition rows with an integrated counter are considered.

Transitions which have to consider a high number of input signals should be mapped to special transition rows with IPGs implemented as sum-of-product logic. Another option

is to combine two transition rows and place a MUX to select between the outputs of the two IPGs. Its select input is then driven by another FSM input, so increasing the total input signal sensitivity list by one. Finally, by enhancing the NSR to latch the current state, sub-state-machines which are able to return to the caller may be implemented.

References

1. Hartenstein, R.: Coarse Grain Reconfigurable Architecture. In: Proceedings of the 2001 Conference on Asia South Pacific Design Automation, Yokohama, Japan, pp. 564–570 (2001)
2. Katz, R.H.: Contemporary Logic Design. The Benjamin/Cummings Publishing Company, Inc. (1994)
3. Rabaey, J.M., Chandrakasan, A., Nikolic, B.: Digital Integrated Circuits. Prentice Hall, Upper Saddle River (2003)
4. Bukowiec, A.: Synthesis of Finite State Machines for Programmable Devices Based on Multi-Level Implementation. PhD thesis, University of Zielona Gora, Poland (2008), http://zbc.uz.zgora.pl/Content/14528/PhD-ABukowiec.pdf (retrieved 2009-11-03)
5. Liu, Z., Arslan, T., Khawam, S., Lindsay, I.: A High Performance Synthesisable Unsymmetrical Reconfigurable Fabric For Heterogeneous Finite State Machines. In: Proceedings of the ASP-DAC, January 18-21, vol. 1, pp. 639–644 (2005)
6. Milligan, G., Vanderbauwhede, W.: Implementation of Finite State Machines on a Reconfigurable Device. In: Proceedings of the second NASA/ESA Conference on Adaptive Hardware and Systems, Edinburgh, August 5-8, pp. 386–396 (2007)
7. McElvain, K.: LGSynth93 Benchmark Set: Version 4.0 (1993), http://www.cbl.ncsu.edu/pub/Benchmark_dirs/LGSynth93/
8. Lin, M.B.: On the design of fast large fan-in CMOS multiplexers. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19(8), 963–967 (2000)
9. Alioto, M., Palumbo, G.: Interconnect-Aware Design of Fast Large Fan-In CMOS Multiplexers. IEEE Transactions on Circuits and Systems II: Express Briefs 54(6), 484–488 (2007)
10. Xilinx, Inc.: Virtex Series Configuration Architecture User Guide (XAPP151). v1.7 edn., October 20 (2004)

MEMS Dynamic Optically Reconfigurable Gate Array Usable under a Space Radiation Environment

Daisaku Seto and Minoru Watanabe

Electrical and Electronic Engineering
Shizuoka University
3-5-1 Johoku, Hamamatsu, Shizuoka 432-8561, Japan
tmwatan@ipc.shizuoka.ac.jp

Abstract. Embedded devices used for spacecraft, satellites, and space stations are vulnerable to the effects of high-energy charged particles. To resolve single-event latch-up (SEL)-associated troubles more flexibly using limited hardware resources in a space environment, reconfigurable devices such as field programmable gate arrays (FPGAs) are suitable. However, such reconfigurable systems present the shortcoming that the circuit itself on the gate array is not robust. The configuration context on a configuration SRAM also suffers from single-event upsets (SEUs) and SELs. This paper therefore proposes an MEMS dynamic optically reconfigurable gate array that is usable under a space radiation environment. The technique enables rapid recovery of a programmable device that has been damaged by high-energy charged particles. It uses incorrect configuration data including some error bits that had been damaged by particles. The configuration data are transferred using wireless communications and are retained on an EEPROM/SRAM.

1 Introduction

Embedded devices used for spacecraft, satellites, and space stations are vulnerable to effects of high-energy charged particles. They can cause single-event upsets (SEUs) and single-event latch-ups (SELs) in the devices in cases where such particles are incident to embedded devices. The SEUs cause temporary hardware failures and alter the contents of SRAMs, DRAMs, and flip-flops, whereas SELs cause permanent hardware damage [1][2][3][4]. Although such SEUs and SELs are unavoidable, the effects of SEUs are correctable using triple-module redundancy (TMR) [5][6]; SEL troubles can be averted and corrected using backup hardware.

However, reconfigurable devices such as field programmable gate arrays (FPGAs) are very suitable to resolve SEL-associated troubles more flexibly using limited hardware resources in a space environment. When SELs permanently damage a part of an FPGA, the configuration context can be changed remotely using wireless communication. However, such a reconfigurable system includes an important shortcoming: the gate array's circuit itself is not robust. A configuration context on a configuration SRAM also suffers from SEUs and SELs; the hardware of other non-programmable VLSIs is more robust. It has been confirmed that TMR implementations are useful for such FPGA hardware trouble and for data troubles [7][8]. Therefore, important remaining concerns are how to recover real-time operations on a programmable gate array

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 134–144, 2010.

rapidly when the device is damaged by SEUs or SELs and how to remedy unavoidable damage of configuration data.

Configuration contexts are always affected by radiation in space when configuration data are transferred with wireless communication and are retained on an EEPROM or an SRAM (EEPROM/SRAM). Of course, although some damage might be corrected by error checking and correction methods [9], long-term storage of error-free configuration data is difficult under a space radiation environment. Furthermore, FPGA reconfiguration times are very slow. For that reason, rapid recovery of a real-time system on an FPGA is impossible.

As rapidly reconfigurable devices, optically reconfigurable gate arrays (ORGAs) have been developed that combine a holographic memory and an optically programmable gate array VLSI [10]–[14]. Contexts of the gate array are stored in a holographic memory, from which they are read out optically and programmed optically onto the gate array VLSI using photodiodes. Such parallel configuration enables extremely fast reconfiguration. Furthermore, such ORGA architectures present the possibility of providing a virtual gate count that is much larger than those of currently available VLSIs by exploiting the large storage capability of the holographic memory. To date, for realization of a high gate count ORGA-VLSI, Dynamic Optically Reconfigurable Gate Arrays (DORGAs) have been proposed: they use the junction capacitance of photodiodes as dynamic memory, thereby obviating the static configuration memory [11]. In fact, a 51,272 gate count DORGA-VLSI has been reported [13]. However, programming for a holographic memory is optically executed in conventional ORGAs. For that reason, simple programming that is similar to that of FPGAs has remained a difficult undertaking because a complicated writer is required for programming an ORGA [14].

On the other hand, as a microelectromechanical system (MEMS) technology, Texas Instruments Inc. has recently developed a digital micromirror device (DMD) [17]. The DMD chip is a spatial light modulator. Its switching speed is μs order and its light efficiency is extremely high. Such devices are useful as electrically rewritable holographic memories.

This paper therefore proposes an MEMS FPGA that is usable in a space radiation environment. The technique enables rapid recovery of a programmable device that has been damaged by high-energy charged particles. It uses incorrect configuration data including some error bits: the configuration data have already been damaged by particles while they are transferred with wireless communication and are retained on an EEPROM/SRAM.

2 Rapidly Reparable Hardware System

A rapidly reparable hardware system using an MEMS is presented in Fig. 1. This system comprises a wireless communication circuit, an EEPROM/SRAM, an MEMS device, and an optically reconfigurable gate array VLSI (ORGA-VLSI) [11][12][14]. In the system, configuration contexts are sent by wireless communication, after which they are stored in an EEPROM/SRAM. However, the configuration context might include error bits that have been created during their transfer by wireless communication. As they are received, they are retained on the EEPROM/SRAM, where the amount of error bits

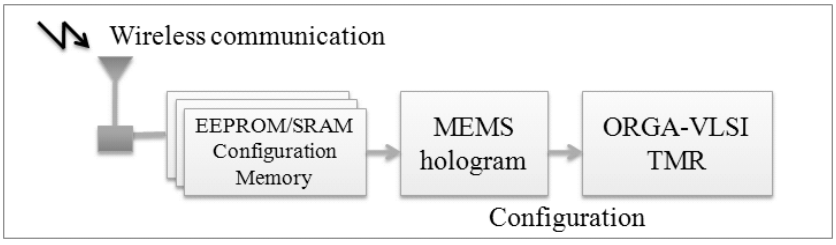
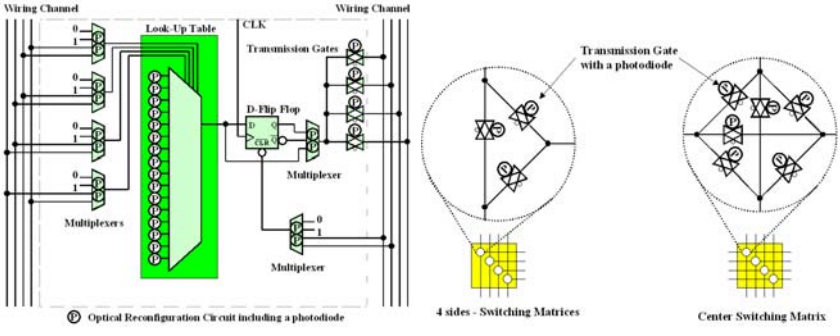


Fig. 1. Rapidly reparable hardware system using MEMS for spacecraft, satellites, and space stations



(a) (b)

Fig. 2. Structures of (a) an optically reconfigurable logic block and (b) an optically reconfigurable switching matrix on a fabricated ORGA-VLSI chip

included on the configuration contexts is increased. Therefore, using this method, the configuration context information is not communicated directly and is not stored directly on an EEPROM/SRAM. In advance, the configuration context information is converted to the corresponding holographic memory information. Then, the holographic memory information is sent on wireless communication and is stored on an EEPROM/SRAM. The stored holographic memory information is transferred cyclically to the MEMS device. On the other hand, dynamic optically reconfigurable gate array (DORGA)-VLSIs always include a fine-grain gate array similar to that of FPGAs. Structures of an optically reconfigurable logic block and an optically reconfigurable switching matrix on the fabricated ORGA-VLSI chip are portrayed in Figs. 2(a) and 2(b). Any Boolean function can be programmed onto optically reconfigurable logic blocks and any wiring between optically reconfigurable logic blocks can be achieved by the optically reconfigurable switching matrices, as well as those of FPGAs. An optically reconfigurable logic block consists of a four-input-one-output look-up table (LUT), some multiplexers, transmission gates, and a delay type flip-flop. These functions are optically reconfigurable using 32 photodiodes arranged perfectly in parallel. Similarly, switching matrices can be reconfigured optically. All programming points of

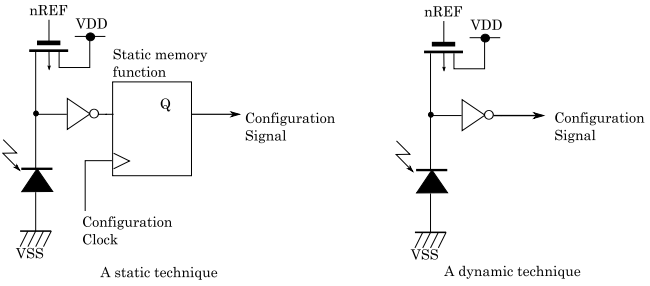


Fig. 3. Optical reconfiguration circuits using static and dynamic techniques

all transmission gates are connected to photodiodes. Finally, all programming elements of a gate array on the ORGA-VLSI chip can be optically reconfigured simultaneously. Therefore, high-speed configuration is possible.

An example of a static configuration memory of a previously proposed ORGA is shown on the left side of Fig. 3. Conventional ORGAs invariably adopt such a static configuration memory to retain a configuration context. In contrast, in the DORGA architecture, photodiodes are used not only for detecting optical configuration contexts but also for maintaining the state of a single configuration context using junction capacitance of the photodiodes, as shown on the right side of Fig. 3. Thereby, the architecture perfectly removes static configuration memories. For that reason, this architecture can produce VLSI with a very high gate count [13].

Moreover, the DORGA-VLSI has a high fault tolerance because a DORGA has a parallel configuration capability that never prevents reconfiguration of any location of its gate array, even if certain gates or certain configuration circuits are damaged by high-energy charged particles. However, in FPGAs, partial damage of certain configuration circuits makes its entire configuration impossible. The clear benefit is that although SEUs and SELs occur and can affect an ORGA-VLSI, functions can be executed perfectly on other non-faulty areas by reconfiguring the ORGA-VLSI.

In this system, a holographic configuration context stored on an MEMS device can be programmed onto a DORGA-VLSI in an extremely short time by exploiting two-dimensional optical connections between the MEMS device and DORGA-VLSI when an SEU occurs. In a holographic memory, each bit of a reconfiguration context can be generated from the entire area of its holographic memory pattern on the MEMS device. Consequently, an optical majority voting operation is executed automatically for each configuration bit. Such situations are shown in Figs. 4(a) and 4(b). Figure 4(a) shows a normal case using a correct holographic memory with a single bright bit. In this case, each area of the holographic memory is designed to be transparent at the same phase points and to be opaque at different phase points for a single bright point on a DORGA-VLSI plane. Results show that the bright point has high light intensity, which signifies a binary state H by focusing same-phase light waves. On the other hand, Fig. 4(b) shows the same single bit generation case, but from a damaged holographic memory.

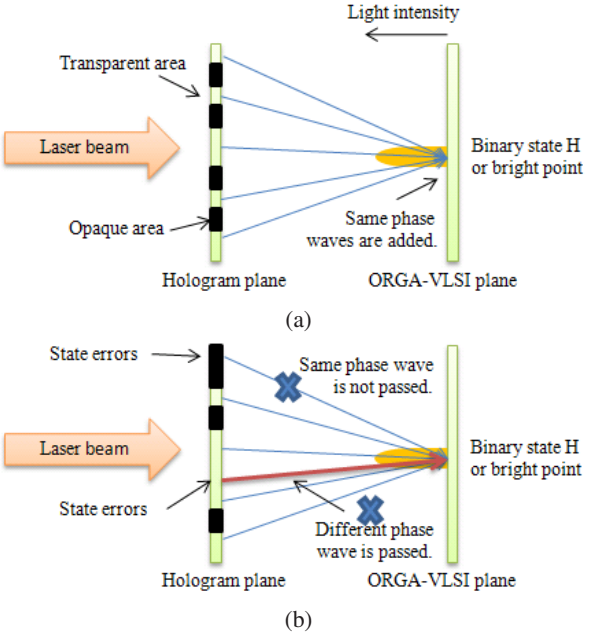


Fig. 4. Configuration context generations from a holographic memory. Panel (a) presents a normal case using a correct holographic memory with a single bright bit or binary state H. Panel (b) depicts the same single bit generation case, but from a damaged holographic memory.

A certain amount of damage might cause transparent points to be transformed to opaque points. In such an instance, although light power at a bright point of the DORGA-VLSI plane is slightly decreased, a bright bit still keeps high level by correcting correct phase waves from the other points of its holographic memory. Furthermore, a certain amount of damage might cause opaque points to be transparent points. In this case, light power at the bright point of the DORGA-VLSI plane might also be reduced slightly by adding a different phase light wave to a certain correct phase light wave. However, the bright point on the DORGA-VLSI can still keeps high level. Such situations can be considered as an optical majority voting operation. Even if configuration data include some error bits, the optical majority voting operation can repair the error bits robustly.

Regarding SELs, the system requires a certain period to repair a gate array because of a serial transfer, just as FPGAs do. However, the SELs' probability of occurring is much lower than that of SEUs. For that reason, this never becomes a serious shortcoming. If an SEL occurs, then the optical buffering technique repairs the gate array by a serial transfer of wireless communication or by placement of another configuration pattern on the EEPROM/SRAM. Even if configuration data of the serial transfer include some error bits, the optical majority voting operation can remove the error bits on configuration data. Therefore, a robust configuration can be achieved for SELs.

3 MEMS Dynamic ORGA Architecture

Texas Instruments Inc. recently developed a digital micromirror device (DMD) as one technology used in microelectromechanical systems (MEMS) [17]. The DMD device chip is a type of spatial light modulator. A photograph of the chip is presented in Fig. 6(b). It consists of 1,024 $\times$ 768 mirrors, each of which is 10.8 $\times$ 10.8 μm^2. The

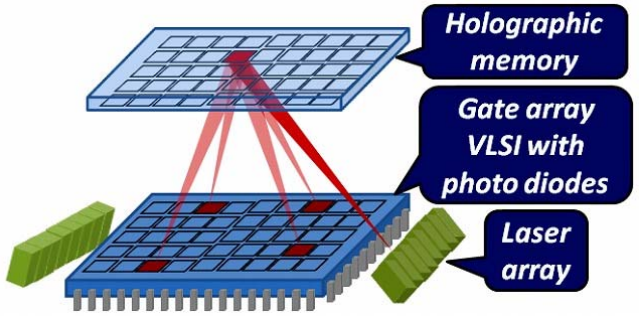
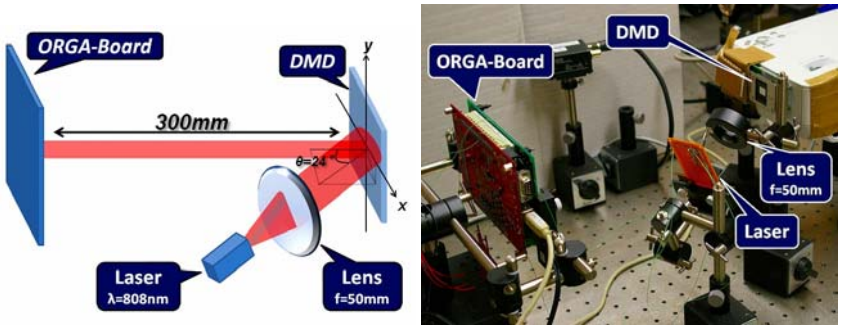
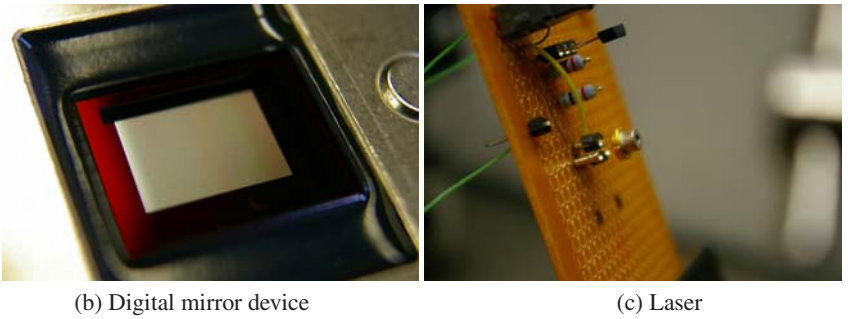


Fig. 5. Overview of an experimental system



(a) Experimental system



(b) Digital mirror device (c) Laser

Fig. 6. Photograph of an experimental system

DMD device is controllable using a personal computer. When a light beam is applied to the device, mirrors on the DMD chip can reflect a binary data pattern or video image. The switching speed and light efficiency far surpass those of other spatial light modulators, e.g. liquid crystal spatial light modulators. The micromirrors are mounted on tiny hinges, which enable them to tilt either toward the light source or away from it. Each mirror can be switched on and off up to several thousand times per second. Such a device is useful as an electrically rewritable holographic memory.

Figure 5 presents an overview of an MEMS dynamic optically reconfigurable gate array. The MEMS dynamic optically reconfigurable gate array comprises laser sources, an optical holographic memory, and a dynamic ORGA-VLSI, which uses photodiode memory architecture. The MEMS holographic memory can store numerous reconfiguration contexts, which are addressable by a laser array. The diffraction pattern from the holographic memory can be received as a reconfiguration context on a photodiode array of a programmable gate array on the dynamic ORGA-VLSI. By virtue of these features, this architecture enables nanosecond-order reconfiguration and multiple reconfiguration contexts. In addition, since the MEMS device can be programmed electrically, the circuit information of the MEMS dynamic optically reconfigurable gate array can be programmed electrically, as well as FPGAs.

3.1 Hologram Generation

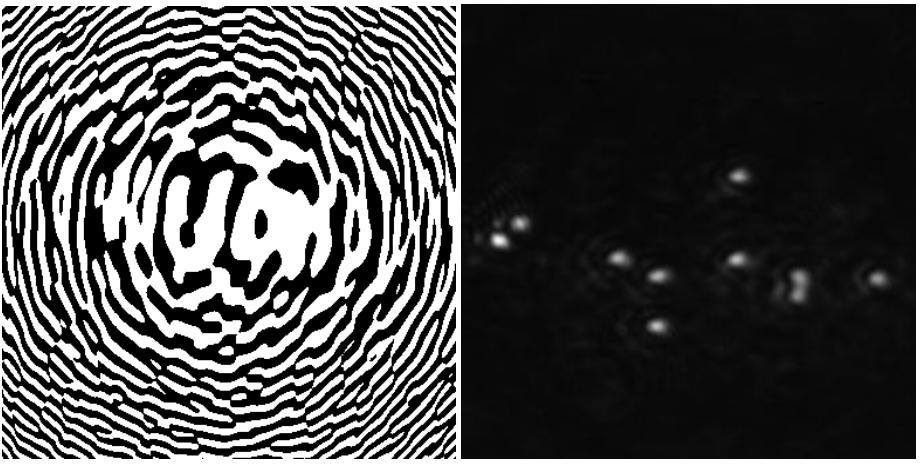
Here, a calculation method for two-dimensional holographic medium is introduced. An aperture plane of target lasers, a holographic plane, and a DORGA-VLSI plane are parallelized. The laser beam is collimated. The collimated reference wave from the laser propagates into the holographic plane. The holographic medium comprises rectangular pixels of size $\delta_x \times \delta_y$ on the $x_1 - y_1$ holographic plane. The pixels are assumed as binary values. On the other hand, the input object is made up of rectangular pixels of size $d_x \times d_y$ on the $x_2 - y_2$ object plane. The pixels can be modulated to be either on or off. The intensity distribution of a holographic medium is calculable using the following equation:

$$H(x_1, y_1) \propto \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} O(x_2, y_2) \sin(kr) dx_2 dy_2,$$
$$r = \sqrt{Z_L^2 + (x_1 - x_2)^2 + (y_1 - y_2)^2}, \tag{1}$$

where $O(x_2, y_2)$ is a binary value of a reconfiguration context, k is the wave number, and Z_L is a distance between the holographic plane and the object plane. The value $H(x_1, y_1)$ is normalized as 0–1 for the minimum intensity H_{min} and maximum intensity H_{max}, as the following.

$$H'(x_1, y_1) = \frac{H(x_1, y_1) - H_{min}}{H_{max} - H_{min}}. \tag{2}$$

Finally, the normalized image H' is used for implementing a holographic memory. Other areas on the holographic plane are opaque to the illumination.

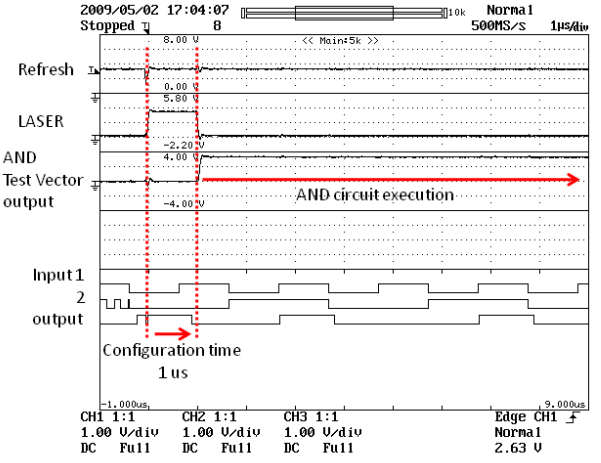


(a) Binary holographic memory (b) Diffraction pattern

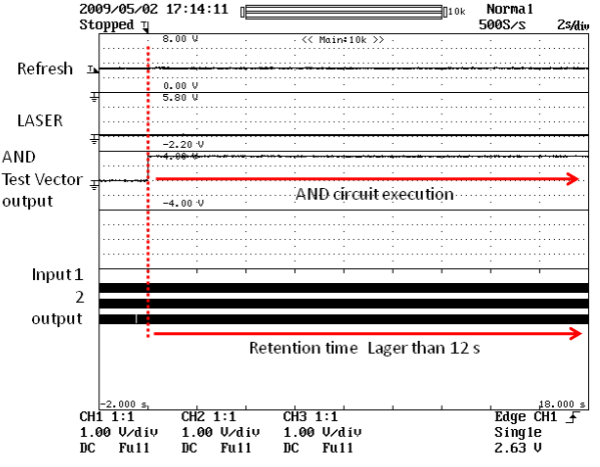
Fig. 7. Binary holographic memory pattern and its diffraction pattern of an AND circuit

3.2 Experimental System

For this study, only one configuration system was constructed as the first step for MEMS dynamic optically reconfigurable gate array development. Figure 6(b) portrays a block diagram of an MEMS dynamic optically reconfigurable gate array. The MEMS optically reconfigurable gate array was constructed using an 808 nm, 150 mW semiconductor laser, a digital micromirror device as a holographic memory, and an emulated DORGA-VLSI. Figure 7(a) shows that an MEMS holographic memory pattern of an AND circuit was calculated. The holographic memory pattern was displayed on the DMD device. After programming, all mirrors on the DMD device were moved to $\pm 12^{\circ}$ angles corresponding to the AND holographic memory pattern. The collimated beam from the laser source, which is incident to the DMD device, is reflected to the DORGA-VLSI. The DORGA-VLSI was placed 300 mm distant from the DMD device. For this experiment, an emulated DORGA-VLSI chip was used. It had been fabricated using a 0.35 μm triple-metal CMOS process. Although the DORGA-VLSI chip includes a static configuration memory, the static configuration memory was disabled in this experiment to emulate the DORGA architecture. Therefore, virtually buffered outputs of junction capacitances of photodiodes are connected directly to the gate array programming points. Photodiodes were constructed between the N-well layer and the P-substrate. The photodiode size and distance between photodiodes were designed as 25.5 $\times$ 25.5 μm and as 90 μm to facilitate the optical alignment. The gate array structure is fundamentally identical to that of typical FPGAs. The DORGA-VLSI chip includes 4 logic blocks, 5 switching matrices, and 12 I/O bits. In all, 340 photodiodes were used to program the gate array.



(a) Configuration time (1 μs)



(b) Retention time (12 s)

Fig. 8. Timing diagram presenting the configuration time and retention time of an AND circuit

4 Experimental Results

Using this experimental system, an AND circuit was implemented on the MEMS dynamic optically reconfigurable gate array. The configuration was executed successfully. A CCD captured image of an AND circuit at the position of DORGA-VLSI is presented in Fig. 7(b), as generated from an AND holographic memory on the DMD device. The reconfiguration time and retention time of the photodiode memory architecture were measured, respectively, as 1 μs and 12 s. The reconfiguration time was sufficiently faster

than that of current FPGAs, and the retention time was sufficiently longer than that of DRAMs. Additionally, we estimated the switching speed of mirrors on the MEMS holographic memory. Results confirm that less than 22 μs wireless programming is possible. Therefore, for SELs, high-speed recovery is possible.

5 Conclusion

This paper has proposed an MEMS dynamic optically reconfigurable gate array that is useful in a radiation-rich space environment. This paper has described a novel MEMS dynamic optically reconfigurable gate array with an MEMS binary hologram to exploit switching of both an MEMS binary hologram and a laser array to achieve optical reconfiguration. Results confirmed that the reconfiguration time and retention time of the MEMS dynamic optically reconfigurable gate array are, respectively, 1 μs and 12 s. Consequently, extremely rapid reconfiguration and sufficiently long retention time are achievable. In addition, programming of an MEMS holographic memory or wireless programming in less than 22 μs was confirmed as possible. Such architecture opens the possibility of using programmable devices in a space radiation environment.

In particular, current embedded systems used in space invariably require enclosure within aluminum plates to protect the systems from high-energy charged particles. Those plates constitute most of the system weight. In contrast, because the architecture described herein is extremely robust, only thin aluminum plates need be used. Consequently, this architecture is expected to allow production much lighter total systems. Therefore, the technique is useful for space-embedded systems in radiation-rich environments.

Acknowledgments

This research was supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Innovative Areas, No. 20200027. The VLSI chip in this study was fabricated in the chip fabrication program of VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Rohm Co. Ltd. and Toppan Printing Co. Ltd.

References

1. Redant, S., Marec, R., Baguena, L., Liegeon, E., Soucarre, J., Van Thielen, B., Beeckman, G., Ribeiro, P., Fernandez-Leon, A., Glass, B.: Radiation Test Results on First Silicon in the Design Against Radiation Effects (DARE) Library. IEEE Trans. on Nuclear Science 52(5), 1550–1554 (2005)
2. Makihara, A., Sakaide, Y., Tsuchiya, Y., Arimitsu, T., Asai, H., Iide, Y., Shindou, H., Kuboyama, S., Matsuda, S.: Single-Event Effects in 0.18 um CMOS Commercial Processes. IEEE Trans. on Nuclear Science 50(6), 2135–2138 (2003)
3. Ikeda, N., Shindou, H., Iide, Y., Asai, H., Kubo, S., Matsuda, S.: Evaluation of the Errors of Commercial Semiconductor Devices in a Space Radiation Environment. Trans. of the Institute of Electronics, Information and Communication Engineers J88-B(1), 108–116 (2005)

4. Lin, Y., He, L.: Devices and architecture concurrent optimization for FPGA transient soft error rate. In: International Conference on Computer Aided Design (2007)
5. Stroud, C.E.: Reliability of Majority Voting Based VLSI Fault-Tolerant Circuits. IEEE Trans. on VLSI Systems 2(4), 516–521 (1994)
6. Radu, M., Pitica, D., Posteuca, C.: Reliability and failure analysis of voting circuits in hardware redundant design. In: International Symposium on Electronic Materials and Packaging, pp. 421–423 (2000)
7. Miller, G., Carmichael, C., Jet Propulsion Labs: Single-Event Upset Mitigation for Xilinx FPGA Block Memories, XILINX Application Note, Virtex-II FPGAs (2007)
8. Barbour, A.E.: A reconfigurable fault-tolerant system. In: Midwest Symposium on Circuits and Systems, pp. 189–194 (1992)
9. Peter, J.-L.: ECC design of a custom DRAM storage unit. In: IEEE VLSI Test Symposium, pp. 171–173 (1993)
10. Mumbru, J., Panotopoulos, G., Psaltis, D., An, X., Mok, F., Ay, S., Barna, S., Fossum, E.: Optically Programmable Gate Array. In: SPIE of Optics in Computing 2000, vol. 4089, pp. 763–771 (2000)
11. Yamaguchi, N., Watanabe, M.: Liquid crystal holographic configurations for ORGAs. Applied Optics 47(28), 4692–4700 (2008)
12. Seto, D., Watanabe, M.: A dynamic optically reconfigurable gate array - perfect emulation. IEEE Journal of Quantum Electronics 44(5), 493–500 (2008)
13. Watanabe, M., Kobayashi, F.: Dynamic Optically Reconfigurable Gate Array. Japanese Journal of Applied Physics 45(4B), 3510–3515 (2006)
14. Kubota, S., Watanabe, M.: Programmable Optically Reconfigurable Gate Array Architecture and its writer. Applied Optics 48(2), 302–308 (2009)
15. Yatagai, T.: Optical space-variant logic-gate array based on spatial encoding technique. Opt. Lett. 11, 260–262 (1986)
16. Fukushima, S., Kurokawa, T.: Programmable hybrid parallel processing for real-time digital logic operations. Opt. Lett. 12, 965–967 (1987)
17. Texas Instruments, DLP, `http://www.ti.com/`

An FPGA Accelerator for Hash Tree Generation in the Merkle Signature Scheme

Abdulhadi Shoufan

Center for Advanced Security Research Darmstadt CASED, Germany*
abdul.shoufan@cased.de

Abstract. Merkle Signature Scheme relies on secure hash functions and is, therefore, assumed to be resistant to attacks by quantum computers. The generation of the Merkle public key, however, is highly time-consuming because of the huge number of hash operations required to set up a complete hash tree. Fortunately, setting up such trees features inherent parallelism, which may be utilized for accelerating this process using a specific hardware platform. This paper presents a flexible and efficient hardware architecture on an FPGA platform to accelerate the generation of Merkle hash trees. Timing measurements on a prototype with different parameters show a considerable performance boost compared to a similar software solution.

1 Introduction

Current public-key cryptosystems rely on the computational complexity of different mathematical problems such as the factoring of large integers in RSA [1] or the calculation of the discrete logarithm in elliptic curve cryptography [2]. For these approaches various hardware accelerators and coprocessor architectures are available, see, e.g., [3] and [4]. These algorithms are assumed to become insecure in the era of quantum computers [5]. Therefore, several solutions for post-quantum cryptography have been proposed in the literature such as hash-based, code-based, lattice-based, and multivariate-quadratic-equation-based cryptosystems, see [6, 7, 8, 9]. Generally, these solutions suffer from efficiency problems regarding execution time and data and key sizes. To tackle the performance problems in code-based and multivariate-quadratic-equation based approaches some hardware architectures have been proposed recently, see [10, 11, 12, 13].

Merkle signature scheme (MSS) is a hash-based cryptosystem. MSS relies on the Winternitz one-time signature scheme (W-OTS) [6] on the one hand. On the other, MSS employs a hash tree for authenticating the Winternitz verification keys. The construction of this hash tree is a highly time-consuming task especially for large trees. This paper presents a novel hardware architecture for the construction of MSS hash trees based on an FPGA-platform. A prototype is implemented on a Xilinx Virtex-5 device and tested through a dedicated API.

* This work was supported by CASED.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 145–156, 2010.

Performance measurements show the advantage of this solution over a pure software implementation.

The remainder of the paper is structured as follows. Section 2 introduces the Winternitz OTS. Section 3 details the Merkle signature scheme. Section 4 presents the architecture of the proposed MSS hash tree generator. Section 5 provides some implementation aspects and the results.

2 Winternitz One-Time Signature Scheme

The Winternitz One-Time Signature Scheme (W-OTS) [6] is an expansion of the Lamport-Diffie Scheme [14], which is used to sign messages bit-wise. W-OTS signs messages block-wise and results, therefore, in shorter signatures. In the following, the functionality of W-OTS will be illustrated with the aid of Fig. 1, where Alice intends to sign an 8-bit message (m=10110111) to Bob.

Algorithm 1. W-OTS Key Generation

Require: Winternitz parameter w; Message length and hash value length n.
Ensure: Signature key $X^W \in \{0,1\}^{n\times v}$; Verification key $Y^W \in \{0,1\}^n$.

1: Determine $v = v_1 + v_2$, where $v_1 = \lceil \frac{n}{w} \rceil$ and $v_2 = \left\lfloor \frac{\lfloor log_2 v_1 \rfloor + w + 1}{w} \right\rfloor$;
2: Choose $x_1, \ldots, x_v \in \{0,1\}^n$ uniformly at random;
3: Set $X^W = (x_1, \ldots, x_v)$;
4: Compute $y_i = H^{2^w-1}(x_i)$ for $i = 1, \ldots, v$;
5: Compute $Y^W = H(y_1 \| \ldots \| y_v)$;
6: **return** (X^W, Y^W).

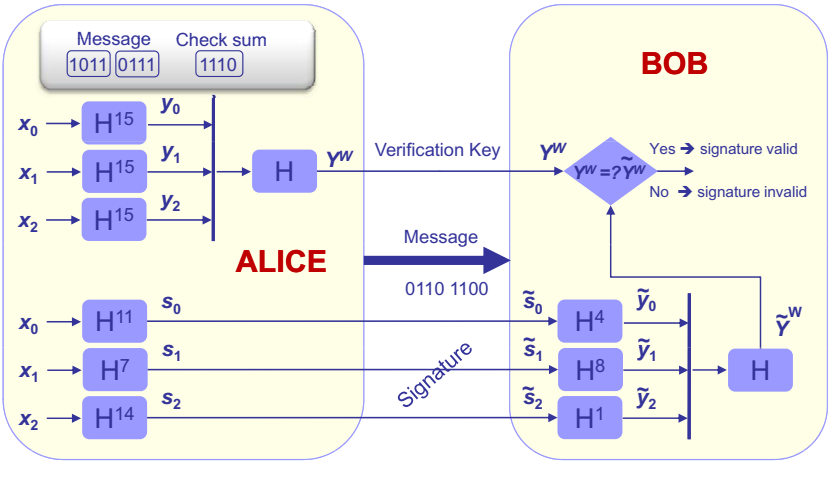


Fig. 1. Illustration of Winternitz One-Time Signature Scheme

2.1 Key Generation

First, Alice determines a checksum, which equals 1110 in this example. The checksum must also be signed for security reasons. Following, the message and the checksum are divided into blocks of the width $w = 4$ in this case. This width is denoted as *Winternitz parameter*. Having three blocks to be signed, three random numbers x_0, x_1, and x_2 of length n each are then generated. We denote these numbers as *signature sub-keys*. All the signature sub-keys form the *signature key* X^W, i.e. the private key, which must be kept secret. A hash function is applied 15 times to each x_i to determine the corresponding *verification sub-key* y_i. The number 15 corresponds to the maximal decimal that may be represented by a bit vector of width $w = 4$. All y_i's are concatenated and hashed to form the *verification key* Y^W, which is published.

Because of its relevance to our purpose, we give the process of generating a Winternitz key pair as a pseudo code in Algorithm 1 . Typically, digital signatures are not determined for a long message, but for the hash value thereof. Assuming an n-bit hash function, Table 1 gives an overview of parameters and data sizes supported by our architecture.

2.2 Signing Process

To sign the message and the checksum, Alice then applies the hash function 11 times to x_0, 7 times to x_1, and 14 times to x_2. The values 11, 7, and 14 correspond to the decimals represented by the given message and checksum blocks. Finally, the *digital signature* S^W, which consists of three *sub-signatures* s_0, s_1, and s_2, is sent to Bob.

Note that the Winternitz parameter can be used for trading off the signature length and the computation overhead. Specifically, with larger w values less blocks must be signed, i.e. the signature gets shorter, however, the computation overhead increases for both, generating the keys and signing the message.

2.3 Verification Process

Upon receiving the message, the verification key, and the signature (which may now be altered to $\tilde{s}_0, \tilde{s}_1, \tilde{s}_2$ en route), Bob first determines the checksum and divides the message and the checksum into three blocks, in the same way Alice did before. Following, Bob applies the hash function 4 times to $\tilde{s}_0$, 8 times to $\tilde{s}_1$, and once to $\tilde{s}_2$. In other words, Bob completes the number of hash operations applied to each x_i to 15. The resulting *potential* verification sub-keys $\tilde{y}_0$, $\tilde{y}_1$, and $\tilde{y}_2$ are then concatenated and hashed again. Bob compares the resulting potential verification key $\tilde{Y}^W$ with the received verification key Y^W. If they are identical, the signature is valid, otherwise not.

Note: Attacking a signature scheme aims at falsifying either the message or the signature, whereas both forms have the same effect for Bob. To simplify presentation, we assume that an attacker manipulates the signature. Bob always regards the received signature, therefore, as potential, which is indicated by a tilde.

Table 1. Parameters supported by our architecture

Parameter	Description	Size (Bit)
n	Hash value width. Width of all the seeds, all the Winternitz signature sub-keys (x's), all the verification sub-keys (y's), all the verification keys (Y^W's), and all the sub-signatures (s's)	512
w	Winternitz parameter	Adjustable (4,6,8)
v_1	Message block number	Depends on w
v_2	Checksum block number	Depends on w
v	$v_1 + v_2$	Depends on w

3 Merkle Signature Scheme

The security of W-OTS relies on using the key pair (X^W, Y^W) for signing only one message. To sign another message, a new key pair must be generated and the new verification key must be delivered authentically. However, an authentic delivery of the verification key is as difficult as the authentic delivery of the message itself. Merkle proposed a solution for this problem based on *hash trees* [6]. The Merkle signature scheme (MSS) bases the authenticity of all the verification keys on the authenticity of only one public key (Y^M), which is placed at the root of a hash tree, see Fig. 2. The tree leaves are Winternitz verification keys Y^W's, which are generated using Algorithm 1. Each internal node represents an auxiliary verification key, which results from hashing both its child keys. It holds, for instance, that $Y_{0-1} = H(Y_0^W || Y_1^W)$ and $Y^M = H(Y_{0-3} || Y_{4-7})$. Obviously, a hash tree of height h_{max} can be used to sign $2^{h_{max}}$ messages. In MSS an *authentication path* refers to all the keys, which are the siblings of the keys belonging to the path from the currently used leaf to the root. To sign the 4-*th* message, for example, the sender submits the leaf index $i = 4$, the Winternitz signature S_4^W and the authentication path $A_4 = (Y_5^W, Y_{6-7}, Y_{0-3})$. Based on $\tilde{S}_4^W$ the receiver determines $\tilde{Y}_4^W$ as depicted in the last section. Then, the potential Merkle key is determined as follows: $\tilde{Y}_{4-5} = H(\tilde{Y}_4^W || Y_5^W)$, $\tilde{Y}_{4-7} = H(\tilde{Y}_{4-5} || Y_{6-7})$, and

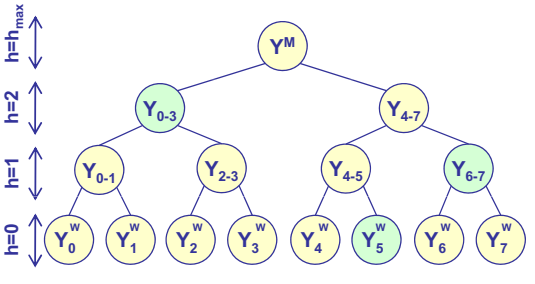


Fig. 2. Merkle Signature Scheme Tree Example

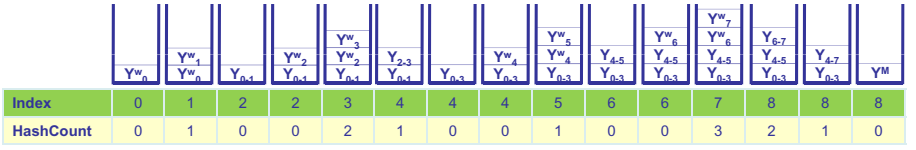


Fig. 3. MSS Stack Content During Constructing Tree of Fig. 2

$\tilde{Y}^M = H(Y_{0-3}||\tilde{Y}_{4-7})$. $\tilde{Y}^M$ is then compared with Y^M. If they are equal, the signature is valid.

Constructing large trees using MSS, however, raises two problems regarding memory usage and computation overhead. It can be shown that to generate a tree and save all the corresponding keys for $h_{max} = 20$, $w = 4$, and $n = 512$, a total of 71 gigabit memory capacity and $2.2 \cdot 10^9$ hash function executions are required.

To solve the memory problem a stack-based approach is used for the tree construction [6], which only demands a stack size of $h_{max}+1$ nodes, see Algorithm 2. Fig. 6 illustrates this algorithm by applying it to the example tree of Fig. 2. *Index* refers to the current leaf. *HashCount* denotes the number of hashing operations, that must be performed in step 17. The idea of this algorithm is to hash keys as soon as possible. This occurs each time, when two keys from the same tree level are available on the stack. This justifies the stack size of $h_{max} + 1$ nodes mentioned previously.

Note: At the end of tree construction using Algorithm 2 all the keys except for Y^M are lost. To be able to retrieve these keys in the signing phase later, a chained pseudo random number generator (PRNG) is used, which generates in each iteration besides the key a new seed used to generate the next key. After completing the tree construction, only the initial seed, which is used to generate the first Winternitz signature sub-key, must be stored for recovering all the keys and seeds in the signing phase.

4 MSS Hash Tree Generator

While Algorithm 2 minimizes the memory usage it still demands huge computational overhead. In [15] a solution for this performance problem was proposed which relies on chaining smaller trees. While this solution accelerates the initial determination of the root key, the signature process becomes slower, as each sign operation must be followed by additional steps to prepare future trees.

In contrast, the proposed MSS Hash Tree Generator (HTG) tackles the performance problem without deteriorating the signature process, as the complete tree is generated in the beginning, as in the original Merkle scheme.

4.1 HTG General Architecture

Fig. 4 depicts the general architecture of the proposed generator. Recall that the goal of the tree construction is to determine the Merkle public key Y^M.

Algorithm 2. MSS Tree Construction

Require: Hash tree height h_{max}.
Ensure: Merkle key Y^M.

```
1: Index := 0; HashCount := 0;
2: loop
3:    if HashCount = 0 then
4:       if Index = 2^{h_max} then
5:          Y^M := stack.pop();
6:          return Y^M.
7:       end if
8:       Generate a Winternitz verification key Y^W using Algorithm 1;
9:       stack.push(Y^W);
10:      Index := Index + 1;
11:      i := Index;
12:      while i mod 2 = 0 do
13:         HashCount := HashCount + 1;
14:         i := i/2;
15:      end while
16:   else
17:      Node = H(stack.pop()||stack.pop());
18:      stack.push(Node);
19:      HashCount = HashCount − 1;
20:   end if
21: end loop
```

Therefore, this key is the only output of this generator, which expects only an initial seed $SEED_0$ as an input. $SEED_0$ is provided by the host, which writes these 512-bit data into an input FIFO on the FPGA and signals that by writing a command into a command register. The FPGA writes the determined public key Y^M into an output FIFO and signals that by setting a bit in a special status register, which interrupts the software upon completing the tree generation. The FIFOs and the registers are not depicted in Fig. 4 for brevity. The width of the HTG data path including the input and output FIFOs is 64 bit. This corresponds to the width of the local bus on the used FPGA card on the one hand, and to the word width of the used hash function SHA-512 on the other, as will be seen later.

HTG is a high-performance implementation of Algorithm 2, that implies Algorithm 1. The latter demands the generation of pseudo random numbers (x's) in Step 2. For this purpose we use the pseudo random number generator specified in [16] as given in Algorithm 3. This algorithm is mapped to the module PRNG together with one of the modules SHA512 in the FHM units, see Section 4.2.

An FHM mainly includes a hash module SHA512. Besides the data to be hashed, the FHM receives the number of times these data must be hashed. This number is either 1 if an x is to be generated (Algorithm 3, Step 1) or $2^w - 1$ if an y is to be determined (Algorithm 1, Step 4).

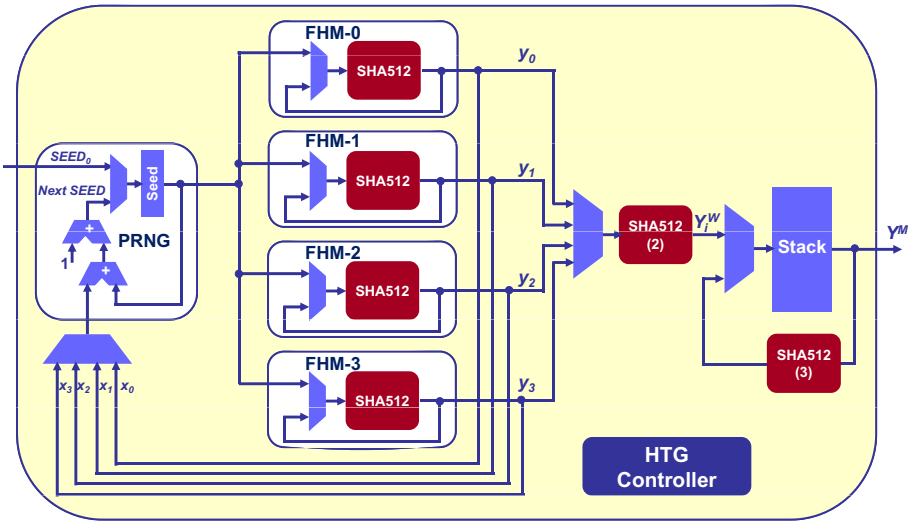


Fig. 4. Hash Tree Generator

4.2 HTG Functionality

Fig. 5 illustrates the generation of signature and verification sub-keys *schematically*: After receiving $SEED_0$, FHM-0 computes x_0 by applying the hash function once to $SEED_0$. On the one hand, x_0 is then fed back inside FHM-0 which is started again to hash x_0 $2^w - 1$ times to compute y_0. On the other, x_0 is fed back to the PRNG to compute the new seed S_1. The latter is then sent to FHM-1 to compute x_1 and y_1, and so on. When FHM-0 completes computing y_0, the seed S_4 is already available, so that FHM-0 can continue computing x_4 without waiting.

The hash module SHA512-(2) determines a verification key Y_i^W out of its v sub-keys y's (Algorithm 1, Step 5). As the hash function SHA-512 works on 1024-bit blocks (see Section 4.4), SHA512-(2) processes two 512-bit verification sub-keys at once.

The most-right hash module SHA512-(3) performs the hashing operations needed to compute the internal nodes and the root of the Merkle tree (Algorithm 2, Step 17). This task is done with the aid of the Stack under the control of the HTG Controller, which interacts with these modules to perform Algorithm 2.

Algorithm 3. Pseudo Random Number Generator

Require: $SEED_{in} \in \{0,1\}^n$.
Ensure: Pseudo random number x; New seed $SEED_{out}$.
1: $x = H(SEED_{in})$;
2: $SEED_{out} = (1 + SEED_{in} + x) \bmod 2^n$;
3: **return** $(SEED_{out}, x)$.

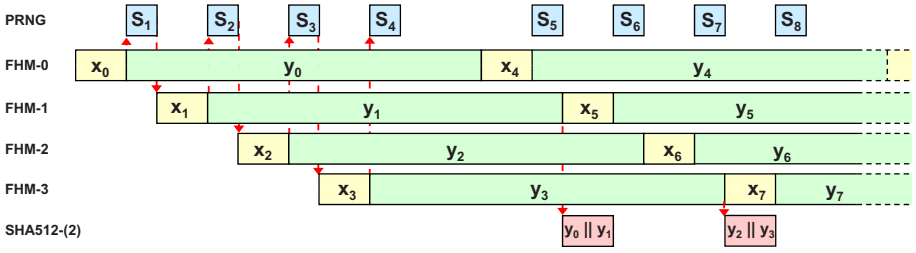


Fig. 5. Generator Mapping and Scheduling

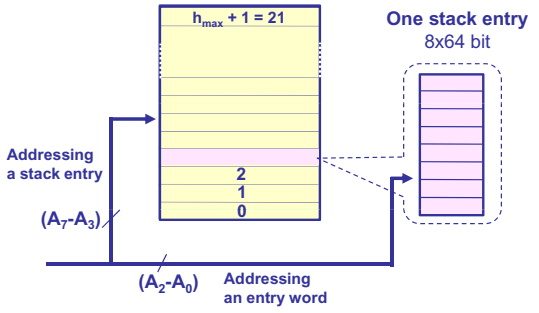


Fig. 6. Stack Organization For $h_{max} = 20$

4.3 HTG Stack Organization

For $h_{max} = 20$ the stack size is 21 entries, which may be addressed using 5 address lines, see Fig. 5. As each entry is 512-bit key and the HTG data path width is 64 bit, three additional address lines are required to address the 8 64-bit words of each key. The address lines stem from the HTG Controller and are not depicted in Fig. 4, for clarity. The stack is realized using block RAMs on the FPGA. As Virtex-5 BRAMs maximally support 32-bit words, at least 2 BRAMs are needed to work with 64-bit words.

4.4 SHA512 Module

All hash modules seen in Fig. 4 are identical and based on the hash function SHA-512 which belongs to the SHA-2 family [17]. SHA-512 operates on 1024-bit data blocks, which are extended to 80 64-bit words W_i's by applying some shift, rotation, and xor operations. A data block is hashed through 80 processing rounds, which are applied to an 8×64-bit register (A, B, C, D, E, F, G, H). This register is initiated with a constant value at the start. In each round, one word

Algorithm 4. SHA-512 Round

1: $T_1 := H + \Sigma_1(E) + Ch(E,F,G) + K_i + W_i$;
2: $T_2 := \Sigma_0(A) + Maj(A,B,C)$;
3: $H := G$;
4: $G := F$;
5: $F := E$;
6: $E := D + T_1$;
7: $D := C$;
8: $C := B$;
9: $B := A$;
10: $A := T_1 + T_2$;

of the extended message block W_i and an additional round-specific constant K_i are incorporated. The extended message words are given as follows:

$$W_i = \begin{cases} M_i^{(i)} & \text{if } 0 \leq i \leq 15 \\ \sigma_1^{\{512\}}(W_{i-2}) + W_{i-7} + \sigma_0^{\{512\}}(W_{i-15}) + W_{i-16} & \text{if } 16 \leq i \leq 79 \end{cases}$$

Thus, the first 16 words correspond to the message words themselves. The other words are determined using the functions $\sigma_0^{\{512\}}(x)$ and $\sigma_1^{\{512\}}(x)$, which are given as follows:

$$\begin{aligned} \sigma_0^{\{512\}}(x) &= ROTR^1(x) \ \oplus ROTR^8(x) \ \oplus SHR^7(x) \\ \sigma_1^{\{512\}}(x) &= ROTR^{19}(x) \oplus ROTR^{61}(x) \oplus SHR^6(x) \end{aligned}$$

Algorithm 4 depicts how the register words (A, B, C, D, E, F, G, H) are updated in each round. $\Sigma_0()$, $\Sigma_1()$, $Ch()$, $Maj()$ are all logical functions including negation, AND, XOR, shift, and rotation:

$$\begin{aligned} Ch(x,y,z) &= (x \wedge y) \quad \oplus (\neg x \wedge z) \\ Maj(x,y,z) &= (x \wedge y) \quad \oplus (x \wedge z) \quad \oplus (y \wedge z) \\ \Sigma_0^{\{512\}}(x) &= ROTR^{28}(x) \oplus ROTR^{34}(x) \oplus ROTR^{39}(x) \\ \Sigma_1^{\{512\}}(x) &= ROTR^{14}(x) \oplus ROTR^{18}(x) \oplus ROTR^{41}(x) \end{aligned}$$

Obviously, the calculations of A and E are the most timing-critical tasks in a round because of the several additions needed to determine these words. For performance reasons, we divide the round path into four pipeline stages as depicted in Fig. 7. By means of this division, no more than three words are summed in one stage. Note that this design demands the duplication of the functions $\Sigma_1()$, $Ch()$, and the addition in stage 3. With the aid of this pipelining structure, the SHA-512 module takes 86 clock cycles to hash one data block and works at a clock frequency of 190 MHz for a stand-alone implementation. This corresponds to a processing throughput of 2.4 Gbs.

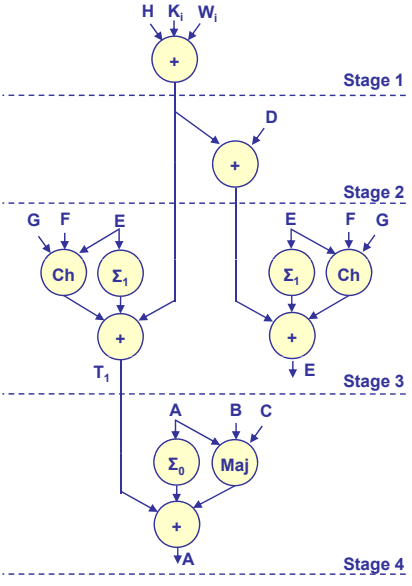


Fig. 7. Pipelined Data Flow Illustration for SHA-512 Module

5 Implementation and Results

The MSS hash tree generator was implemented on the PCI card ADM-XRC-5T1 [18], which is equipped with the Virtex-5 FPGA LX110T from Xilinx. For functionality and performance evaluation we developed a software API and tested the FPGA implementation against an open-source software implementation in the toolkit Flexiprovider [19]. Table 2 gives some performance figures and the resource usage of the HTG for $h_{max} = 10$ and $h_{max} = 20$ and for different values of the Winternitz parameter w. Additionally, several variants of the generator with different number of the FHM instances were realized and tested to show the influence of the parallelization on the performance and resource usage. Table 2 includes, furthermore, the timing figures obtained from running the Flexiprovider solution on an Intel Core2Duo E6400, 2.13 GHz with 5 GB RAM. The utilization ratio given in the last column relates to the total slice number of 17,280 on LX110T. The HTG generator uses additionally 4 block RAMs for the stack and for the input and output FIFOs needed to communicate with the host. From the results of Table 2 the following conclusions can be drawn:

1. The larger the Winternitz parameter, the longer the tree construction takes. Recall that for a larger w value less sub-signatures must be determined, i.e. the signature will be shorter, see Section 2.
2. The hardware speedup, but also the resource usage, increase with the number of FHM instances. Thus this number may be used as performance-cost trade-off parameter.

Table 2. FPGA Performance Compared to Software Implementation

Parameters		$h_{max} = 10$		$h_{max} = 20$		Speedup	Utilized Slices (Ratio)
w	FHMs	FPGA	Software	FPGA	Software		
4	1	1031 ms	11.8 s	1057 s	3.4 h	11	5,014 (29%)
6		2777 ms	31.2 s	2844 s	8.9 h	11	
8		8216 ms	91.8 s	8413 s	26.1 h	11	
4	4	288 ms	11.8 s	295 s	3.4 h	41	7,084 (41%)
6		791 ms	31.2 s	810 s	8.9 h	39	
8		2330 ms	91.8 s	2386 s	26.1 h	39	
4	8	143 ms	11.8 s	146 s	3.4 h	83	11,498 (67%)
6		394 ms	31.2 s	403 s	8.9 h	79	
8		1176 ms	91.8 s	1204 s	26.1 h	78	

3. The speedup is independent of the tree height. Thus, this architecture is scalable for larger trees, which is suitable for servers supporting message authentication. Note that only the stack size may need extension when larger trees should be supported.

Obviously, the presented architecture does not only show the feasibility of hardware solutions for the sophisticated hash-based cryptography, but also their high-performance which is indispensable for the acceptance of this class of cryptosystems.

6 Conclusion and Outlook

A novel hardware architecture for efficient generation of Merkle trees was presented, which shows the advantage of modern FPGAs to answer performance questions regarding post-quantum cryptography. Besides the parallelism grade, many parameters such as the Winternitz parameter and the tree height may be adjusted to attain desired design objectives.

Expanding the proposed architecture to perform message signing and signature verification is a part of future work. Message signing seems to be straightforward regarding the determination of the Winternitz signature, as the same FHM unit may be used. FHM, however, will read variable number of hashing operations, which corresponds to the decimal value of the message or checksum block. In contrast, determining the authentication path is a highly complex task as it depends on successive pre-calculations after each signing operation. Implementing the verification process is less demanding as no authentication path needs to be determined. A hardware implementation of the signing and verification processes seem to be especially beneficial if high throughput is required which is the case for servers, e.g. an online banking server. Hardware acceleration on the client side, in contrast, is assumed to be less useful as MSS relies on hash functions which are fast enough for causal authentication operations.

References

1. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM 21 (1978)
2. Koblitz, N.: Elleptic Curve Cryptosystems. Mathematics of Computation 48, 203–209 (1987)
3. McIvor, C., McLoone, M., McCanny, J.: Hardware Elliptic Curve Cryptographic Processor Over $rmGF(p)$. TCAS 53(9), 1946–1957 (2006)
4. Hani, M., Lin, T., Shaikh-Husin, N.: FPGA implementation of RSA public-key cryptographic coprocessor. TENCON, 6–11 (2000)
5. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings of 35th Annual Symposium on Foundation of Computer Science (1994)
6. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
7. McEliece, R.J.: A Public Key Cryptosystem Based on Algebraic Coding Theory. DSN Progress Report 42-44, 114–116 (1978)
8. Lenstra, A.K., Lenstra Jr., H.W., Lovász, L.: Factoring polynomials with rational coefficients. Math., 515–534 (1982)
9. Fell, H., Diffie, W.: Analysis of a public key approach based on polynomial substitution. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 340–349. Springer, Heidelberg (1986)
10. Shoufan, A., Wink, T., Molter, G., Huss, S., Strenzke, F.: A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms. In: 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2009 (2009)
11. Beuchat, J.C., Sendrier, N., Tisserand, A., Villard, G.: FPGA Implementation of a recently published signature scheme. Rapport de recherche RR LIP 2004-14 (2004)
12. Balasubramanian, S., et al.: Fast Multivariate Signature Generation in Hardware: The Case of Rainbow. In: 19th IEEE Int. Conf. on Application-specific Systems, Architectures and Processors, ASAP 2008 (2008)
13. El-Hadedy, M., Gligoroski, D., Knapskog, S.J.: High Performance Implementation of a Public Key Block Cipher - MQQ, for FPGA Platforms. In: International Conference on ReConFigurable Computing and FPGAs, ReConFig 2008 (2008)
14. Lamport, L.: Constructing digital signatures from a one-way function. SRI International (1979)
15. Buchmann, J., García, L.C.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – an improved merkle signature scheme. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 349–363. Springer, Heidelberg (2006)
16. NIST: Digital signature standard (dss), fips pub 186-2 (2007), `http://csrc.nist.gov/publications/fips/`
17. NIST: Secure hash standard (shs), fips pub 186-3 (2008), `http://csrc.nist.gov/publications/fips/`
18. Alpha-Data, `http://www.alpha-data.com`
19. The FlexiProvider group at Technische Universität Darmstadt: Flexiprovider, an open source java cryptographic service provider, `http://www.flexiprovider.de/` (2001-2009)

A Fused Hybrid Floating-Point and Fixed-Point Dot-Product for FPGAs*

Antonio Roldao Lopes and George A. Constantinides

Electrical & Electronic Engineering,
Imperial College London,
London SW7 2BT,
England
{aroldao,g.constantinides}@ic.ac.uk

Abstract. Dot-products are one of the essential and recurrent building blocks in scientific computing, and often take-up a large proportion of the scientific acceleration circuitry. The acceleration of dot-products is very well suited for Field Programmable Gate Arrays (FPGAs) since these devices can be configured to employ wide parallelism, deep pipelining and exploit highly efficient datapaths. In this paper we present a dot-product implementation which operates using a hybrid floating-point and fixed-point number system. This design receives floating-point inputs, and generates a floating-point output. Internally it makes use of a configurable word-length fixed-point number system. The internal representation can be tuned to match the desired accuracy. Results using a high-end Xilinx FPGA and an order 150 dot-product demonstrate that, for equivalent accuracy metrics, it is possible to utilize 3.8 times fewer resources, operate at 1.62 times faster clock frequency, and achieve a significant reduction in latency when compared to a direct floating-point core based dot-product. Combining these results and utilizing the spare resources to instantiate more units in parallel, it is possible to achieve an overall speed-up of at least 5 times.

1 Introduction

The dot-product computation, also known as vector scalar product or vector-by-vector multiplication, is a basic operation in linear algebra. This operation is also a building block in other fundamental algebraic operations such as matrix-by-vector, and matrix-by-matrix multiplications. All these operations are recurrent and central to many scientific algorithms, which range from solution finding for systems of linear equations [1] to the generation of complex biomedical images [2]. With their prolific employment, and often intensive computational requirements, it is important to explore methods that allow the acceleration of this basic operation.

* The authors would like to acknowledge the support of the EPSRC (Grant EP/C549481/1 and EP/E00024X/1) and the contributions of Dr. Eric Kerrigan.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 157–168, 2010.

In computer arithmetic there are a two widely used number systems, fixed-point and floating-point [3]. The former representation is commonly found in digital signal processors. This number system is fast, requires a modest amount of resources, and is well suited to modern commercial FPGA architectures. The latter number system is more flexible and has the advantage of supporting a wide range of values. Nonetheless, the digital logic required by floating-point is more complex, and this complexity comes at a significant performance and silicon cost.

Power consumption has become an important issue in accelerating scientific computing using typical high performance microprocessors, hence it has become increasingly important to explore alternative means of acceleration. This has positioned Field Programmable Gate Arrays (FPGAs) as a key component in the exploration of highly optimized reconfigurable architectures where speed-up can be provided by exploring wide-parallelism, deep-pipelining, fast and efficient datapaths, and through the usage of customized number systems.

In a typical microprocessor a floating-point unit can take operands from any source computation. But in a custom hardware accelerator, we have prior knowledge of the source and relationship between each of the operands. We can therefore take advantage of this knowledge and only perform normalizations, denormalizations, and alignments [4] when necessary. We extend this reasoning to allow for a customized hardware block where the interfaces abide by floating-point standards but internally the design is optimized for efficient FPGA implementation.

The main contributions of this paper are thus:

- a parameterizable hybrid floating-point and fixed-point dot-product design,
- a report on performance, latency, and resource utilization results for the proposed hybrid design (FX) and a standard floating-point (FP) core based design,
- an empirical error analysis for the absolute and relative solution error for both FX and FP designs,
- a comparison of the solution accuracy versus resource utilization trade-offs reported by FX and FP demonstrating that in most of the design space the FX implementation is superior.

After discussing the relevant background in Section 2, we present an overview of the hybrid dot-product method in Section 3. Section 4 details the proposed hardware design as well as a direct floating-point dot-product implementation. In the same section, we make a comparison between resource utilization, performance, and latency for the implementations. Section 5 describes a precision study based on a Monte-Carlo simulation. Section 6 presents the most significant findings of this work, showing that the proposed design offers a superior trade-off between accuracy and resource utilization. Section 7 concludes the paper.

2 Background

A dot-product is defined as:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = r, \tag{1}$$

where a and b are both vectors of order n, and r is the resulting real-valued scalar quantity. This operation involves adding-up the pairwise products of the two vectors, and requires $2n-1$ scalar operations. Taking advantage of the associativity of addition over the real numbers, dot-products can be highly parallelized and deeply-pipelined, as shown in Fig. 1, and used in [1,4,5].

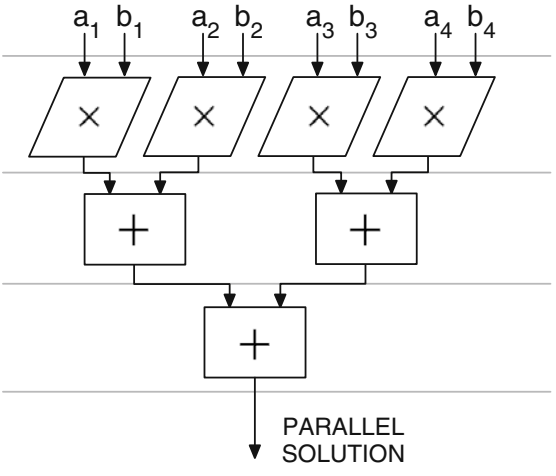


Fig. 1. A parallel dot-product of order 4

A number x is represented in floating-point using the notation in (2), where S represents a sign bit, M a mantissa, and E an exponent.

$$x = (-1)^S \times 1.M \times 2^{E-bias} \tag{2}$$

In accordance with the IEEE 754 standard for binary FP arithmetic the mantissa is an unsigned number and a normalized floating-point number will always have a single one in the most-significant-bit (MSB) position. This bit is implicit and therefore the mantissa does not need to store it. The exponent is represented as a non-negative integer from which a constant *bias* is subtracted.

Floating-point multiplications require a number of stages besides the multiplication of the mantissas and the addition of exponents. These stages include the normalization, rounding, and re-normalization of the mantissa, if necessary [3]. In the case of floating-point additions two initial stages are required, one to

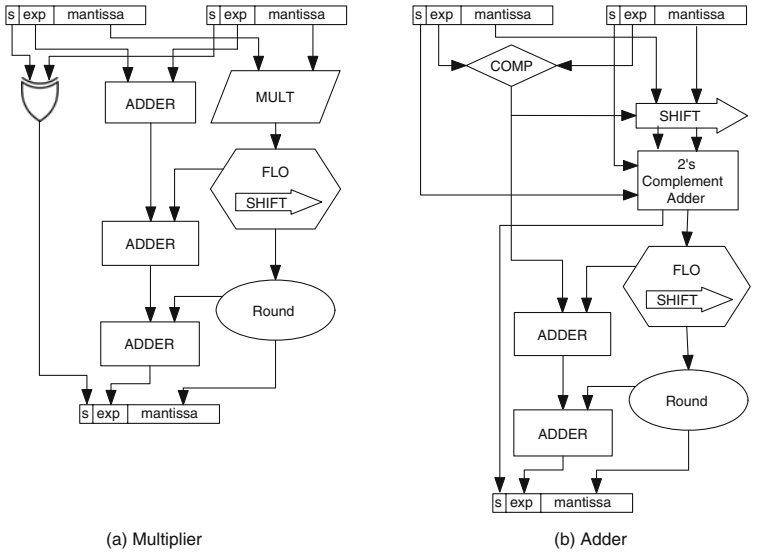


Fig. 2. Floating-point multiplier and adder diagrams showing the alignment stage in the adder and the normalization, rounding and re-normalization stages on both operations. FLO represents "finding leading one".

detect which exponent is the highest, and another to align the mantissa of the lowest number to the same magnitude of the larger number. These stages are illustrated in Fig. 2.

Floating-point arithmetic defined by the IEEE 754 standard [3] applies to atomic scalar operations, not to composite computations such as dot-product. As such, because of the non-associativity of floating-point addition, re-ordering of operands changes roundoff error. Thus we should see floating-point realizations of dot-products as producing a "family" of possible arithmetic accuracies rather than one single accuracy. Our scheme aims to be indistinguishable from this family under an appropriate measure of accuracy, while out-performing a straight-forward floating-point core based implementation.

In the fully parallelized and deeply pipeline dot-product circuit depicted in Fig. 1, where each floating-point operation output is connected to an adder input, there is a recurrent connection between a normalization, rounding and re-normalization circuit and a mantissa alignment circuitry. This recurrent logic consumes significant resources and increases the latency of such operations. In this work we address this wastage and propose a dot-product design which fuses the entire dot-product datapath. The design inputs two floating-point number vectors and generates a floating-point output. Internally it makes use of a configurable word-length fixed-point number system, which eliminates the recurrent normalization and alignment stages present in straight-forward floating-point implementation. In this proposed implementation, the customizable word-length in

the adder reduction-tree increases by 1-bit at each stage, automatically avoiding overflow. The corrected exponent is separately calculated and produced at the output of the dot-product circuit.

In previous related work, Underwood has performed a study which compared the performance of dot-products in FPGAs and CPUs [6]. In this 2004 paper it was predicted that FPGA-based floating-point operations would overtake CPUs by at least an order of magnitude by 2009. In this paper, we demonstrate that such performance is indeed possible by configuring FPGA resources into highly optimized parallel and pipelined datapaths.

In [7], a number of reduction circuits, which are an inherent part of dot-product operation, are studied. For these circuits, the authors have set three fundamental requirements. These include: no stalling on multiple input sets of arbitrary size, the number of adders must be fixed, and buffers are only allowed to grow up to a reasonable size. The proposed designs successfully address the target requirements, and solves the problem arising in summation-reduction using high latency pipelined floating-point cores. In contrast our design aims at reducing overall latency and minimize area consumption by deviating from a floating-point core-based design.

In [8], the authors propose a floating-point fused dot-product unit. This unit is limited to order 2, nonetheless significant improvements are reported. These improvements include a reduction of latency by 80% when compared to a fully parallel approach and 50% when compared to a serial approach. In terms of resource utilization, the fused approach saves 42% when compared to a fully parallel implementation.

The FloPoCo project [9] is an open-source initiative which provides a framework to automatically generate floating-point operators. The precision of these operators can be customized to exploit and maximize the flexibility provided by FPGA. The generated VHDL code is synthesizable and portable to any mainstream FPGA architecture. However there is no primitive to generate a highly efficient dot-product datapath.

In [4], Langhammer proposes a tool that automatically fuses a floating-point datapath. In this paper it is reported that efficiencies gained by fusing the entire datapaths result in a typical 50% improvement in terms of logic, latency, and power reduction. The focus is on an automated flow taking best advantage of the underlying architecture for non-standard floating-point, rather than on the application-specific hybrid representation we discuss.

In this work we present a fused-hybrid dot-product design that can provide up to a 5 times speed-up in throughput as well as a reduction in latency by 5 times.

3 Hybrid Dot-Product Design

The proposed design considers the dot-product operation in its entirety. This allows for the elimination of redundant circuitry that is present in straightforward floating-point core based implementations. This redundant circuitry, as

illustrated in the Section 2, consumes significant resources, accounts for the comparatively high latency, and increases the complexity of placement and routing which in turn lowers the overall operating frequency. Our proposed design is fully parallelized and deeply pipelined, and can receive a new set of input vectors at each clock cycle.

This proposed design, as depicted in Fig. 3, comprises of a number of interconnected blocks. At the inputs it receives two sets of n single precision floating-point numbers. Each of these floating-point numbers is partitioned into its sign-bit, exponent-bits, and mantissa-bits. At this initial partitioning, the leading hidden bit which can be 1 or 0, depending on the exponent, is concatenated on to the mantissa. Subsequently a pair-wise multiplication is performed between corresponding mantissas of each vector, exponents are added together, and respective sign-bits XORed. At the output of the multiplication stage, the mantissas can be truncated or expanded into a desirable internal word-length. This allows us to trade-off resource utilization with solution accuracy, as described in Section 5. In parallel, all the exponents are compared in a tree with $\lceil \log_2 n \rceil$ stages. The highest exponent becomes the reference and all the mantissas are aligned to this reference. After this alignment, the mantissas are converted into 2's complement number representation. In the next stage, the sum-reduction is performed on the mantissas, and at each level of this reduction-tree the word-length is increased by 1-bit, to prevent overflow. After this summation, the fixed-point number is reconverted to sign and magnitude, aligned so as to drop the leading one, and the exponent corrected. From these values, an IEEE 754 compliant floating-point number is generated and output.

4 Implementation

We have implemented a number of designs; the first set of designs is a reference core based floating-point implementation, which we will refer to as "FPn" for an order-n dot-product; the other set of designs are based on our proposed scheme, as described in the previous section, we will refer to these as "FXn". Both sets were targeted, placed and routed onto a Xilinx Virtex6LX760-2 FPGA, and the toolchain used was Xilinx ISE version 11.3. This toolchain was set to optimize for speed with maximum effort. These parameterizable designs can also be trivially modified to operate on Altera FPGAs.

For all designs, the input and output format uses IEEE 754 single precision floating-point. However, in both sets of designs, it is possible to trade-off area against accuracy; in the FP designs this can be done by using a greater precision internally to the dot-product, *e.g.* double precision. In our design, this trade-off is achieved by varying the fixed-point word-length, p.

4.1 Resource Utilization

To measure resource utilization as a function of the internal word-length, p, we have selected Look-Up-Tables (LUTs) slices as the resource of interest because

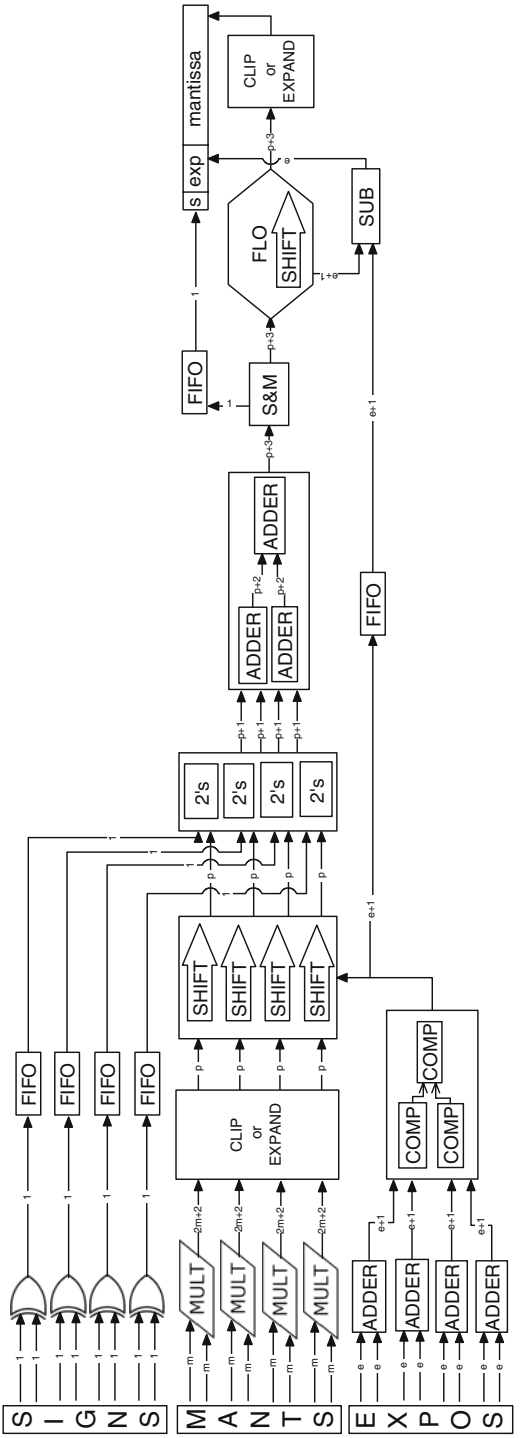
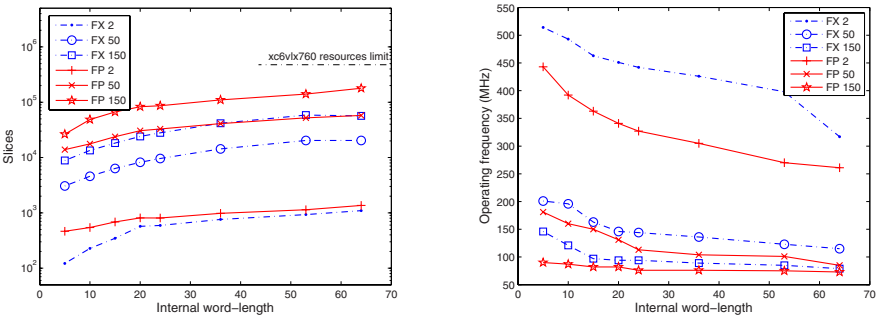


Fig. 3. The proposed hybrid floating-point and fixed-point dot-product circuit for $n = 4$. In this circuit, data flows from left to right, and each signal is annotated with its word-length. Floating-point numbers are multiplied together to produce $n(2m + 2)$-bit mantissas. After this multiplication, these mantissas are truncated or expanded to the desired internal precision p. Subsequently, all mantissas are aligned to the highest exponent. After the alignment, all mantissas are converted to the two's complement representation and added together. With the resulting sum, the first leading one (FLO) is searched and its position is used to produce an aligned mantissa and the final corrected exponent. The aligned mantissa may be truncated or expanded before being output. The FLO block generates an output in a single clock cycle, and this unit is the one that relatively exhausts the highest number of resources.



(a) Slice utilization (b) Operating frequency

Fig. 4. Resource utilization and performance for the Virtex6 LX760

these are the limiting factor on the target FPGA. In Fig. 4(a) there are three lines describing slice utilization for each the two sets of implementations. Resource utilization growth is linear with input vector order. This growth becomes quadratic when varying the internal word-length. Both sets of designs utilize the same number of DSP48 blocks.

It is important to note that a fixed-point implementation with p internal bits will, in general, have a different solution accuracy when compared to a floating-point implementation of p internal bits. This issue is addressed in Section 5, where we show our approach provides a superior trade-off for comparable accuracy.

4.2 Performance and Latency

The operating frequency as a function of internal word-length, p, has been depicted in Fig. 4(b) for both sets of implementations. Since these circuits are fully pipelined and parallelizable their performance in terms of operations per second is given by $(2n-1) \times$ frequency.

The latencies for the FX and the FP sets of circuits, as a function of input vectors orders n, are described in (3) and (4) respectively. In (4), L_m and L_s represent the latencies of the individual floating-point cores for multiplication and addition, respectively. These latencies vary with word-length, and in the case of 24-bit mantissas (IEEE 754 single precision), L_m is 8 and L_s is 12, when utilizing the floating-point modules from the Xilinx CoreGen library. Thus the latency of the proposed scheme is also superior when $n \geq 5$.

$$\text{FX Latency}(n) = 2\lceil \log_2 n \rceil + 29 \tag{3}$$

$$\text{FP Latency}(n) = L_m + L_s \lceil \log_2 n \rceil \tag{4}$$

5 Precision Study

In any finite number representation numbers have a limited accuracy. This accuracy can be improved by increasing the word-length, which in bit-parallel hardware translates to more resources being allocated. In this section, we create two random test-benches for vectors of order 2 and 150. Each of these test-benches comprises of 10000 randomly generated pairs of vectors. Each of these vectors comprises of single precision floating-point numbers with exponents uniformly distributed between -50 and 50, and mantissas uniformly distributed over their natural range.

We have also implemented a software dot-product which emulates the hardware. This software utilizes the MPFR library which is a C library for multiple-precision floating-point computations with correct rounding [10]. Utilizing this software and emulating the dot-product with 128-bits of internal precision, we have generated our reference values. From these values we were able to calculate the error as function of word-length and of resource utilization, as described in the following section.

5.1 Word-Length and Error

Running the 10000 randomly generated vectors and varying the word-length we have produced the two plots as depicted in Fig. 5; the first plot shows circuits with input vectors of order 2; the other plot depict circuits with input vectors of order 150. Note that since the error is data dependent, we use the measures of accuracy given in (5) where S denotes the input test set; $test_i$ is the result produced by the unit under test; and ref_i is the result produced by our high accuracy MPFR implementation.

$$\text{error} \quad = \underset{i \in S}{min/max/median} \left| \frac{ref_i - test_i}{ref_i} \right| \tag{5}$$

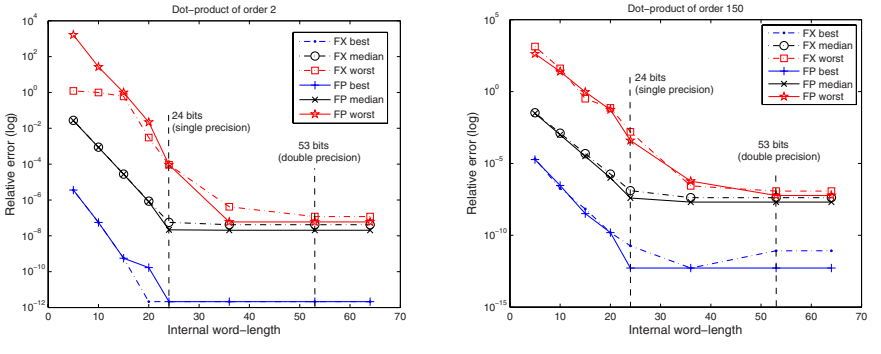


Fig. 5. Error as a function of word-length for dot-product order 2 and 150

It is possible to observe that the floating-point error lines become flat for word-lengths significantly greater than 24-bits. This reflects the precision of the input values which are all single precision.

6 Error and Resource Utilization

In the previous section we demonstrated how the internal word-length affected the solution accuracy. In this section we translate word-length into resource utilization and show how it is possible to trade-off resources with solution accuracy. It is further demonstrated that by using our proposed design, it is possible to achieve the same median accuracy, in terms of relative and absolute error, while saving significant resources when compared to the FP design. This is demonstrated through the four plots in Fig. 6. For example, using our proposed implementation, with 24 internal bits, it is possible to achieve a better solution accuracy than using the straight-forward floating-point implementation with 20 internal bits, while consuming almost 4 times less resources. The solution accuracy of our design levels-off at a slightly lower accuracy then the straight-forward floating-point scheme. Therefore, beyond this level of accuracy, the only option is to use the FP implementation. Apart from this extreme case, our proposed

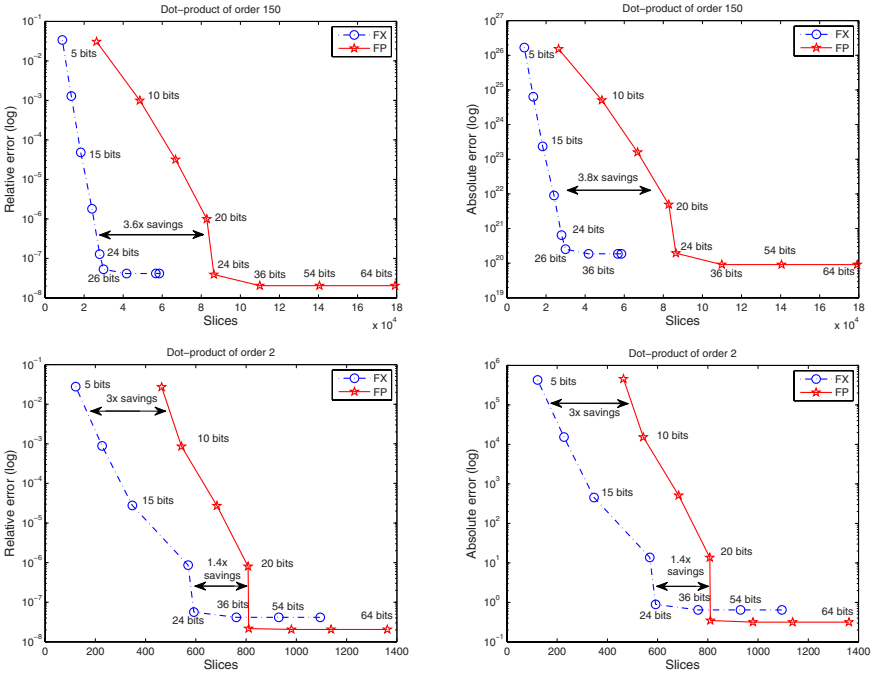


Fig. 6. Error as a function of slices utilization for dot-product of order 2 and 150

design provides a significantly better trade-off between solution accuracy and resource utilization for all of its accuracy range. These plots were generated using median error values, however it is also important to note that both the minimum and maximum error values provide similar trade-offs.

7 Conclusions

This paper proposes new FPGA-based dot-product design which takes advantage of deep-pipelining, wide-parallelism, highly-efficient datapaths, and makes use of a hybrid number representation system. This design inputs and outputs floating-point numbers, but internally makes use of a customizable dynamic fixed-point (FX) number representation. It analyzes and compares the resource utilization, performance and latency of the a hybrid design and a direct core based floating-point design. An empirical study of solution accuracy as a function of resource utilization is presented. From this study it is demonstrated that the newly proposed design can provide better solutions utilizing significantly fewer resources.

It is demonstrated that it is possible to utilize 3.8 times fewer resources, operate at 1.62 times faster clock frequency, and achieve a significant reduction in latency when compared to a floating-point based dot-product. Combining these results and utilizing the spare resources, to instantiate more units in parallel, it is possible to achieve an overall speed-up of at least 5 times.

Future work could be focused on further improvements to the accuracy, resource utilization, and latency by using various number partitioning schemes.

References

1. Boland, D., Constantinides, G.: An FPGA-based Implementation of the MINRES algorithm. In: Proc. of Field Programmable Logic, pp. 379–384 (2008)
2. Junaid, K., Ravindrann, G.: FPGA Accelerator For Medical Image Compression System. In: Proc. of International Conference on Biomedical Engineering, pp. 396–399 (2006)
3. IEEE, 754 Standard for Binary Floating-Point Arithmetic (1985), `http://grouper.ieee.org/groups/754/` (accessed on 25/10/2009)
4. Langhammer, M.: Floating point datapath synthesis for FPGAs. In: Proc. of Field Programmable Logic and Applications, pp. 355–360 (2008)
5. Roldao, A., Constantinides, G.: High Throughput FPGA-based Floating Point Conjugate Gradient Implementation. In: Proc. of Applied Reconfigurable Computing, pp. 75–86 (2008)
6. Underwood, K., Hemmert, S.: Closing the gap: CPU and FPGA Trends in sustainable floating-point BLAS Performance. In: Proc. of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 219–228 (2004)
7. Zhuo, L., Morris, G., Prasanna, V.: High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs. IEEE Transactions on Parallel and Distributed Systems 18, 1377–1392 (2007)

8. Saleh, H., Swartzlander, E.: A Floating-point Fused Dot-product Unit. In: Proc. of IEEE International Conference on Computer Design, pp. 427–431 (2008)
9. Dinechin, F.: FloPoCo is a generator of arithmetic cores (Floating-Point Cores), http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/ (accessed on 19/10/2009)
10. Collaborative Project, Multi-precision Floating Point Library, http://www.mpfr.org/ (accessed on 02/01/2009)

Optimising Memory Bandwidth Use for Matrix-Vector Multiplication in Iterative Methods

David Boland[1] and George A. Constantinides[2]

Electrical and Electronic Engineering Department,
Imperial College London, London SW7 2AZ, UK
dpb03@ic.ac.uk, george.constantinides@ieee.org

Abstract. Computing the solution to a system of linear equations is a fundamental problem in scientific computing, and its acceleration has drawn wide interest in the FPGA community [1, 2, 3]. One class of algorithms to solve these systems, iterative methods, has drawn particular interest, with recent literature showing large performance improvements over general purpose processors (GPPs). In several iterative methods, this performance gain is largely a result of parallelisation of the matrix-vector multiplication, an operation that occurs in many applications and hence has also been widely studied on FPGAs [4, 5]. However, whilst the performance of matrix-vector multiplication on FPGAs is generally I/O bound [4], the nature of iterative methods allows the use of on-chip memory buffers to increase the bandwidth, providing the potential for significantly more parallelism [6]. Unfortunately, existing approaches have generally only either been capable of solving large matrices with limited improvement over GPPs [4, 5, 6], or achieve high performance for relatively small matrices [2, 3]. This paper proposes hardware designs to take advantage of symmetrical and banded matrix structure, as well as methods to optimise the RAM use, in order to both increase the performance and retain this performance for larger order matrices.

1 Introduction

The large amount of resources available on modern FPGAs have made them suitable for accelerating floating point applications. The solution to a system of linear equations is a recurring sub-problem within many scientific computing problems [7], and hence there is considerable value in accelerating this operation. Iterative methods are one type of algorithm to solve a system of linear equations and recently studies have shown that by using FPGAs on these algorithms it is possible to achieve performance improvements of up to an order of magnitude over general purpose processors (GPPs) [2, 3].

The reason FPGAs are capable of accelerating iterative methods, such as the conjugate gradient and minimum residual (MINRES) algorithms [8], is that these algorithms often contain lots of inherent parallelism, the majority of which originates from a repeated matrix-vector multiplication. Furthermore, as it can

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 169–181, 2010.

be shown that in general this operation consumes the best part of the execution time of the algorithms [9], parallel execution of this operation significantly reduces the overall execution time. Unfortunately, highly parallel matrix-vector multiplication circuits require the use of the on-chip RAM to buffer data so as to provide the desired bandwidth, and hence the available RAM on the FPGA limits the maximum matrix order that can be implemented.

Given many problems in scientific computing result in large matrices, it is of interest to determine the extent to which this performance can be maintained for such matrices. To achieve this, this paper proposes hardware architectures for performing matrix-vector multiplication that can take advantage of banded matrix structure, symmetry within the matrix, or both. Banded matrices are sparse matrices of a specific structure such that all of the non-zero values lie within a specified bandwidth of the diagonal, and these arise in many problems, for example when solving partial differential equations [10]. Symmetric matrices are square matrices that are equal to its transpose, and these are of particular interest as both conjugate gradient and MINRES algorithms will only converge to a solution provided the input matrix is symmetric.

This paper will demonstrate that by exploiting these properties, it is possible to reduce the RAM requirements. Furthermore, as embedded RAMs on FPGAs have specific structures, this paper proposes an optimisation strategy using integer linear programming (ILP) in order to translate this reduced RAM requirement into the minimum use of embedded RAMs. Finally, it will be shown that by applying these strategies for saving RAM, it is possible to move the source of limitation on parallelism from RAMs to be a function of the look-up tables and dedicated multipliers which are used to construct floating point components. As a result, this work also goes on to describe parameterisable hardware architectures, depending upon matrix characteristics such as the bandsize and matrix order, which can scale to larger matrices and obtain as much parallelism as possible. The main contributions of this paper can be summarised as follows:

- Hardware architectures for banded matrices and symmetric matrices that can significantly extend the scalability to large order matrices and achieve higher degrees of parallelism,
- An optimisation strategy to reduce the number of embedded RAMs depending upon problem specification,
- Hardware architectures that can trade parallelism with FPGA resources to achieve greater scalability.

This paper begins with a survey of existing implementations of matrix-vector multiplication in Section 2, before describing our architectures in Section 3. Some results showing the benefit of this approach are then given in Section 4, before the work is concluded in Section 5.

2 Related Work

There has been a large amount of research into FPGA acceleration of floating point matrix-vector multiplication. The two main factors that distinguish these

approaches are the method to store the matrix and how the on-chip RAM is utilised.

At one extreme is the work by El-Kurdi *et al.* [5]. This implements a streaming approach such that the 'stripes' containing the non-zeros for the matrix and the vector are held in off-chip RAM and streamed through a set of processing elements, with one processing element for each stripe to achieve the maximum parallelism. The advantage of this approach is that due to the streaming nature, it can operate on arbitrarily large matrices, provided there is sufficient off-chip RAM. The disadvantage of this approach is that the maximum number of stripes and hence the maximum parallelism is limited by the I/O bandwidth from a 1.76 single precision GFLOPs peak on a Stratix S80 to 1.5 single precision GFLOPs.

The work by Morris *et al.* [1] is slightly different, storing the matrix in a more traditional fashion, using Compressed Sparse Row (CSR) format [11], which consists of all non-zero values of the matrix, an index to the column the value lies in, and an index for when each new row begins. The hardware then must match several matrix values with their corresponding vector element and performs the multiplications in parallel, before accumulating the results. In comparison to the work by El-Kurdi *et al.*, it stores the vector on-chip to perform these parallel multiplications and this slightly improves the maximum performance. However, it is still limited by I/O, and storing these vectors on chip means its scalability depends on the available RAM to store these vectors. Further work by Zhuo *et al.* [4,12] examined in detail the floating point data hazards, improving the reduction circuits that accumulate these results to use less silicon, but the performance was still limited by I/O to be 2.88 single precision GFLOPs, or 2.16 GFLOPs in practical simulations on a Virtex2 Pro.

DeLorimier and DeHon [6] create an implementation which similarly targets sparse matrix-vector multiplication for matrices stored in CSR format, but it is specifically aimed at accelerating this function within iterative methods. This operation is special as the same matrix is used for every iteration, and hence this approach suggests loading the matrix into on-chip embedded RAM once, from which it can be re-used multiple times allowing much more parallelism as a result of the significantly higher memory bandwidth. Using this method, it achieved a performance of up to 1.5 sustained double precision GFLOPs on a Virtex2-6000. However, the maximum matrix size and performance in this work is limited by the available memory to store the matrix.

The work by Lopes *et al.* [2] and previous work by the authors [3] acknowledge that the more modern FPGAs have much larger memories and the floating point support has improved, and hence maximise the performance of the conjugate gradient algorithm and MINRES respectively, by storing on dense matrices using the on-chip RAM. With dense matrices, there is no need for matching vector elements, and hence matrix-vector multiplication can be achieved easily using a pipelined dot-product core consisting of a vector multiplier and adder-tree, as shown in Figure 1. In both works, this proved to be the major performance increase, with the latter reporting up to 53 sustained single precision GFLOPs on a Virtex5 330, which could translate to approximately a factor of 10 performance

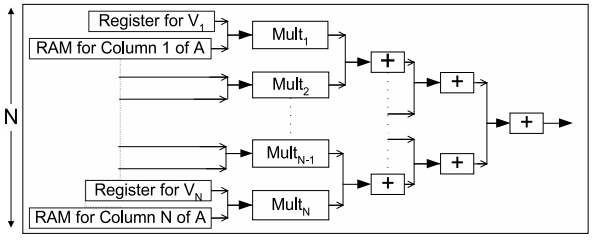


Fig. 1. Dot Product Circuit

increase over the peak theoretical performance of a Pentium 4, for matrices of orders up to 145.

The work by Lopes *et al.* was then extended for banded matrices to examine RAM savings [13]; this allowed the maximum order to be extended from 92 in the dense case to 236 in the banded case for a thin band size of 5. However this work only implemented a basic architecture which performs parallel multiplication for the size of the band, but stores the entire vector in registers meaning that resource use still grows with matrix order, an approach which will be shown to be inefficient.

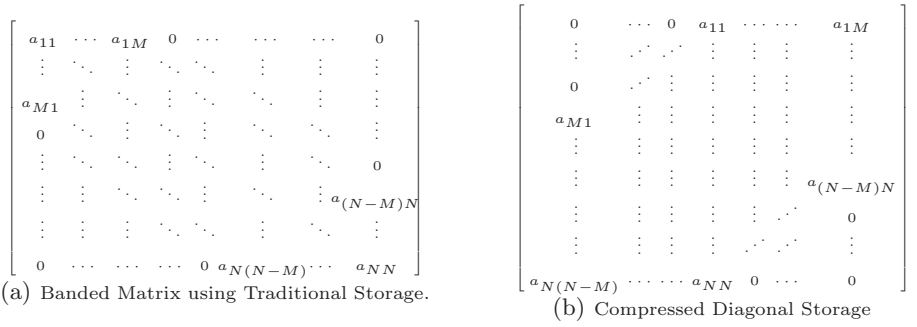
The aim of this work is to describe hardware architectures and RAMs configurations to perform the matrix-vector multiplication with the minimum hardware in such a way that they could easily be plugged into an implementation of an iterative method such as [2] or [3].

3 Performing Matrix-Vector Multiplication

This section describes simple modifications to the architecture as shown in Figure 1 to solve matrices with specific structures using a high level of parallelism. We begin with a detailed description of banded matrices before describing the hardware architectures and RAM configurations to implement matrix-vector multiplication for this type of matrix, discussing in detail how this same approach can be used to handle both thin and wide bands. We then describe how this approach can easily be extended to handle symmetric matrices, reducing the RAM requirements, before discussing our procedure to optimise the use of RAM and LUT resources on an FPGA given this RAM requirement. Finally, we discuss our approach to trade parallelism for scalability for larger matrices.

3.1 Matrix-Vector Multiplication for Banded Matrices

Banded matrices are matrices where all the non-zero elements lie within some known bandsize M from the main diagonal, as shown in Figure 2(a). As the location of the non-zeros is known *a priori*, simple structures can be used to hold these values such as Compressed Diagonal Storage (CDS) [11], shown in Figure 2(b).



(a) Banded Matrix using Traditional Storage.

(b) Compressed Diagonal Storage

Fig. 2. Methods to Store a Banded Matrix. Each column will be stored in a separate RAM, as shown in Figures 1 and 5.

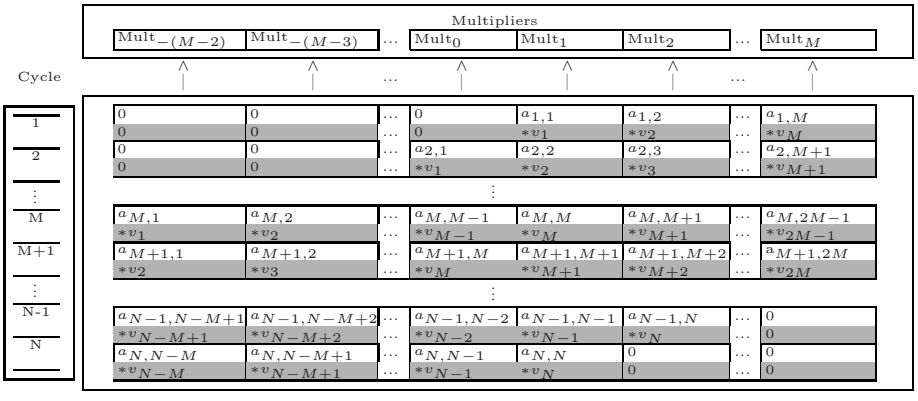


Fig. 3. Required Multiplication over time. In this figure, the values in grey represent the required vector elements, whilst the values in white represent the required matrix elements from RAM. Any 0 value refers to a multiplication that need not be performed.

Using CDS to store the matrix, all zeros that do not fall into the band are not stored. This corresponds to $(N-M)(N-M+1)$ saved elements. However, as is clear from Figure 2(b), there are still some zeros in this storage. These zeros do not reflect any in the original matrix, rather they reflect the fact that at the band ends there are no elements and hence zeros are added instead. This corresponds to a total of $M(M+1)$ additional zeros. This implies that if $2M-1 > N$, the amount of added zeros created from this redundancy could be greater than the number of zeros that are avoided by using this storage format. This section discusses these cases separately.

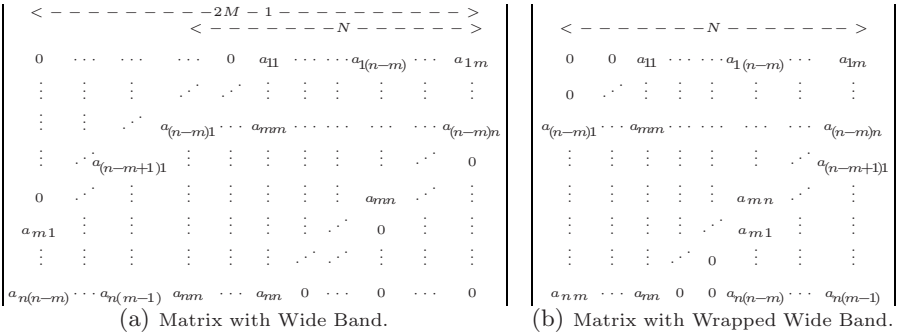
Thin-Bands ($2M-1 \leq N$). In comparison to the method for dense matrices, the first difference is that instead of using N parallel multipliers, it is only necessary to perform parallel multiplications for the bandsize $(2M-1)$, as the

result of any other multiplications would be zero. The other slight complexity is that if the matrix is stored using CDS, the associated vector element for each RAM will change at each cycle. This is demonstrated in Figure 3 which shows the desired multiplications over time.

However, from Figure 3, it should be clear that the required vector element for each multiplier is simply shifted once per clock cycle. This would require little additional hardware in comparison to Figure 1, which uses a vector of registers, as the shift could be achieved using a serial-in-parallel-out shift-register. Furthermore, this shift register need only be of size $2M - 1$, as opposed to a vector of N registers.

Wide-Bands ($2M - 1 > N$). There are two issues when using wide bands. The first is the excessive storage, as mentioned above, the other is that when using a banded matrix, the number of parallel multiplication is equal to $2M - 1$, but if $2M - 1 > N$, this would mean the number of multipliers is greater than the size of the vector, and hence any such multiplications would correspond to a multiplication by zero.

As a result, in order to minimise resources, the number of parallel multipliers should be restricted to N. To map this to the RAMs, the proposed solution to 'wrap' the data in the RAM around N columns, as shown in Figure 4. The vector can also easily be 'wrapped' by feeding the output of the final output of the shift register back into the input, and adding a multiplexer to choose between this input and the vector input, this is shown in Figure 5. The control for this multiplexer is simple and requires little additional hardware.



(a) Matrix with Wide Band. (b) Matrix with Wrapped Wide Band.

Fig. 4. Wrapped Wide Bands

Whilst using N columns of RAM to store the matrix appears to be no better than a dense implementation, there are two main benefits to this wrapping approach. The first is that it allows the same hardware to be used for both cases; the second is that, excluding the dense case, using the optimisation process described in Section 3.3, it is possible to save some RAM.

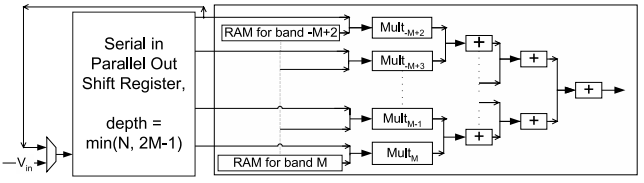


Fig. 5. Banded Dot Product Circuits

3.2 Matrix-Vector Multiplication for Symmetric Matrices

With symmetric matrices, it is only necessary to either store the lower or upper diagonal matrix. Interestingly, extending CDS (Figure 2(b)) to only store the symmetric portion is straightforward: all that needs to be done is to remove the columns that only hold the redundant data, *i.e.* all the columns to the left of that holding the diagonal. However, whilst this reduces the RAM requirements, in order to use the same architecture (Figure 5), one must emulate the behaviour of the extra RAMs used for band storage. Interestingly, the organisation of the RAMs in CDS makes it quite simple to achieve this. Observing Figure 2(b), the values that have been removed in the symmetric storage are simply seen to be a delayed version of other columns, the required delays shown in Figure 6.

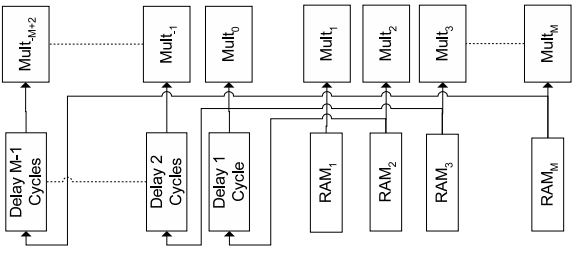


Fig. 6. Symmetric Shift Register

Implementing Delays for Symmetry on FPGAs. Using FPGAs, there are three potential methods to create this delay. The simple method would be to use FIFOs made up of either shift registers or RAMs. The problem with using this method is that if the delay is large, these FIFOs may also become large and this may use a lot of resources.

Alternatively, some FPGAs have embedded RAMs which can implement true dual-port memory [14]. In this case, one port could access the current value, and the other port select the delayed value, meaning the delay could then be implemented simply by using a delayed counter which would require minimal additional circuitry. However, there are more subtle issues when using embedded RAMs. Xilinx BRAMs on a Virtex 5 are 36KBit, and can be configured in one of two ways: as 2-18KBit Block RAMs implementing simple dual-port RAM;

or one single 36KBit true dual-port RAM [14]. This implies that by using the Block RAMs in true dual-port fashion, the amount of flexibility of the RAMs is reduced. Viewing this in another way, the likelihood of a large portion of an embedded RAM being empty is heavily increased, and this can reduce the number of RAMs available and impact the potential parallelism.

The final choice is that if the delay required for symmetry is greater than the size of RAM needed to store a column, then the same RAM can be used to also feed the multiplier for the symmetrical delay without requiring a second port, as only one of the two multipliers will require input data at any given time. Given these three options, determining the optimum use of resources is described in the following section.

3.3 Minimising RAM Use

The amount of RAM required to store the matrix is dependent upon the the size of matrix N, the bandsize M, and the number of LUTs on the FPGA the user is willing to allow to be used in the place of embedded RAMs. In order to optimise the configuration, this section proposes an integer linear programming (ILP) formulation, which can be solved by many existing solvers, such as CPLEX [15]. A high-level description of this ILP is given in Figure 7(a), with the formal ILP problem given in Figure 7(b). This section discusses how this formal ILP is obtained.

Given input matrices of order N and bandsize M for P problems,
min: *Overall RAM Use*

subject to:
Matrix Memory Constraints
Symmetric Delay Constraints

Available Register Constraints

(a) ILP Problem General Description.

$$\textbf{min:} \sum_{i=1}^{M} 2x_i + y_{1i} + y_{2i}$$

subject to:

$$\forall i, 2Bx_i + By_{1i} + z_{1i} \geq r_i$$
$$\forall i, 2Bx_i + By_{2i} + z_{2i} \geq s_i$$
$$\sum_{i=1}^{M} z_{1i} + \sum_{i=1}^{M} z_{2i} \leq R$$
$$\forall i, x_i, y_{1i}, y_{2i}, z_{1i}, z_{2i} \in \mathbb{Z}$$

(b) Formal ILP Problem.

Fig. 7. Minimising RAM using ILP

Notation. As shown in Figure 2(b), the matrix columns in CDS contain trailing zeros. It is not necessary to store them, and hence the RAM requirement for each column decreases. In contrast, as shown in Figure 6, the symmetric delay required increases for each column. As each column has different requirements, the variable i is used as an index for the M columns, with $i = 1$ being the column containing the diagonals. For the ILP, the values for the RAMs required and symmetric delay required for column i can then be denoted as r_i, and s_i respectively. The maximum capacity of the BRAMs in terms of the number of words they can store is denoted as B, and the number of registers allocated by the

user as an alternative to embedded RAMs as R; the choice of R will typically be the number of unused registers, and hence will vary depending upon the type of FPGA and the number of resources used for other functions in a given hardware implementation.

There are three choices that to store matrix elements and to implement the delays: true dual-port RAM, simple dual-port RAM or shift-registers, and as described in Section 3.2, true dual-port RAM can both store matrix elements and implement symmetric delay, whereas separate simple dual-port RAM or shift-registers are needed to implement this delay. To simplify the notation, for each column i, integer variables which represent the number of true dual-port RAMs, simple dual-port RAMs, simple dual-port RAMs used for delay for symmetry, shift-registers and shift-registers used for delay for symmetry, are denoted as $x_i, y_{1i}, y_{2i}, z_{1i}, z_{2i}$ respectively. As the RAMs and shift registers are physical components, these must be integer variables, making this an ILP.

Objective Function. The aim is to minimise the RAM use, so the objective function is a summation of the variables for the various RAMs. However, as the true dual-port RAMs are twice the size of the simple dual-port RAMs, the cost for all x_i variables is twice that of y_{1i} and y_{2i}.

Matrix Memory and Symmetric Delay Constraints. The three types of storage must satisfy the matrix memory and symmetric delay constraints for each column. The complexity in this approach is that, as mentioned in Section 3.2, the true-dual port rams contribute to both the symmetric delay and matrix memory constraints, and thus this same variable appears in both inequalities.

It should be noted that the values r_i and s_i must be calculated prior to implementing the ILP. As i increases, the RAM requirement for each column of a matrix incrementally decreases, and hence the memory requirement for r_i is given by $(N-i+1)$. Furthermore, in previous works [6,2,3] it has been highlighted that due to the deep pipelines in floating point operators on FPGAs, in order to maintain high sustained performance it was necessary to perform matrix-vector multiplication on many different problems in a pipeline, and each of these problems would have to be stored in RAM. This can easily be incorporated into the model by modifying the memory requirement for P problems to be $P(N-i+1)$. In contrast, as i increases, the symmetric delay requirement for each column increases incrementally, but for delays, it is no longer necessary to store multiple problems, and hence s_i can be found to be $i-1$. Also, as mentioned in Section 3.2, if $i > N$, then there is no need for a separate RAM to implement the extra delay, and hence s_i is given by $i-1$ if $i \leq N$ and 0 if $i > N$.

Finally, one should note that by replacing symmetric delay constraints with extra memory constraints, it is possible to use this same ILP for banded matrices.

Available Register Constraints. A final constraint is added to allow a user to trade BRAM with registers. This is likely to be problem dependent, determined by the FPGA resources used elsewhere and the total available resources.

3.4 Trading Performance with Slices

It will be shown in Section 4 that the significant reduction in memory use that this method provides implies that it would no longer be the memory available that limits the maximum matrix order of the matrix-vector multiplication, rather the slices and multipliers become the limiting factor.

The reason for the growth in slices is that the number of floating point units required to perform the parallel multiplication grows according to $\min(N, 2M-1)$. The solution would be to perform partial multiplications of size $\lceil \min(N, 2M-1)/\alpha \rceil$, where α is an integer, and use a reduction circuit, several of which are discussed in [12], to sum these partial multiplications. Any problems caused by the ceiling function are avoided by extra multiplications by zero. As it is desirable to achieve as much parallelism as possible, α should be as small as possible. The circuit required for the case of $\alpha = 2$, which would perform half the multiplication of the first half of the row during odd cycle and the second during even cycles, is shown in Figure 8. For different choices of α, only the reduction circuit would change, many of which are discussed in [12].

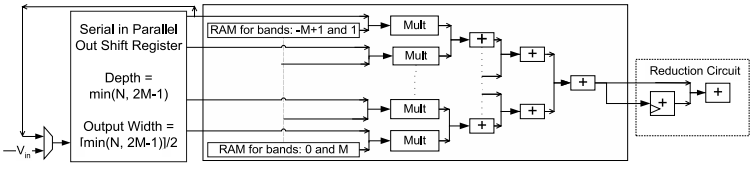


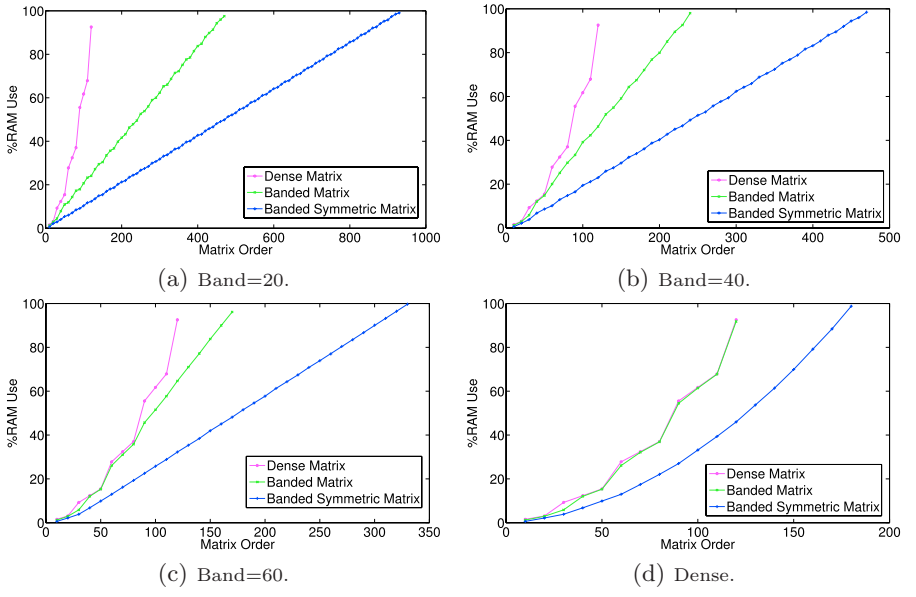
Fig. 8. Example Parallel Circuit for Large Dense Matrices

The problem with this circuit is that parallelism is reduced by approximately a factor of α and hence the maximum performance of the circuit will similarly decrease. The counter side is that the number of slices will decrease by approximately the same factor as the number of multipliers and size of the reduction tree is reduced. This method could therefore be used for any general circuit to trade resources in terms of DSPs and registers for scalability.

4 Results

4.1 RAM Use

The main benefit of this work is that it significantly reduces the RAM use. Figure 9 consists of four graphs showing the percentage of embedded RAMs of a Virtex 5 LX330T that are required to hold banded matrices of warying widths. In these examples, the number of pipelined problems has been set to $P = 20$, whilst the number of registers that can be used instead of RAMs has been set to be equal to the size of one simple dual-port RAM, which corresponds to approximately 0.5% of the slices of the FPGA.



(a) Band=20. (b) Band=40.

(c) Band=60. (d) Dense.

Fig. 9. RAM Use

The greater scalability of this approach is clearly shown for the thinnest bandsize, when $M = 20$, which demonstrates that a large amount resources can be saved in comparison to the basic method storing a dense matrix, the maximum matrix order can be extended from 120 to 470 in the banded case and 930 in the symmetric banded case. It should be noted that this bandwidth would, at the maximum, require 39 parallel floating point multiplications, and hence could not be fed using off-chip RAM. As the bandsize increases, though this difference gets smaller, it is still significant. However, it is interesting to note that the difference between the dense and banded case decreases much faster than the difference between the dense and the symmetric case. The reason for this is that the symmetric delay is only a function of N, whereas storing the band instead of implementing this delay is a function of N and P. It is also worth noting that in the graph where there is a wide band of $M = 60$, there is indeed still RAM savings using the banded format as opposed to storing it in the dense format, as mentioned in section 3.1.

4.2 Parallelism

The other claims of this work are that by reducing the amount of RAM used, it is possible to obtain greater parallelism, and this parallelism becomes limited by the registers and DSPs and hence it becomes necessary to trade parallelism for scalability. In order to demonstrate this, Figure 10 compares the resource use, post place and route, of matrix vector multiplication for a dense matrix of increasing order, using the architecture from Figure 1 that was used in previous

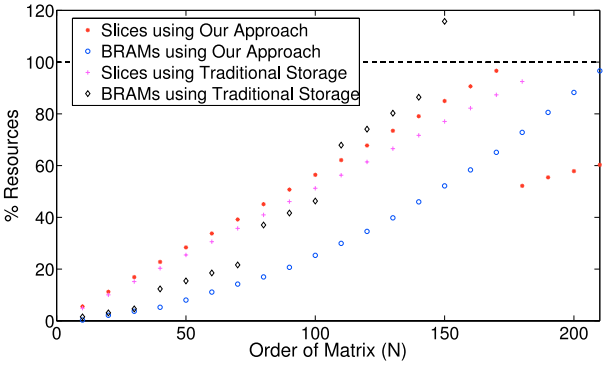


Fig. 10. Resource Use

works and this approach. The number of problems has been set to $P = 14$ for this is what was used for the largest matrix in [3], the previous work which claims the highest performance.

This graph demonstrates several points of interest. Firstly it shows that this work approaches the maximum performance achievable on an FPGA for matrix-vector multiplication in IEEE single precision floating point, in that it uses almost all of the slices and DSPs, with most of these being used as floating point components. This is unlike the method for traditional storage, in which case the RAM limits the maximum possible parallelism; it is clear, for a matrix order of 150, that the RAM required for traditional storage exceeds that available on the FPGA. It is also interesting that the transition for RAMs is much smoother for our method than for using traditional storage; this is a result of our approach reducing the probability RAMs being largely empty, as mentioned in Section 3.2. In addition, this graph demonstrates the value of reducing the parallelism to increase the scalability. Finally, one should note that whilst there is still limited scalability, this is due to the focus on a dense matrix with a large number of pipelined problems. However, this is nonetheless a valuable test as this component could easily be plugged into the iterative solvers in [2,3] to improve their performance and scalability.

5 Conclusion

Overall, this work has described how to create a parameterisable circuit to implement matrix-vector multiplication that could be plugged into existing hardware implementations of iterative methods. Furthermore, it has shown that by taking into account symmetry and banded matrices, only simple hardware changes to an implementation of matrix-vector multiplication circuit using a pipelined dot-product circuit, along with an optimisation strategy for RAM use, are required to significantly improve both the scalability and performance of the circuit.

Finally, one should note that any algorithm containing matrix-vector multiplication which is suitable for on-chip buffering of data could use this circuit, whilst the contributions to reduce RAM use on FPGAs could be applied to any circuit that stores banded or symmetric matrices on-chip.

References

1. Morris, G.R., Prasanna, V.K., Anderson, R.D.: A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer. In: Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines, pp. 3–12 (2006)
2. Lopes, A.R., Constantinides, G.A.: A high throughput FPGA-based floating point conjugate gradient implementation. In: Proc. Applied Reconfigurable Recomputing, pp. 75–86 (2008)
3. Boland, D., Constantinides, G.: An FPGA-based implementation of the MINRES algorithm. In: Proc. Int. Conf. Field Programmable Logic and Applications, September 2008, pp. 379–384 (2008)
4. Zhuo, L., Prasanna, V.K.: Sparse matrix-vector multiplication on FPGAs. In: Proc. ACM/SIGDA 13th Int. Symp. on Field-Programmable Gate Arrays, pp. 63–74. ACM, New York (2005)
5. El-Kurdi, Y., Gross, W.J., Giannacopoulos, D.: Sparse matrix-vector multiplication for finite element method matrices on FPGAs. In: Proc. IEEE Symp. Field-Programmable Custom Computing Machines, pp. 293–294 (2006)
6. de Lorimier, M., DeHon, A.: Floating-point sparse matrix-vector multiply for FPGAs. In: Proc. ACM/SIGDA 13th Int. Symp. on Field-Programmable Gate Arrays, pp. 75–85. ACM, New York (2005)
7. Heath, M.T.: Scientific Computing. McGraw-Hill Higher Education, New York (2001)
8. Golub, G.H., Loan, C.F.V.: Matrix computations, 3rd edn. Johns Hopkins University Press, Baltimore (1996)
9. Hoekstra, A.G., Sloot, P., Hoffmann, W., Hertzberger, L.: Time complexity of a parallel conjugate gradient solver for light scattering simulations: Theory and spmd implementation, Tech. Rep. (1992)
10. Sewell, G.: The numerical solution of ordinary and partial differential equations. Academic Press Professional, Inc., San Diego (1988)
11. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., der Vorst, H.V.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd edn. SIAM, Philadelphia (1994)
12. Zhuo, L., Morris, G.R., Prasanna, V.K.: High-performance reduction circuits using deeply pipelined operators on FPGAs. IEEE Trans. Parallel Distrib. Syst. 18(10), 1377–1392 (2007)
13. Lopes, A., Constantinides, G., Kerrigan, E.C.: A floating-point solver for band structured linear equations. In: Proc. Int. Conf. Field Programmable Technology, pp. 353–356 (2008)
14. Xilinx, Virtex-5 FPGA User Guide
15. Ilog, Inc., Solver cplex (2009), `http://www.ilog.fr/products/cplex/` (accessed November 2, 2009)

Design of a Financial Application Driven Multivariate Gaussian Random Number Generator for an FPGA

Chalermpol Saiprasert, Christos-Savvas Bouganis,
and George A. Constantinides

Department of Electrical & Electronic Engineering, Imperial College London
Exhibition Road, London SW7 2BT, United Kingdom
{cs405,christos-savvas.bouganis,g.constantinides}@imperial.ac.uk

Abstract. A Multivariate Gaussian random number generator (MVGRNG) is a pre-requisite for most Monte Carlo simulations for financial applications, especially those that involve many correlated assets. In recent years, Field Programmable Gate Arrays (FPGAs) have received a lot of attention as a target platform for the implementation of such a generator due to the high throughput performance that can be achieved. In this work it is demonstrated that the choice of the objective function employed for the hardware optimization of the MVRNG core, has a considerable impact on the final performance of the application of interest. Two of the most important financial applications, Value-at-Risk estimation and option pricing are considered in this paper. Experimental results have shown that the suitability of the chosen objective function for the optimization of the hardware MVRNG core depends on the structure of the targeted distribution. An improvement in performance of up to 96% is reported for VaR calculation while up to 81% improvement is observed for option pricing when a suitable objective function for the optimization of the MVRNG core is considered while maintaining the same level of hardware resources.

1 Introduction

Monte Carlo simulation plays an important role in many scientific applications, one of which is financial mathematics. The multivariate Gaussian distribution is a pre-requisite for such simulations as it captures the correlation between sources of uncertainties that affect the values of the financial instruments. As the number of financial instruments continues to increase, the computation of these simulations has been intensified. Field Programmable Gate Arrays (FPGAs) have been demonstrated to be a good candidate for the acceleration of random number generators due to their fine grain parallelism, and many works have been presented in recent years on the acceleration of many financial applications [1],[2]. However, to the best of the authors' knowledge, no published work in the literature has addressed the question regarding the relative performance of the various optimization objective functions in the design of a random number

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 182–193, 2010.

generator block focusing on the impact that this has on the performance of a financial application.

In this work, we focus on the implementation of the Multivariate Gaussian random number generator (MVGRNG) on an FPGA platform and we investigate the impact of the design decisions taken for the optimization of the MVGRNG to the performance of a financial application. The application under investigation are the estimation of the Value-at-Risk (VaR) of a financial portfolio and option pricing, two of the most widely used applications in financial industry. Existing approaches in the literature regarding an FPGA based MVGRNG can be found in [3], [4] and [5]. The work presented in this paper is based on the framework proposed in [5], as it can accommodate any objective function for the hardware optimization of a MVGRNG design. Three design criteria are proposed for the design of a MVGRNG targeting an FPGA device, and their impact to the estimation of VaR calculation and option pricing are investigated for a set of hardware resources.

2 Related Work

The first FPGA-based multivariate Gaussian random number generator was presented in [3]. The authors decompose the input correlation matrix, which encapsulates the correlation of the distribution of interest, using Cholesky decomposition in order to take advantage of the lower triangular property of the resulting matrix. The resulting design is able to serially generate a vector of multivariate Gaussian random numbers every N clock cycles, where N denotes the dimensionality of the distribution. In [3], DSP48 blocks are used for the implementation of a MVGRNG on an FPGA platform, requiring N blocks for an N-dimensional Gaussian distribution. However, the drawback of this approach is the restriction in resource allocation since the dimensionality of the distribution dictates the number of DSP48 blocks to be utilized. In their approach, the minimization of the mean square error between the approximated correlation matrix and the target one is implemented.

An alternative method which addresses the problem encountered in [3] is presented in [4]. An algorithm, based on the use of Singular Value Decomposition, is introduced to approximate the lower triangular matrix, the result of applying Cholesky decomposition on the correlation matrix, by trading off the error in the approximation of the input correlation matrix for an improved resource usage. The approach in [4] requires $2K$ DSP48 blocks to produce a vector of size N, where K denotes the number of decomposition levels required to approximate the lower triangular matrix while maintaining the same throughput as in [3]. In addition to an improved resource utilization, [4] offers the flexibility to produce a hardware system that meets any given resource constraint. However, it has been shown in [4] that allocating a fixed precision to all of the computation paths of the architecture does not lead to the optimum resource utilization.

In [5], the precision issue mentioned above has been exploited and word-length optimization techniques have been introduced to produce an architecture with

multiple precisions in its datapath. The algorithm presented in [5] further reduces the required hardware resources in comparison to [3] and [4]. In addition, an analysis of the correlation of errors due to truncation operations in the computation datapath has been presented and its effects have been modeled in the objective function targeting the optimum usage of the hardware resources.

To the best of the authors' knowledge, the existing approaches in the literature have not investigated the impact of the objective function employed in the optimization of the MVGRNG core on the performance of the application that requires such a block and on the hardware requirements of the core. The authors in [3] carried out an evaluation of the performance of their approach for a financial application, the Delta-Gamma asset simulator, but the optimization of the hardware architecture of a MVGRNG over different objective functions has not been considered. This work considers two of the most important financial applications involving Monte Carlo simulation, the estimation of the Value-at-Risk (VaR) and option pricing.

3 Generating Multivariate Gaussian Random Samples

Following from [5], in order to generate random samples from a given multivariate Gaussian distribution with mean $\mathbf{m}$ and correlation $\boldsymbol{\Sigma}$, the eigenvalue decomposition technique is used [6]. In this technique, an algorithm known as Singular Value Decomposition is used to decompose the correlation matrix $\boldsymbol{\Sigma}$. As a result of the decomposition, $\boldsymbol{\Sigma}$ can be expressed as a linear combination of three separable matrices $\mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$ where $\mathbf{U}$ is an orthogonal matrix ($\mathbf{U}\mathbf{U}^T = \mathbf{I}$) containing eigenvectors $\mathbf{u}_1, ..., \mathbf{u}_N$ while $\boldsymbol{\Lambda}$ is a diagonal matrix with diagonal elements being eigenvalues $\lambda_1, ..., \lambda_N$. If we let $\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}^{1/2}$, multivariate Gaussian random samples that follow $N(\mathbf{m}, \boldsymbol{\Sigma})$ can be generated as in (1), where $\mathbf{z} \sim N(\mathbf{0}, \mathbf{I})$.

$$\begin{aligned}\mathbf{x} &= \mathbf{A}\mathbf{z} + \mathbf{m} = \mathbf{U}\boldsymbol{\Lambda}^{1/2}\mathbf{z} + \mathbf{m} \\ &= (\sqrt{\lambda}_1\mathbf{u}_1 z_1 + \sqrt{\lambda}_2\mathbf{u}_2 z_2 + ... + \sqrt{\lambda}_K\mathbf{u}_K z_K) + \mathbf{m} \\ &= \sum_{i=1}^{K}(\sqrt{\lambda}_i\mathbf{u}_i z_i) + \mathbf{m}. \end{aligned} \tag{1}$$

The full representation of the original correlation matrix $\boldsymbol{\Sigma}$ can be achieved if the number of decomposition levels K is equal to the rank of the correlation matrix. Thus, the correlation matrix of the original distribution can be approximated by taking into account K levels of decomposition where $K \leq rank(\boldsymbol{\Sigma})$.

4 Hardware Architecture

The proposed hardware architecture for a multivariate Gaussian random number generator for an FPGA implementation is based on the eigenvalue decomposition technique. The generation of random samples from a centralized Gaussian

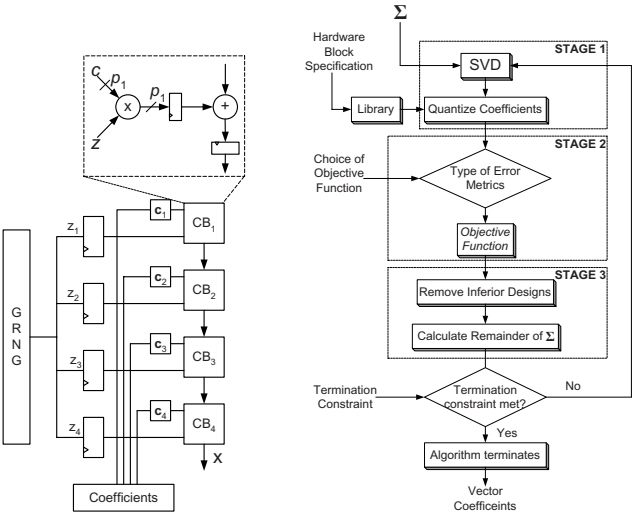
distribution, that is a distribution with zero mean, will be the main focus for the remainder of this paper. Any other non-centralized distribution with the same correlation can be produced by a simple offset of the generated vectors. From (1), the correlation matrix under consideration $\boldsymbol{\Sigma}$ is expressed as $\mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^T$. Hence, the random samples $\mathbf{x}$ can be generated as $\mathbf{x} = \sum_{i=1}^{K} \sqrt{\lambda_i}\mathbf{u}_i z_i \approx \sum_{i=1}^{K} \mathbf{c}_i z_i$, where $\mathbf{c}_i$ denotes a product of $\sqrt{\lambda_i} \cdot \mathbf{u}_i$ after a quantization step with the desired word-length.

In this work, the multivariate Gaussian samples are generated from the sum of products of $\mathbf{c}$ and z and, thus, the various decomposition levels i are mapped onto computational blocks (CB) designed for an FPGA. Each CB performs the multiply-add operation. The architecture is mapped to logic elements only, so that the precision of the datapath can be varied. Using the proposed approach, a MVGRNG design is constructed from any combinations of CBs. Figure 1(a) depicts an example of such architecture where four CBs are used. p_i denotes the precision of the datapath for each level of decomposition while the precision in the adder path p_T is fixed to the maximum precisions of all CBs. Independent univariate Gaussian samples z are produced from the GRNG block.

The multiply-add operation is pipelined so that all the computational blocks operate in parallel in order for an improved throughput. Fixed point number representation is deployed throughout the entire design as it produces designs which consume fewer resources and operate at a higher frequency in comparison to designs that use floating point arithmetic. Similar to other existing approaches in the literature, the elements of each random vector are serially generated resulting to a throughput of one vector per N clock cycles for an N dimensional Gaussian distribution [3] [4] [5].

5 Constructing MVGRNG Blocks

In this section, the methodology of mapping the hardware architecture described in Figure 1(a) to an FPGA is discussed. The main concept behind the methodology used in this work is the exploration of mixed precisions in the computational path of the architecture which was proposed in [4]. The high level overview of the proposed methodology is depicted in Figure 1(b). There are three main stages in the proposed methodology where, in the first stage, the correlation matrix $\boldsymbol{\Sigma}$ is decomposed into a vector $\mathbf{c}$ and its transpose using the Singular Value Decomposition (SVD) algorithm. The resulting vectors are converted into fixed point representation using one of the user-specified word-lengths from the hardware library. In the second stage, the appropriate coefficients are selected in order to minimize the error metric according to the selected objective function which will be described in the next section. The third stage of the proposed approach removes any inferior designs, that is designs which in comparison to another design, use more hardware resources but produce worse approximation error. The last step of this stage calculates the remainder of the original matrix $\boldsymbol{\Sigma}$ which is the starting point for the next iteration. These steps are repeated until the termination condition is met. Examples of the termination condition



(a) Architecture of Multivariate Gaussian Random Number Generator.

(b) Overview of the Proposed Approach.

Fig. 1. Proposed Methodology

are a specified approximation error, a given resource constraint or the number of decomposition levels to be used. The output of the proposed approach is a set of vectors of coefficients $\mathbf{c}_i$ which best approximate the input correlation matrix under a given norm.

The approximated correlation matrix $\overline{\boldsymbol{\Sigma}}$ can be decomposed as $\overline{\boldsymbol{\Sigma}} = \mathbf{C}_K + \mathbf{W}$, where $\mathbf{C}_K$, approximates the input correlation using K levels of decomposition taking into consideration the quantization effects and $\mathbf{W}$ is an expected truncation error matrix, which is a function of the correlation between the error due to truncation operations [5].

In summary, a framework to design a hardware architecture of a MVGRNG with the ability to accommodate different objective functions has been discussed. The coefficients of the correlation matrix of the distribution of interest are chosen based on the objective function of interest. The proposed approach provides an expectation of the generated correlation matrix given the coefficients in the various levels of the architecture taking into consideration all sources of errors injected into the system. It should be noted that the proposed system takes into account the correlation effect due to truncation error in the approximation of the correlation matrix. In the next section, the quality of the random samples produced based on the proposed methodology are used for the estimation of the Value-at-Risk of a financial portfolio and the pricing of options so the impact of the different objective functions on the two financial applications can be investigated.

6 Case Study: MVGRNG in Financial Applications

This section focuses on the two financial applications of interest namely, the estimation of Value-at-Risk of a portfolio and the pricing of an option. The first part of this section provides a background description of the two applications. The discussion is then shifted towards correlated assets where a multivariate Gaussian random number generator is deployed to model the correlation between those underlying assets. Three different objective functions are introduced for the hardware design of the MVGRNG, given a certain constraint on the available resources, and their impact on the performance of the two financial applications is investigated.

6.1 Value-at-Risk of a Financial Portfolio

A portfolio is a collection of financial investments such as stocks, bonds and options. Essentially, the purpose of holding a portfolio is to limit the risk while increasing the possibility of making profit, where the investment is spread over a variety of assets with different degree of risk factors. In order to quantify the expected loss of a portfolio with many correlated assets, the progression of the risk factors affecting this portfolio must be taken into account. Monte Carlo simulation is used to simulate the time evolution of the risk factors due to their stochastic nature. Let us consider a portfolio containing N assets. The market price of each asset is denoted by S_i having drift μ_i and volatility σ_i, where $i = 1 \ldots N$. The correlation between all of the assets in the portfolio under consideration is captured by a correlation matrix $\boldsymbol{\Sigma}$. The dynamics of the price of each asset are modeled as in (2).

$$\ln(S_i(t_E)) = \ln(S_i(t_0)) + \sum_{j=1}^{E} (\mu_i \delta t + x_j), \tag{2}$$

where x_j denotes a multivariate Gaussian random sample that follows $N(0, \boldsymbol{\Sigma})$. The time interval from t_0 to t_E is divided into intervals of length δt [7]. The path taken by the random walk algorithm during the simulation is of interest only for *path-dependent* derivatives such as an Asian-style derivative where the price at maturity is the average price of the path taken over a specified period. A European-style derivative, on the other hand, does not take this into account and the price at maturity is taken at the end of the specified period. Value-at-Risk (VaR) is a measurement used to evaluate a risk of loss of a specific portfolio and describes probabilistically the market risk of a trading portfolio by measuring the worst expected loss over a specific time interval at a given level of confidence.

6.2 Option Pricing

An option is a contract between a buyer and seller which permits a buyer, depending the type of option held, the right to purchase or sell a particular asset at an agreed price on or before the option's expiration time. In this work, the

option called "Chooser option" is considered. The strike price, which is a price when the option is being exercised, of a Chooser option with three underlying assets is determined by the maximum of the sum of the two highest priced assets at the end of a specified period of time. Monte Carlo methods are used to calculate the value of these options where multiple underlying assets are involved and sources of uncertainty are captured by the multivariate Gaussian distribution.

6.3 Objective Functions

In this work, we propose three objective functions for the selection of coefficients and the precision of the computational paths for each decomposition level in the design of the MVGRNG core. The following notation is used for all three objective functions. $\mathbf{R}_{K-1}$ denotes the remaining of the original correlation matrix after $K-1$ levels of decomposition with $\mathbf{R}_0$ defined as the original correlation matrix at the start of the algorithm. The max(.) operator is an element-wise operation for a matrix of interest. Table 1 illustrates the three objective functions to be used in this work. The first objective function selects the coefficients based on the minimization of the maximum absolute error in the approximation of the correlation matrix while the second and third objective functions minimize the relative and mean square error in the correlation matrix approximation respectively. All three objective functions take into account the correlation effect between the truncation errors to the final correlation matrix.

Table 1. Objective Functions

Error to be Minimized	Objective Functions
Maximum Absolute Error	$\max \left\lvert \mathbf{R}_{K-1} - (\mathbf{c}_K \mathbf{c}_K^T + \mathbf{W}) \right\rvert$
Maximum Relative Error	$\max \left\lvert \frac{\mathbf{R}_{K-1} - (\mathbf{c}_K \mathbf{c}_K^T + \mathbf{W})}{\mathbf{R}_{K-1}} \right\rvert$
Mean Square Error	$\frac{1}{N^2} \left\lVert \mathbf{R}_{K-1} - (\mathbf{c}_K \mathbf{c}_K^T + \mathbf{W}) \right\rVert^2$

6.4 Framework

Figure 2 shows the framework deployed in this work. There are two main components in the system. The first part generates random samples from a given multivariate Gaussian distribution which captures the correlation between the assets of interest. These random numbers are produced by the proposed system where the generator is mapped onto an FPGA using mixed precision in the computational paths. The second part of the framework performs the simulation of the two financial applications under consideration using the randomly generated data.

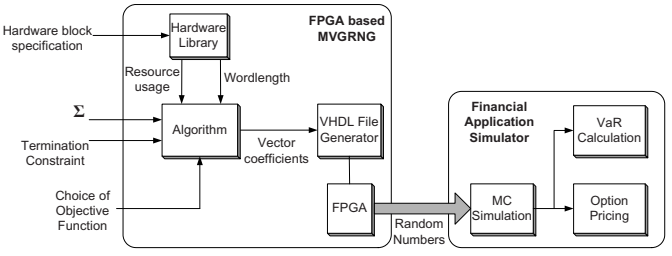


Fig. 2. High Level Overview of Framework

6.5 Experimental Setup

The purpose of this investigation is to assess the performance of the random number generator core when the already mentioned objective functions are used for its optimization, for a given resource constraint, using the calculation of VaR and option pricing application as testbenches. Since we are interested in the impact of the objective function for the design of the MVGRNG hardware block on the performance of the applications under investigation, it is adequate for this work for the two applications to be implemented on a CPU. The target device for the MVGRNG is a Stratix III EP3SE50F484C2 FPGA from Altera and Quartus II is utilized as a hardware synthesis tool. The precision of the univariate Gaussian random samples z is kept constant throughout the design at 18 bits with 3 bits dedicated for the integer part and 14 bits dedicated for the fractional part. The set of computational blocks (CBs) that is used in the proposed framework is chosen in order to cover the range between 10 to 18 bits precision in an almost uniform way where three CBs with 10, 14 and 18 bits precision have been pre-defined in the hardware library. After the synthesis stage from Quartus II, the resource utilization of 212, 296 and 332 LUTs is reported for CBs using 10, 14 and 18 bits precision respectively. These results are used by the three objective functions under consideration in order to optimize for the desired architecture. In this work two types of input correlation matrices are considered, Type I denotes matrices with high cross correlation between the underlying assets whereas Type II denotes correlation matrices with very low cross correlation.

6.6 Experiment I: Calculation of VaR

In the first experiment, four portfolios each with five correlated assets are considered. The underlying correlations between the five assets for each portfolio are modeled by the four correlation matrices, namely **A**, **B**, **C** and **D**. **A** and **B** are of Type I while **C** and **D** are of Type II. Cross correlation values for Type I matrices are above 0.9 while that of Type II matrices are below 0.01. A fixed resource constraint is used as a terminating condition for the three objective functions of interest where the constraint is set to 1660 LUTs since 1660 is equivalent to five DSP48 blocks which is the amount of hardware resources required by [3], an

approach that is based solely on DSP48 blocks for producing multivariate Gaussian random numbers on an FPGA platform, for a 5x5 correlation matrix.

In total, four MVGRNG hardware blocks are investigated and compared. The first three generators are implemented on FPGA using the proposed methodology optimizing for three different error metrics as seen in objective functions 1, 2 and 3 respectively, see Table 1. The fourth MVGRNG is generated using [3]. The generated random numbers are then used for the calculation of VaR of both European-style and Asian-style derivatives and produce the risk assessment for the four portfolios. The results are compared to a reference design implemented on general purpose processor using double precision floating point number representation using the same "seed" for the univariate Gaussian random number generator block.

6.7 Experiment II: Option Pricing

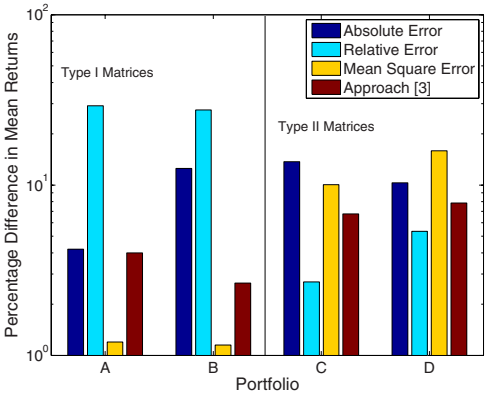
In this experiment two types of Chooser option with 3 assets are considered. The strike price of Chooser option 1 is defined as the sum of all of its assets while that of Chooser option 2 is defined as the maximum of the sum of the two highest priced assets at the end of a specified period of time. Taking into account the two different correlation matrix types I and II we end up with four possible cases of Chooser options, Chooser option 1 and 2 both with high and low cross correlations between their assets. The same procedure as in Experiment I is taken in order to generate the MVGRNG hardware blocks with the exception of the termination condition being set to 996 LUTs for resource constraint which is equivalent to three DSP48 blocks, the amount of hardware required by [3].

6.8 Results: Evaluation of Hardware MVGRNG Blocks

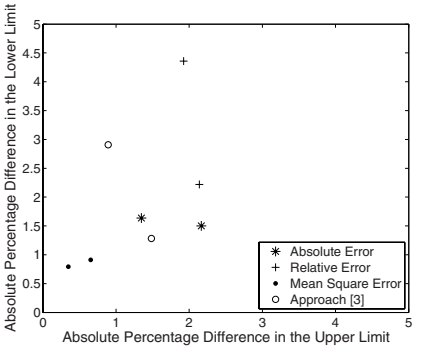
A comparison is made between maturity price of the asset simulated from the random numbers produced from the four different objective functions of interest. We denote the results from the reference design which uses double precision floating point as the actual values. Table 2 summarizes the best objective functions, which produce the closest results for the generation of random numbers from a MVGRNG to the actual values, for each financial application. For both applications where the underlying assets are highly correlated, the best results

Table 2. Summary of Best Performing Objective Function for Financial Applications of Interest

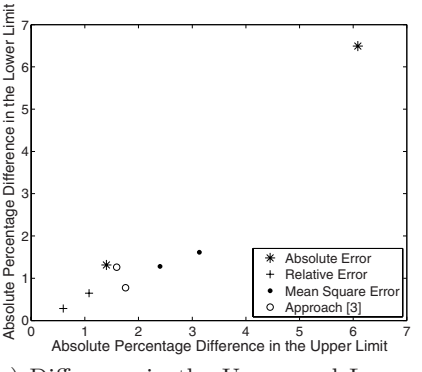
Experiment	Financial Applications	Best Metric	
		Type I	Type II
I	European-style VaR	MSE	REL
	Asian-style VaR	MSE	REL
II	Pricing of Chooser Option 1	MSE	REL
	Pricing of Chooser Option 2	MSE	[3]



(a) Absolute Percentage Difference in the Mean Returns.



(b) Difference in the Upper and Lower Limits of the 95% Confidence Interval for Type I Matrices **A** and **B**.

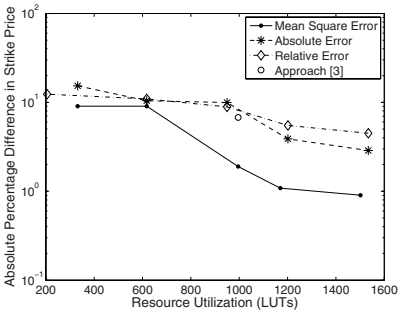


(c) Difference in the Upper and Lower Limits of the 95% Confidence Interval for Type II Matrices **C** and **D**.

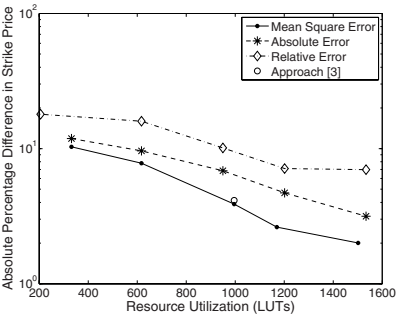
Fig. 3. European-style VaR

are obtained by employing the objective function based on the mean square error. On the other hand, with low cross correlation, the objective function which is based on the relative error is the best metric for three out of four cases with an exception of the pricing of Chooser Option 2 where [3] provides the best results.

A selection of plots from the two experiments are shown to illustrate the behaviour of the two financial applications. Figure 3(a) illustrates a plot of the difference in mean return of each portfolio with respect to the mean return of the reference architecture for a European-style VaR. It can be seen that the mean square error metric gives the nearest approximations to the actual values for Type I matrices while the relative error is the best metric for Type II matrices. It can be deduced from the plots that an improvement in performance within the range of 5% to 96% is reported for Type I matrices while 24% to 80% improvement is observed for Type II matrices using the four different objective functions for the same hardware resource utilization for the MVGRNG block.

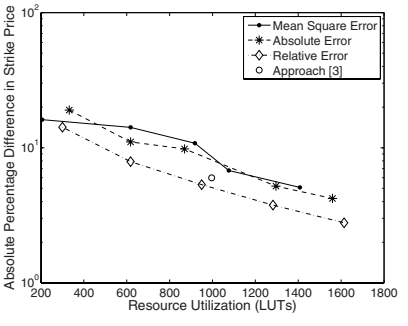


(a) Difference in Strike Price for Chooser Option 1.

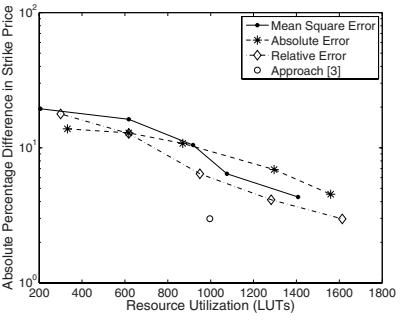


(b) Difference in Strike Price for Chooser Option 2.

Fig. 4. Chooser Options 1 and 2 with Type I Matrix



(a) Difference in Strike Price for Chooser Option 1.



(b) Difference in Strike Price for Chooser Option 2.

Fig. 5. Chooser Options 1 and 2 with Type II Matrix

The upper and lower limits of the value of the returns of each portfolio given a 95% level of confidence are investigated. The upper limit is defined as the maximum expected return while the lower limit is the minimum expected return of each portfolio over a specified confidence interval. The difference of the upper and lower limits for Type I and Type II matrices are plotted in Figures 3(b) and 3(c). The plots reinforce the trend previously observed for the best objective functions for Type I and II matrices.

Figures 4 and 5 illustrate plots of the deviation of the strike price from the actual values for Chooser options 1 and 2 over a range of resources. All graphs show that the deviation from the actual strike price decreases as the resource utilization of the MVGRNG hardware block increases. One important point from these plots is the ability for the proposed approach to produce designs across all design space whereas [3] will only offer one design for a given correlation matrix. The plots indicate that for Type I matrix, an improvement in performance within the range of 17% to 81% and 11% to 71% is reported for Chooser options 1 and

2 respectively using the four different objective functions. For Type II matrix, an improvement in performance within the range of 12% to 51% is reported for Chooser option 1 and 8% to 39% for Chooser option 2 using the three four objective functions.

In terms of the hardware performance, the operating frequency of all designs is in the range of 380 to 420 MHz, where the mean is 401.22MHz.

7 Conclusion

In this paper, a methodology to construct and implement a customized hardware architecture to produce random samples from a multivariate Gaussian distribution tailored made for a specific financial application is described. Three design criteria for the optimization of the MVGRNG are proposed and their impact on the estimation of VaR financial problem and option pricing are investigated. Simulation results have shown that, for an application with highly correlated assets, the objective function which optimizes for the mean square error in the approximation of the target correlation matrix of the multivariate Gaussian distribution provides the best performance for the same hardware resource utilization. On the other hand, for an application with low cross correlation between its assets the objective function based on the relative error provides the best results. An improvement in performance of up to 96% is reported for VaR calculation while up to 81% improvement is observed for option pricing using the four different objective functions.

References

1. Zhang, G., Leong, P.H., Lee, D.-U., Villasenor, J.D., Cheung, R.C., Luk, W.: Ziggurat-based hardware gaussian random number generator. In: Proceedings IEEE International Symposium on Field Programmable Logic and Applications, pp. 275–280 (2005)
2. Woods, N.A., VanCourt, T.: FPGA acceleration of quasi-monte carlo in finance. In: Proceedings IEEE International Conference on Field Programmable Logic and Applications, pp. 335–340 (2008)
3. Thomas, D.B., Luk, W.: Multivariate gaussian random number generation targeting reconfigurable hardware. ACM Transactions on Reconfigurable Technology and Systems 1(2), 1–29 (2008)
4. Saiprasert, C., Bouganis, C.-S., Constantinides, G.A.: Multivariate gaussian random number generator targeting specific resource utilization in an FPGA. In: Woods, R., Compton, K., Bouganis, C., Diniz, P.C. (eds.) ARC 2008. LNCS, vol. 4943, pp. 233–244. Springer, Heidelberg (2008)
5. Saiprasert, C., Bouganis, C.-S., Constantinides, G.A.: An optimized hardware architecture of a multivariate gaussian random number generator. ACM Transactions on Reconfigurable Technology and Systems (2009) (to appear)
6. Chan, N.H., Wong, H.Y.: Simulation Techniques in Financial Risk Management. Wiley, New Jersey (2006)
7. Glasserman, P.: Monte Carlo Methods in Financial Engineering. Springer, New York (2004)

3D Compaction: A Novel Blocking-Aware Algorithm for Online Hardware Task Scheduling and Placement on 2D Partially Reconfigurable Devices

Thomas Marconi, Yi Lu, Koen Bertels, and Georgi Gaydadjiev

Computer Engineering Laboratory, EEMCS
TU Delft, The Netherlands
{thomas,yilu,koen,georgi}@ce.et.tudelft.nl
http://ce.et.tudelft.nl

Abstract. Few of the benefits of exploiting partially reconfigurable devices are power consumption reduction, cost reduction, and customized performance improvement. To obtain these benefits, one main problem needs to be solved is the task scheduling and placement. Existing algorithms tend to allocate tasks at positions where can block future tasks to be scheduled earlier denoted as "blocking-effect". To tackle this effect, a novel 3D total contiguous surface (3DTCS) heuristic is proposed for equipping our scheduling and placement algorithm with blocking-awareness. The proposed algorithm is evaluated with both synthetic and real workloads (e.g. MDTC, matrix multiplication, hamming code, sorting, FIR, ADPCM, etc). The proposed algorithm not only has better scheduling and placement quality but also has shorter algorithm execution time compared to existing algorithms.

1 Introduction

Hardware task scheduling and placement algorithms can be divided into two main classes: offline and online. Offline assumes that all tasks properties (e.g. task sizes, execution times, reconfiguration times, etc) are known in advance. The offline version can then do various optimizations before runtime. As a result, the offline version has a better performance than the online version. However, the offline version is not applicable for general multipurpose systems in which the properties of arriving tasks are unknown beforehand. In general multipurpose systems, the online version is needed.

In the offline version, the algorithm can make offline decisions. Hence, the time needed for making decisions in the offline version is not taken into account for the overall application time. In contrast, the online version needs to take decisions at runtime; as a result, the algorithm execution time is computed as an additional time for the overall application time. Therefore, the goal of the online version is not only to get better scheduling and placement quality but also to have a low runtime overhead.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 194–206, 2010.

Online scheduling and placement algorithms have to find a block of hardware resources for running each arriving task on a 2D partially reconfigurable device. When there are no available resources for allocating the hardware task at its arrival time, the algorithms have to schedule the task for future execution. Here, the algorithms need to find the earliest starting time and free space for executing the task on the device in the future.

Many algorithms have been proposed to solve the scheduling and placement issue mentioned above, such as: Horizon [1], Stuffing [1], Classified Stuffing [2], Intelligent Stuffing [3], Reuse and Partial Reuse [4], Window-based Stuffing [5], and Compact Reservation [6]. However, none of them has a blocking-aware ability; the existing algorithms have a tendency to block future tasks to be scheduled earlier, referred as "blocking-effect". As a result, wasted area (volume), schedule time, and waiting time will increase significantly. To solve this problem, we propose a novel 3D total contiguous surface (3DTCS) heuristic to equip our algorithm with blocking-awareness. The goal of the proposed algorithm is not only to achieve better quality but also to have lower runtime overhead.

The main contributions of this paper are:

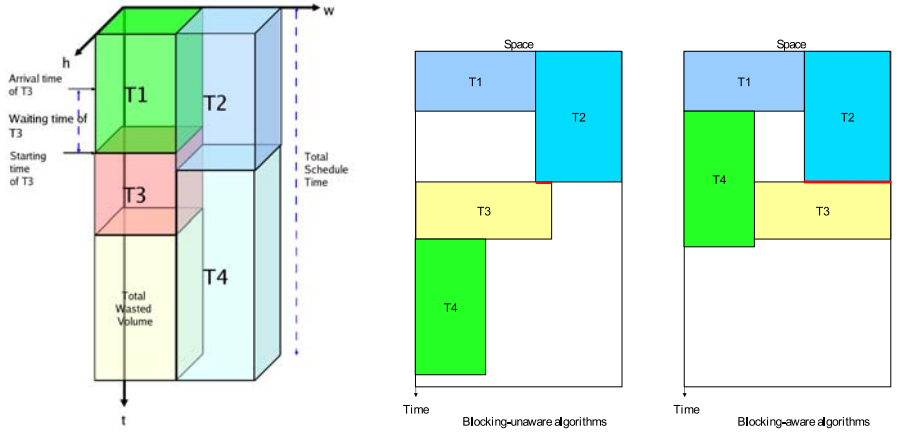
- the first blocking-aware online hardware task scheduling and placement algorithm;
- a novel 3D total contiguous surface (3DTCS) heuristic;
- a novel 3D Compaction (3DC) algorithm.

The rest of the paper is organized as follows. In Section 2, we introduce the problem of scheduling and placement on 2D area models. We give a short review of existing algorithms in Section 3. In Section 4, we present basic idea of our blocking-aware algorithm. The 3DTCS heuristic is described in Section 5. In Section 6, we present our proposed algorithm in detail. The algorithm is evaluated in Section 7. Finally, we conclude in Section 8.

2 Problem of Scheduling and Placement on 2D Area Models

This problem definition is an extension of the definition of the problem of scheduling and placement on 1D area models as presented in [3]. Given a task set representing a multitasking application with their arrival times a_i, life-times lt_i, widths w_i and heights h_i, online scheduling and placement algorithms targeting the 2D area models of partially reconfigurable devices have to determine placements and starting times for the task set such that there are no overlaps in space and time among all tasks. The goals of the algorithms are: a) to utilize effectively the available FPGA resources (minimize wasted volume); b) to accelerate the overall application on the FPGA (minimize schedule time); c) to start executing arriving tasks on the FPGA earlier (minimize waiting time) and d) to keep the runtime overhead low (minimize the algorithm execution time).

We define the total wasted volume as the overall number of area-time units that are not utilized as illustrated in Figure 1(a). Total schedule time is the total



(a) (b)

Fig. 1. Problem of scheduling and placement on 2D area models (a) and basic idea of blocking-aware algorithm (b)

number of time units for the execution of all tasks. Waiting time is the difference between starting and arrival times for each task (in time units). The algorithm execution time is the time needed to schedule and place the arriving task.

3 Related Work

In [1], Steiger et al. proposed the Horizon and Stuffing algorithms both for 1D and 2D area models. The Horizon guarantees that arriving tasks are only scheduled when they do not overlap in time or space with other scheduled tasks. The Stuffing schedules arriving tasks to arbitrary free areas that will exist in the future by imitating future task terminations and starts. In [1], the authors presented that the Stuffing outperforms the Horizon in scheduling and placement quality.

To tackle the drawback of the 1D Stuffing, Chen and Hsiung in [2] proposed their 1D Classified Stuffing. By classifying incoming tasks before scheduling and placement, the 1D Classified Stuffing performs better than the original 1D Stuffing.

In [3], Marconi et al. proposed their 1D Intelligent Stuffing to solve the problems of both the 1D Stuffing and Classified Stuffing. The main difference between their algorithm and previous 1D algorithms is the additional alignment flag of each free segment. The flag determines the placement location of the task within the corresponding free segment. By utilizing this flag, the 1D Intelligent Stuffing outperforms the previously mentioned 1D algorithms.

In [4], Lu et al. introduced their 1D reuse and partial reuse (RPR). The algorithm reuses already placed tasks to reduce reconfiguration time. As a result, the RPR outperforms the 1D Stuffing.

In [5], Zhou et al. proposed their 2D Window-based Stuffing to tackle the drawback of 2D Stuffing. By using time windows instead of the time events, the 2D Window-based Stuffing outperforms previous 2D Stuffing. The drawback of their 2D Window-based Stuffing is a high running time cost. To reduce the high runtime cost of Window-based Stuffing, they proposed their Compact Reservation (CR) in [6]. The main idea of the CR is the computation of the earliest available time (EA) matrix for every incoming task. That contains the earliest starting times for scheduling and placing the arriving task. The CR outperforms the original 2D Stuffing and their previous 2D Window-based Stuffing.

4 Basic Idea of Blocking-Aware Algorithm

Blocking-unaware algorithms do not care of their task placement positions whether they will block other incoming tasks or not in the future. They behave like drivers who park their vehicles wherever they want. Their parking places may be stumbling blocks for other drivers to park their cars. Figure 1(b) (left) illustrates the behavior of online scheduling and placement algorithms that do not have blocking-awareness. In this simple example, task T3 is becoming an obstacle for task T4 to be scheduled earlier.

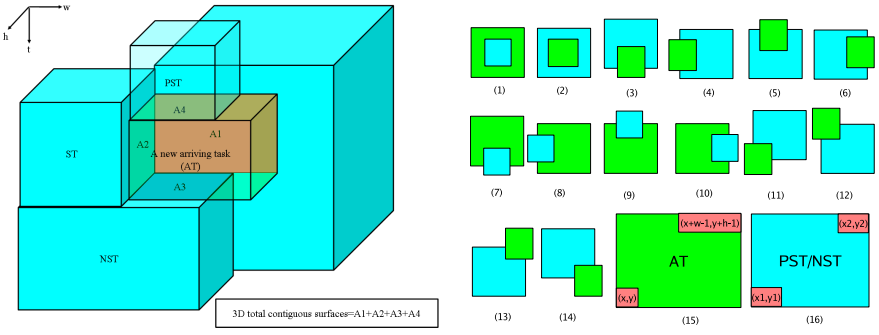
To tackle this problem, we introduce an algorithm that has an awareness to avoid placement that will be an obstacle for other future tasks. By placing task T3 to a different location as shown in Figure 1(b) (right), the proposed algorithm can avoid task T3 to be an encumbrance for task T4 to be started earlier. By scheduling T4 earlier, the FPGA can finish executing task T4 faster. To give the algorithm the necessary knowledge to avoid this "blocking-effect", the algorithm places tasks at locations as much as possible touching its prior tasks illustrated as bold lines on the figure. In next section, we will give a more detail explanation of this heuristic, termed 3D total contiguous surface (3DTCS).

5 3D Total Contiguous Surface (3DTCS) Heuristic

A hardware task on a 2D partially reconfigurable device using 2D area models can be illustrated as a 3D box. The first two dimensions are the required area (wh) on the device for running the task. The other dimension is the time dimension (t). To pack hardware tasks compactly during run time at the earliest time, we propose a new heuristic, named 3D total contiguous surface (3DTCS) heuristic.

The 3DTCS is the sum of all surfaces of an arriving task that is contacted with the surfaces of other scheduled tasks as depicted in Figure 2(a). The 3DTCS contains two components:

- the horizontal contiguous surfaces with previous scheduled tasks and next scheduled tasks;
- the vertical contiguous surfaces with scheduled tasks and the FPGA boundary.



(a) (b)

Fig. 2. 3D total contiguous surface (3DTCS) heuristic (a) and horizontal contiguous surfaces (b)

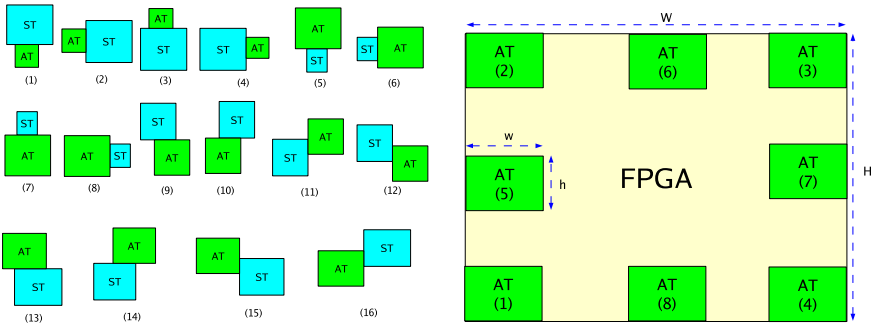
Table 1. Computations of horizontal contiguous surfaces for positions in Figure 2(b)(1)-(14)

Positions	Horizontal contiguous surfaces
(1)	$(x_2 - x_1 + 1)(y_2 - y_1 + 1)$
(2)	wh
(3)	$w(y + h - y_1)$
(4)	$(x + w - x_1)h$
(5)	$w(y_2 - y + 1)$
(6)	$(x_2 - x + 1)h$
(7)	$(x_2 - x_1 + 1)(y_2 - y + 1)$
(8)	$(x_2 - x + 1)(y_2 - y_1 + 1)$
(9)	$(x_2 - x_1 + 1)(y + h - y_1)$
(10)	$(x + w - x_1)(y_2 - y_1 + 1)$
(11)	$(x + w - x_1)(y + h - y_1)$
(12)	$(x + w - x_1)(y_2 - y + 1)$
(13)	$(x_2 - x + 1)(y_2 - y + 1)$
(14)	$(x_2 - x + 1)(y + h - y_1)$

In a simple example depicted in Figure 2(a), the horizontal contiguous surfaces with a previous scheduled task (PST) A4 and with a next scheduled task (NST) A3 in the figure give this heuristic an awareness on avoiding "blocking-effect"; while the other surfaces A1 and A2 (vertical contiguous surfaces) give this heuristic to better pack tasks in time and space. As a result, the proposed algorithm has a full 3D-view of the positions of all scheduled and placed tasks.

Intuitively, a higher 3DTCS value will result in more compaction both in space and time. This 3DTCS heuristic gives our proposed 3D compaction algorithm with blocking-aware ability to pack tasks better as it has a more complete view of all dimensions.

Figure 2(b)(1)-(14) and Table 1 show all the placement positions and their corresponding computations of horizontal contiguous surfaces. The arriving task (AT), with width w and height h, has a bottom-left coordinate (x, y) as shown in



(a) (b)

Fig. 3. Vertical contiguous surfaces with scheduled tasks (a) and the FPGA boundary (b)

Table 2. Computations of vertical contiguous surfaces with scheduled tasks for positions in Figure 3(a)(1)-(16)

Positions	Vertical contiguous surfaces with scheduled tasks
(1),(3)	$w.min(lt,(t_f - t_s))$
(2),(4)	$h.min(lt,(t_f - t_s))$
(5),(7)	$(x_2 - x_1 + 1).min(lt,(t_f - t_s))$
(6),(8)	$(y_2 - y_1 + 1).min(lt,(t_f - t_s))$
(9),(14)	$(x_2 - x + 1).min(lt,(t_f - t_s))$
(10),(13)	$(x + w - x_1).min(lt,(t_f - t_s))$
(11),(15)	$(y_2 - y + 1).min(lt,(t_f - t_s))$
(12),(16)	$(y + h - y_1).min(lt,(t_f - t_s))$

Table 3. Computations of vertical contiguous surfaces with the FPGA boundary for positions in Figure 3(b)(1)-(8)

Positions	Vertical contiguous surfaces with the FPGA boundary
(1)-(4)	$(w + h)lt$
(5),(7)	$h.lt$
(6),(8)	$w.lt$

Figure 2(b)(15). The arriving task can be contacted with the previous scheduled task (PST) and (or) the next scheduled task (NST) to produce the horizontal contiguous surfaces. The scheduled task has a bottom-left coordinate (x_1, y_1) and a top-right coordinate (x_2, y_2) as illustrated in Figure 2(b)(16).

The arriving task can be contacted with scheduled tasks and (or) FPGA boundary to produce the vertical contiguous surfaces. All placement positions of the arriving task (AT) and their corresponding computations of vertical contiguous surfaces with the scheduled task (ST) are shown in Figure 3(a) and Table 2. The arriving task with a life-time lt is started execution at time t_s; the finishing

time of scheduled task is denoted as t_f. Computations of vertical contiguous surfaces between the arriving task with the FPGA boundary are illustrated in Figure 3(b) and formulated in Table 3.

6 The 3D Compaction (3DC) Algorithm

Figure 4 shows the pseudocode for the proposed 3D Compaction (3DC). The algorithm maintains two linked lists: the execution list and the reservation list. The execution list saves the information of all currently running tasks sorted in order of increasing finishing times; the reservation list contains the information of all scheduled tasks sorted in order of increasing starting times. The information stored in the lists are the bottom-left coordinate (x_1, y_1), the top-right coordinate (x_2, y_2), the starting time t_s, the finishing time t_f, the task name, the next pointer, and the previous pointer.

In lines 1-13, the algorithm computes the starting time matrix (STM) with respect to the arriving task area wh on the device area WH. The algorithm collects all possible positions that have enough space for the arriving task by scanning the executing and reservation lists. The algorithm fills each element

```
1. for (y=1;y<=H-h+1;y++)
{
    2. for (x=1;x<=W-w+1;x++)
    {
        3. STM(x,y)=a
    }
}

4. for all tasks in execution list
{
    5. for (y=max(1,y1-h+1);y<=min(y2,H-h+1);y++)
    {
        6. for (x=max(1,x1-w+1);x<=min(x2,W-w+1);x++)
        {
            7. if (STM(x,y) < tf)
            {
                8. STM(x,y)=tf
            }
        }
    }
}

9. for all tasks in reservation list
{
    10. for (y=max(1,y1-h+1);y<=min(y2,H-h+1);y++)
    {
        11. for (x=max(1,x1-w+1);x<=min(x2,W-w+1);x++)
        {
            12. if ((STM(x,y) < tf) AND (STM(x,y)+lt>ts))
            {
                13. STM(x,y)=tf
            }
        }
    }
}
```

```
14. collect all positions from STM that have the earliest starting time
15. for all above positions
{
    16. c_3DTCS=compute 3D contact surfaces
    17. c_SFTD=compute sum of finishing time difference
    18. if (c_3DTCS>3DTCS_max AND c_SFTD<SFTD_min)
    {
        19. best_position=current position
        20. 3DTCS_max=c_3DTCS
        21. SFTD_min=c_SFTD
    }
    22. else if (c_3DTCS>3DTCS_max)
    {
        23. best_position=current position
        24. 3DTCS_max=c_3DTCS
    }
    25. else if (c_3DTCS=3DTCS_max AND c_SFTD<SFTD_min)
    {
        26. best_position=current position
        27. SFTD_min=c_SFTD
    }
}
28. if best_starting_time=arrival time
{
    29. add task to the execution list
}
else
{
    30. add task to the reservation list
}
31. if  the reservation list is not empty
{
    32. update reservation list
}
33. if  the execution list is not empty
{
    34. update execution list
}
```

Fig. 4. Pseudocode of 3D Compaction algorithm

of the STM with the arrival time of incoming task a (lines 1-3). The algorithm updates groups of elements that are affected by all executing tasks in execution list (lines 4-8) and by all scheduled tasks in reservation list (lines 9-13).

In line 14, the algorithm collects all best positions (candidates) that have the earliest starting time (best starting time positions: best positions in terms of starting time) from the STM.

Since the algorithm not only wants to get the best position in terms of starting time (time domain) but the best position in terms of space (space domain) as well. To pack compactly tasks, we propose to use the 3DTCS heuristic as presented earlier in Section 5. The algorithm computes the 3DTCS (line 16) using formulas from Table 1 to Table 3 and chooses the best position from all the best starting time positions. Hence, the algorithm does not need to compute the 3DTCS for all positions; it only computes the 3DTCS for the best positions (candidates) (line 15). Intuitively, the highest 3DTCS value gives the best position in terms of packing to avoid "blocking-effect".

Besides the 3DTCS heuristic, the algorithm also uses the sum of finishing time difference (SFTD) heuristic for all scheduled tasks that vertically contacted with the arriving task (referred as a VC set). The algorithm computes current SFTD ($c\_SFTD = \sum_{\forall tasks \in VC} |t_s + lt - t_f|$) in line 17. The SFTD heuristic gives our algorithm an ability to group tasks with similar finishing times to get large free space during deallocations.

The algorithm chooses the position with the highest 3DTCS value and the lowest SFTD value for allocating the arriving task (lines 18-27). Allocating the arriving tasks at the highest 3DTCS compacts the tasks both in time and space; while grouping tasks with similar finishing times creates more possibility to produce larger free space during deallocations.

The algorithm allocates the incoming task when there is available space for the task at its arrival time; otherwise, the algorithm needs to schedule the task for future execution. If the arriving task can be allocated at its arrival time (line 28), it will be executed immediately and added in the execution list (line 29); otherwise, it is inserted in the reservation list (line 30).

When the tasks in the reservation list are executed, they are removed from the reservation list and added in the execution list. The finished tasks in the execution list are deleted after execution. These updating processes are executed when the lists are not empty (lines 31-34).

The time complexity analysis of our 3DC is presented in Table 4. In which W, H, N_{ET}, N_{RT} are the FPGA width, the FPGA height, the number of executing tasks in the execution list, the number of reserved tasks in the reservation list, respectively.

The main difference between our algorithm and existing algorithms is the presence of the 3D compaction ability. Because of this 3D compaction ability, our algorithm can avoid "blocking-effect". In contrast, the existing algorithms do not have the blocking-awareness. Some existing algorithms only have the 2D compaction ability; instead, our algorithm has the 3D compaction ability to

Table 4. Time complexity analysis of 3D Compaction algorithm

Lines	Time Complexity
1-3	$O(WH)$
4-8	$O(WHN_{ET})$
9-13	$O(WHN_{RT})$
14	$O(WH)$
15-27	$O(WH)$
28-30	$O(max(N_{ET}, N_{RT}))$
31-32	$O(N_{RT})$
33-34	$O(N_{ET})$
Total	$O(WHmax(N_{ET}, N_{RT}))$

compact tasks both in time and space domains. Besides, the algorithm also has an ability to group tasks with similar finishing times to achieve larger free space during deallocations. In the CR, every element of their EA matrix is checked to know if it falls into the coverage rectangles of execution and scheduling tasks for updating as shown in [6]. In contrast, our algorithm updates the STM matrix in groups of elements affected by all executing (lines 5-6) and scheduled tasks (lines 10-11); the algorithm does not need to check each element for updating. As a result, our algorithm computes starting times faster than the CR. Moreover, our 3DC does not need to compute boundary values for all reconfigurable units of its free space in the periphery. As a consequence, our algorithm has less runtime overhead compared to the CR as will be presented later.

7 Evaluation

7.1 Evaluation in Terms of Scheduling and Placement Quality

Evaluation with Synthetic Workloads. We have built a discrete-time simulation framework in C to evaluate the proposed algorithm. The framework was compiled and run under Linux operating system on a Pentium-IV 3.4 GHz PC. To better evaluate the algorithm with synthetic workloads, (1) we modeled realistic random hardware tasks to be executed on a realistic target device; (2) we evaluated the algorithm not only in terms of scheduling and placement quality but also in terms of runtime overhead.

To model realistically the synthetic hardware tasks, we use a benchmark set (e.g. MDCT, matrix multiplication, hamming code, sorting, FIR, ADPCM, etc) from [10] and then use the DWARV [9] C-to-VHDL compiler to translate the benchmarks to VHDL. The VHDL code is then synthesized using the Xilinx ISE 8.2.01i_PR_5 tools to obtain the information of hardware task size range as a reference for our random task set generator. The task widths and heights are randomly generated in the range [7..45] reconfigurable units to model hardware tasks between 49 and 2025 reconfigurable units to mimic the results of synthesized hardware units. Every task set consists of 1000 tasks, each of which has a life-time and task size. The life-times are randomly generated in [5..100] time units, while the intertask-arrival periods are randomly chosen between one time

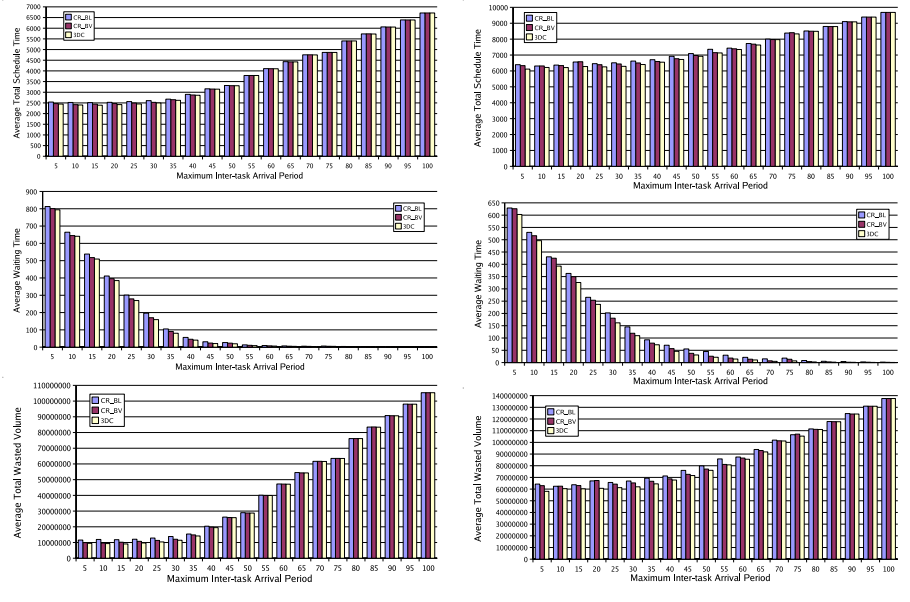
unit and a specified maximum intertask-arrival period. Total tasks per arrival are randomly generated in [1..15]. Since the algorithms are online, the information of arriving tasks is unknown until their arrival times. We model a realistic FPGA with 116 columns and 192 rows of reconfigurable units (Virtex-4 XC4VLX200).

Our 3DC is designed for 2D area models. Therefore for fair comparison, we only compare our algorithm with algorithms that support 2D area models. Since the RPR [4], the Classified Stuffing [2], the Intelligent Stuffing [3] were designed only for 1D area models as shown in Section 3, we do not compare them with our 3DC.

Since the Stuffing outperforms the Horizon as presented in [1], we do not compare our algorithm to the Horizon. In [6], the CR outperforms the original 2D Stuffing [1] and the 2D Window-based Stuffing [5]; therefore, we only compare our algorithm to the CR.

To evaluate the 3DC, we have implemented three different algorithms: the CR [6] using BL (Bottom-Left) scheme (CR_BL), the CR [6] using BV (Boundary Value) scheme [7] (CR_BV), and our 3DC. The evaluation is based on three performance parameters as defined before in Section 2.

The CR does not have a blocking-awareness. Instead, our algorithm uses a 3D compaction for avoiding "blocking-effect". As a consequence, our algorithm has a better quality than the CR. The 3DC has up to 4.8 % less schedule time, 38.4 % less waiting time, and 22.9 % less wasted volume compared to the CR as shown in Figure 5(a).



(a) (b)

Fig. 5. Evaluation with synthetic (a) and real workloads (b)

The system idle time increases when the maximum inter-task arrival period increases; as a result, the average total schedule time and the average wasted volume increase.

The system is busier when the maximum inter-task arrival period decreases; tasks arrive more frequently to the system. Hence, it is more difficult to schedule tasks. Consequently, the average waiting time increases.

Evaluation with Real Workloads. To evaluate the 3DC with real workloads, realistic hardware tasks from [11] were used. In the simulation, we assume that the life-time lt_i is the sum of reconfiguration time rt_i and execution time et_i. The experimental results with real workloads are presented in Figure 5(b).

Figure 5(b) shows that the superiority of our algorithm is not only applicable for synthetic tasks but also for real tasks. Evaluation with real tasks shows that our algorithm has up to 4.6 % less schedule time, 75.1 % less waiting time, and 9.9 % less wasted volume compared to the CR.

7.2 Evaluation in Terms of Algorithm Execution Time

To complete the evaluation, we also study the algorithm execution time since the execution time of online task scheduling and placement is considered as an overhead for the overall execution time of the applications. To show the effect of total number of scheduled and running tasks as well as FPGA area, we do simulation by changing these parameters as presented in Figure 6.

Figure 6 shows that our 3DC runs up to 133 times faster than the CR. The speed up will be higher for more scheduled and running tasks as well as for larger FPGA fabrics. Since the CR uses the boundary value heuristic for searching placement, the CR needs to compute boundary values for all reconfigurable units of its free space in the periphery. In contrast, our 3DC computes the 3DTCS only in one step as presented in Section 5. Moreover, the updating is done per each element of the matrix in the CR; each element is needed to be checked with all executing tasks and scheduled tasks. In contrast, our algorithm updates the matrix in groups of elements located by all executing tasks and scheduled tasks; the algorithm does not need to check each element for updating. As a result, our

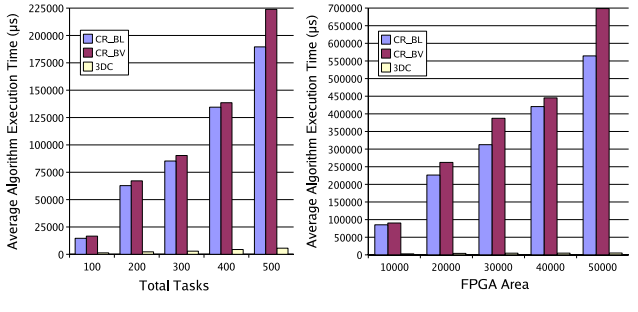


Fig. 6. Evaluation algorithm execution time

3DC has less runtime overhead than the CR by avoiding the CR's long boundary value computation and speeding up the starting times computation.

More FPGA area creates additional area suitable for the arriving task (more free volume) and more total number of scheduled and running tasks forces algorithms to check more tasks; as a result, the algorithms need more time to compute the matrix for finding starting time (all algorithms), all boundary values for all more candidates (CR algorithm) and all 3DTCS for all more candidates (3DC algorithm). Because of its long boundary value and matrix computations, the CR execution time increases faster than our 3DC.

8 Conclusions

To avoid "blocking-effect" of existing algorithms, we have proposed a new 3DTCS heuristic and used it to build a novel blocking-aware algorithm, 3D Compaction (3DC). Because of its 3D compaction, the 3DC places and schedules tasks more compactly. Moreover, the 3DC is equipped with an ability to group similar finishing time tasks to form larger free area for better allocating future tasks. Since the previous algorithm uses the boundary value heuristic for searching suitable placement, it needs to compute the values for all reconfigurable units of its free space in the periphery. In contrast, our 3DC computes the 3DTCS only in one step. In addition, the updating is done per each element of the matrix for finding starting time in the previous algorithm; each element is checked with all executing and scheduled tasks. Our 3DC updates the matrix in groups of elements located by all executing and scheduled tasks. The experimental results show that the 3DC not only has better scheduling and placement quality but also has lower runtime overhead compared to existing algorithms.

A possible direction for future research is to equip the algorithm with an ability to run tasks at different clock speeds or voltages for power saving (power-aware) and to place tasks based on required I/O positions (I/O-aware).

Acknowledgment

This work is sponsored by the hArtes project (IST-035143) supported by the Sixth Framework Programme of the European Community under the thematic area "Embedded Systems".

References

1. Steiger, C., Walder, H., Platzner, M.: Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices. In: Y. K. Cheung, P., Constantinides, G.A. (eds.) FPL 2003. LNCS, vol. 2778, pp. 575–584. Springer, Heidelberg (2003)
2. Chen, Y., Hsiung, P.: Hardware Task Scheduling and Placement in Operating Systems for Dynamically Reconfigurable SoC. In: EUC, pp. 489–498 (2005)
3. Marconi, T., Lu, Y., Bertels, K.L.M., Gaydadjiev, G.N.: Online Hardware Task Scheduling and Placement Algorithm on Partially Reconfigurable Devices. In: Woods, R., Compton, K., Bouganis, C., Diniz, P.C. (eds.) ARC 2008. LNCS, vol. 4943, pp. 306–311. Springer, Heidelberg (2008)

4. Lu, Y., Marconi, T., Bertels, K.L.M., Gaydadjiev, G.N.: Online Task Scheduling for the FPGA-Based Partially Reconfigurable Systems. In: ARC, pp. 216–230 (2009)
5. Zhou, X., Wang, Y., Huang, X., Peng, C.: On-line Scheduling of Real-time Tasks for Reconfigurable Computing System. In: FPT, pp. 57–64 (2006)
6. Zhou, X., Wang, Y., Huang, X., Peng, C.: Fast On-line Task Placement and Scheduling on Reconfigurable Devices. In: FPL, pp. 132–138 (2007)
7. Sharma, D.D., Pradhan, D.K.: A Fast and Efficient Strategy for Submesh Allocation in Mesh-Connected Parallel Computers. In: IPDPS, pp. 682–689 (1993)
8. Xilinx: Virtex-4 FPGA Configuration User Guide. Xilinx user guide UG071 (2008)
9. Yankova, Y.D., Bertels, K.L.M., Vassiliadis, S., Meeuws, R.J., Virginia, A.: Automated hdl generation: Comparative evaluation. In: ISCAS, pp. 2750–2753 (2007)
10. Meeuws, R.J., Yankova, Y.D., Bertels, K.L.M., Gaydadjiev, G.N., Vassiliadis, S.: A Quantitative Prediction Model for Hardware/Software Partitioning. In: FPL, pp. 735–739 (2007)
11. Marconi, T., Lu, Y., Bertels, K.L.M., Gaydadjiev, G.N.: A Novel Fast Online Placement Algorithm on 2D Partially Reconfigurable Devices. In: FPT, pp. 296–299 (2009)

TROUTE: A Reconfigurability-Aware FPGA Router

Karel Bruneel and Dirk Stroobandt

Hardware and Embedded Systems Group, ELIS Dept., Ghent University,
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{karel.bruneel,dirk.stroobandt}@UGent.be

Abstract. Since FPGAs are inherently reconfigurable, making FPGA designs generic does not reduce chip cost, as is the case for ASICs. However, designing and mapping lots of specialized FPGA designs introduces an extra EDA cost. We describe a two staged fully automatic FPGA tool flow that efficiently maps a generic HDL design to multiple specialized FPGA configurations. The mapping is fast enough to be executed online in dynamically reconfigurable systems. In this paper we focus on TROUTE, the routing algorithm used in our tool flow. We used TROUTE to implement reconfigurable Multistage Interconnection Networks and show huge improvements in area, speed and mapping time compared to conventional non-reconfigurable implementations.

1 Introduction

FPGA design differs significantly from ASIC design in the generality of the design solution. Indeed, ASIC designers need to amortize the NRE (non-recurring engineering) cost over a large volume of chip instances. This can be done by making the design more generic, so that it meets the needs of as many customers as possible. Besides the regular input data, a generic design takes *Parameter Data* as input, specifying how the regular input data should be processed. The configuration data can differ for each customer or group of customers and can even change over time. Naturally, more generic ASIC designs will be larger and possibly have somewhat lower performance, but the gains of selling more chip instances will in many cases outweigh these disadvantages.

FPGAs, on the other hand, are fully reconfigurable and therefore can be reused for any function of similar size and complexity. It is thus not useful to make an FPGA design as generic as possible because this will not make the chip any cheaper. On the contrary, you may need to switch to a more expensive FPGA to meet the area and performance cost of the generality. However, making lots of specialized designs now introduces an extra EDA cost as each specialized design must be designed separately and mapped to the FPGA. This is no longer feasible when we wish to switch between different designs at run time, in contrast to the generic ASIC solution.

Let us for example take the case of a communication network where each node has its own 128-bit encryption key. A generic ASIC design would store the encryption key in an internal register. Each node can then be configured for a specific

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 207–218, 2010.

key by writing that register. In an FPGA we could use the same technique, but we could also save area and boost performance of the nodes by generating specialized FPGA bitstreams for each node. However, there are 2^{128} possible keys, making it infeasible to run the FPGA tool chain for each possible key. Also specializing the new configuration at run time for each key that is selected, is infeasible due to the amount of time it takes to synthesize such a new configuration.

To solve this problem we propose a two stage tool flow that drastically decreases the cost of generating a specialized FPGA configuration. The first stage of the tool flow takes a *Parameterizable HDL Design* as input and generates a *Parameterizable FPGA Configuration*. A parameterizable HDL design has two types of inputs: regular inputs and parameter inputs. The latter will not be inputs to the final design, but will be bounded to a constant value in the second stage (thus distinguishing between the various specialized configurations). In our encryption example, the key will be a parameter input. A parameterizable configuration is a set of Boolean functions that generates FPGA configurations given a parameter value. The second stage, generates specialized configurations by evaluating the parameterizable configuration for a parameter value. This can be repeated for multiple parameter values. One can easily see that for large numbers of parameter values the average cost per configuration is approximately the cost of running the second stage because the cost of generating the parameterizable configuration is amortized over all specialized configurations.

In [2,3] we have shown that it is possible to build a two staged tool flow of which the evaluation of the parameterizable configuration runs 5 orders of magnitude faster than a conventional FPGA tool flow without sacrificing too much area and performance compared to a fully specialized FPGA design. In this tool flow only the LUT truth table bits of the configuration are expressed as Boolean functions of the parameters, whereas the routing is fixed for all configurations. In this paper, we present a tool flow that also expresses the routing configuration bits as a function of the parameter inputs. In the experiments section we show that this can result in a better area utilization and performance. Expressing the routing bits as Boolean functions of the parameters requires changes in all stages of the conventional FPGA tool flow (technology mapping, placement and routing). In this paper, we only discuss the changes in the router in detail. The other steps will be addressed only briefly.

2 Staged Mapping Tool Flow

Fig. 1 gives an overview of our mapping tool flow. The tool flow uses a compiler technique called staged compilation, or staged mapping, as we call it in the case of FPGA mapping. In our staged mapping flow the final result, a *Specialized FPGA configuration*, is generated in two steps or stages: the *Generic Stage* and the *Specialization Stage*. In contrast to conventional mapping the design specification is not entirely introduced at the start of the mapping process but a part of this design specification is introduced at each stage. A Parameterizable HDL Design is introduced to the generic stage and the parameter values are introduced to the

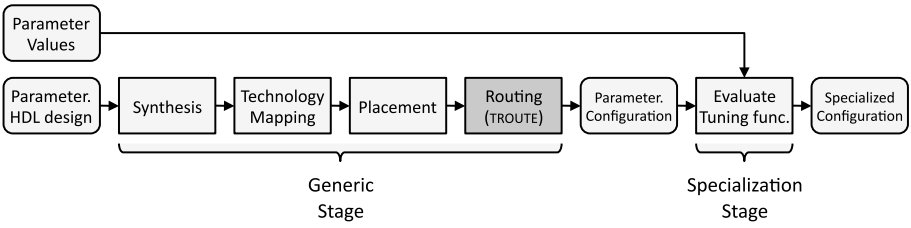


Fig. 1. Overview of our staged mapping tool flow

specialization stage. Each stage processes the result of the previous stage and the extra specification part to form a new intermediate result that will be introduced to the next stage. The generic stage produces a parameterizable configuration and the specialization stage combines this with the parameter values to produce the specialized configuration.

A parameterizable configuration is a function that takes parameter values as arguments and produces a specialized configuration[1]. We represent a parameterizable configuration as a vector of Boolean functions whose elements are associated with the bits in the FPGA's configuration memory. The Boolean functions are called *Tuning Functions*. They are of closed form and have a single output.

The steps needed in the generic stage are similar to those used in conventional FPGA mapping: synthesis, technology mapping, place and route. In Section 3 we will explain these algorithms in more detail. It is important to note here that these algorithms are computationally hard and thus need a large run-time. The specialization stage generates a regular FPGA configuration by evaluation of the parameterizable configuration. This involves evaluating a set of closed form Boolean functions. Hence, the run-time of the second stage is linear in the size of the parameterizable configurations. The specialization stage will thus run a lot faster than the generic stage [2]. Therefore, the staged mapping tool flow is more efficient in generating specialized configurations than a conventional tool flow. This is because our staged flow can reuse the parameterizable configuration for each parameter value. The effort spend in the generic stage thus is divide over all invocations of the specialization stage. For large sets of parameter values the average mapping effort is approximately the effort pent in the specialization stage.

3 Overview of the Generic Stage

The problem faced by the generic stage of our tool flow is to produce a parameterizable configuration given a parameterizable HDL description while optimizing some cost function. For the sake of clarity, we concentrate on minimizing the area used by the parameterizable configuration, but the techniques can be extended for other optimization criteria such as speed or a combination of area

[1] The concept of parameterizable configurations can easily be extended to parameterizable partial configurations, which are functions that produce partial configurations when given the parameter values as argument.

and speed. Without loss of generality, we will also assume a very simple island style target architecture with 4-input LUTs and wires of length 1.

Since both the input and output of the tool flow are parameterizable the internal data structures need to be able to express parameterizability and the algorithms that transform the data structures need to preserve the parameterizability. Similar to conventional FPGA mapping we divide the mapping problem into four subproblems: synthesis, technology mapping, placement and routing. In what follows we give an overview of these algorithms and the data structures used in our tool flow.

3.1 Synthesis

The synthesis step converts the parameterizable HDL description into a gate-level circuit. As we described in Section 2, a parameterizable HDL description distinguishes regular inputs from parameter inputs so, this distinction has to be preserved in the gate-level circuits. This can easily be done by allowing both types of inputs in the gate-level circuit data structure. The synthesis tool simply has to pass the information about the inputs.

3.2 Technology Mapping

During technology mapping the gate-level circuit produced by the synthesis step is mapped on the resources available in the target FPGA architecture while trying to optimize the area of the implementation.

The result of a conventional technology mapper is a mapped circuit containing two types of functional blocks: LUTs and nets. A LUT can be implemented by a physical LUT on the FPGA and a net can be implemented by a subset of the routing switches in the configurable interconnect.

Because the bits in the parameterizable configuration are Boolean functions of the parameter inputs, both the truth table bits as well as the routing bits can change. First, a LUT with a truth table that is function of the parameters is called a *Tunable LUT (TLUT)* [2]. It's easy to see that a TLUT is a generalization of a regular LUT. Second, the way physical LUTs are connected can change depending on the parameters. We thus need functional blocks that reflect the parameterizability of interconnections. We call these blocks *Tunable Connections (TCONs)*. A circuit containing TLUTs (instead of regular LUTs) and TCONs (instead of nets) is called a *Tunable Circuit*.

A TCON has any number of input ports $I = \{i_0, i_1, \ldots, i_{L-1}\}$ and any number of output ports $O = \{o_0, o_1, \ldots, o_{M-1}\}$. Every TCON is associated to a connection function f_{con} that shows how the output ports are connected to the input ports given a parameter value[2] $\boldsymbol{P} = (p_0, p_1, \ldots, p_{N-1}) \in \{0,1\}^N$, see equation (1). Just like a TLUT is a generalization of a regular LUT, it's easy to see that a TCON is a generalization of a net.

$$\begin{aligned} f_{con} : O \times \{0,1\}^N &\rightarrow I \\ (o, \boldsymbol{P}) &\mapsto i \end{aligned} \tag{1}$$

[2] Without loss of generality, we combine all parameters into one parameter vector $\boldsymbol{P}$.

In what follows we will use a TCON with the functionality of a four-way switch, as example. This TCON has two inputs $\{i_0, i_1\}$ and two outputs $\{o_0, o_1\}$. The 1-bit parameter p controls how the inputs are connected to the outputs. When $p = 0$, o_0 is connected to i_0 and o_1 is connected to i_1. When $p = 1$, o_0 is connected to i_1 and o_1 is connected to i_0.

In this paper, we focus on the routing step of the tool flow. Therefore we assume the tunable circuit is given. More information on mapping to TLUTs can be found in [2,3]. To our knowledge there are no technology mappers available that can map to a combination of TLUTs and TCONs.

3.3 Placement

During placement, each of the TLUTs in the tunable circuit is associated to (placed on) one of the physical LUTs of the FPGA while optimizing for a certain property, e.g. the routability of the placement.

Many FPGA placers use simulated annealing to place the mapped circuit. The cost function of a routability-driven placer is an estimate of the total number of wires the router will need to route the design given the current placement. This is calculated as the sum of the estimated number of wires used by the individual nets [1]. This same scheme is used to build a placer for tunable circuits. The only difference lies in the way we estimate the number of wires used by the router to route a TCON. In this paper we concentrate on the routing algorithm and therefore we assume the placement as given.

3.4 Routing

Conventional routers calculate the Boolean values that need to be stored in the configuration bits of the configurable interconnection network so that the physical LUTs are connected as is specified by the nets in the mapped circuit.

Our router is more complicated as it needs to calculate Boolean functions for the configuration bits. On one hand the parameterizable configuration evaluates to a specialized configuration given a parameter value and on the other hand a tunable circuit simplifies to a regular LUT circuit for that same parameter value. Our router will thus calculate Boolean functions for the routing bits so that for any parameter value the specialized configuration implements the connections specified by the regular LUT circuit.

In Section 4 we give a detailed description of an algorithm, called TROUTE that solves this problem.

4 TROUTE

In this section we describe the algorithm TROUTE. Given a placed tunable circuit, it produces Boolean functions for the routing bits of the target FPGA so that the physical LUTs are connected as is specified by the TCONs of the tunable circuit. TROUTE is based on the widely used PATHFINDER algorithm [1,7].

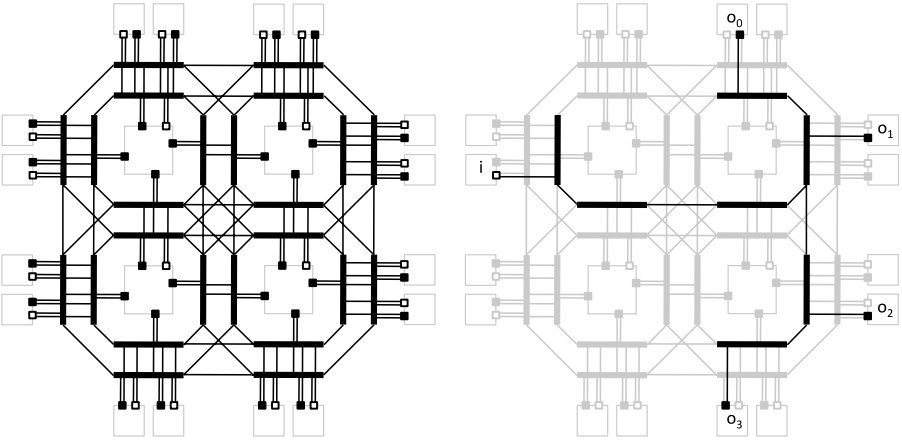


Fig. 2. (a) Resource graph for a simple 2×2 island style FPGA. Wires are solid black lines; Edges are thin lines; Sources are open boxes; And sinks are filled boxes. (b) Routing tree of a net $(i, \{o_0, o_1, o_2, o_3\})$.

4.1 The Resource Graph

Both PATHFINDER and TROUTE uses a directed graph, called the *Resource Graph*, to represent the routing architecture of an FPGA. Because this graph can be constructed for many routing architectures, the algorithms are very flexible.

The resource graph is a directed graph $C = (V, E)$, where the vertices V represent the routing resources (the wires and the ports of the logic blocks). A directed edge (t, h) represents the possibility of routing a signal from resource t (the tail of the edge) to resource h (the head of the edge), by setting a switch. There are two types of port vertices: sources and sinks. Sources represent output ports of logic blocks while sinks represent input ports of logic blocks.

We can construct a resource graph as follows. Create a vertex v_r for each routing resource r (wire or port) of the target FPGA. For each unidirectional switch that, when closed, forces the logic value of resources i on resource o, create a directed edge (v_i, v_o), and for each bidirectional switch that connects resource r to resource s, create two directed edges (r, s) and (s, r). There are many extensions possible to this model [5] (beyond the scope of this paper).

Fig. 2 depicts the resource graph of a simple 2×2 island style FPGA with only length 1 wires and bidirectional switches. The wires are represented by solid black lines, the sinks and sources by small squares. The sinks are filled and the sources are not. For the sake of clarity we have not drawn all edges. The thin lines each represent two edges, one for each sense.

4.2 TCONs, Patterns and Nets

A TCON simplifies to a set of nets for a specific parameter value. We call this set of nets a *Connection Pattern* of the TCON. Each connection pattern describes

one way to connect the output ports to the input ports of a TCON. A TCON can thus be represented as a set of connection patterns and a connection pattern as a set of nets.

When the placement of the LUTs is known the source vertex and the sink vertices associated to respectively the input port and output ports of the nets in the mapped circuit are known. Each net ν in the LUT circuit can thus be associated with an ordered pair (so_ν, SI_ν) containing a source vertex so_ν and a set of sinks vertices SI_ν. Note that due to the definition of a TCON (Section 3.2), the sink set of the nets in any pattern are disjoint.

A routing tree RT_ν for net ν is a rooted tree embedded in the resource graph C that has the source vertex as root and the sink vertices as its leaves. It does not contain any other source or sink vertices. This routing tree contains paths from the source vertex to each sink vertex of the net ν. Fig. 2 shows the routing tree of a net $(i, \{o_0, o_1, o_2, o_3\})$. Once the routing tree RT_ν of a net is found, setting the FPGA's routing bits so all connections represented by the net ν are realized is easy. We just have to set the configuration bits so that the switches associated to the edges in RT_ν are closed, and the switches associated to the edges that only end or start in RT_ν are open.

Analog to the routing tree of a net, a routing graph RG_τ of a TCON τ is a subgraph embedded in the resource graph C. By controlling the switches associated to the edges in RG_τ it should be possible to realize all connections specified by the TCON. Therefore, RG_τ should contain a routing tree $RT_{\pi,\nu}$ for each net ν of each connection pattern π. Since the nets in a pattern coincide, their routing trees have to be disjoint. Nets that are part of different patterns, however, don't coincide. Their routing trees can therefore overlap. We define the routing graph RG_π of a pattern π as the union the routing trees $RT_{\pi,\nu}$.

Routing a tunable circuit thus simplifies to finding a set of disjoint routing graphs, one for each of the TCONs in the tunable circuit.

4.3 Tuning Functions

Every connection pattern π can be associated to a Boolean function of the parameters, called the *Pattern Condition* $f^\pi_{cond}(\boldsymbol{P})$. This pattern condition is true for all parameter values that simplify the TCON to pattern π. Note that a TCON can simplify to the same pattern for several parameter values. Since every net is part of one pattern, a net is also associated to a pattern condition.

Once the routing tree $RT_{\pi,\nu}$ for each of the nets in the TCON routing graph RG_τ is found, the condition for a switch to close is given by the logical OR of all pattern conditions of those nets whose routing trees contain an edge associated to the switch. The tuning function for the configuration bit that controls the switch is equal to this condition or its inverse if the control of the switch is active high or active low respectively.

A possible routing of our 4-way switch example is shown in Fig. 3. The figure on the left shows the routing trees of the two nets in pattern π_0 with condition $f^{\pi_0}_{cond}(\boldsymbol{P}) = p_0$ and the figure on the right shows the routing trees of the two nets in pattern π_1 with condition $f^{\pi_1}_{cond}(\boldsymbol{P}) = \overline{p_0}$. The routing graph of the TCON is

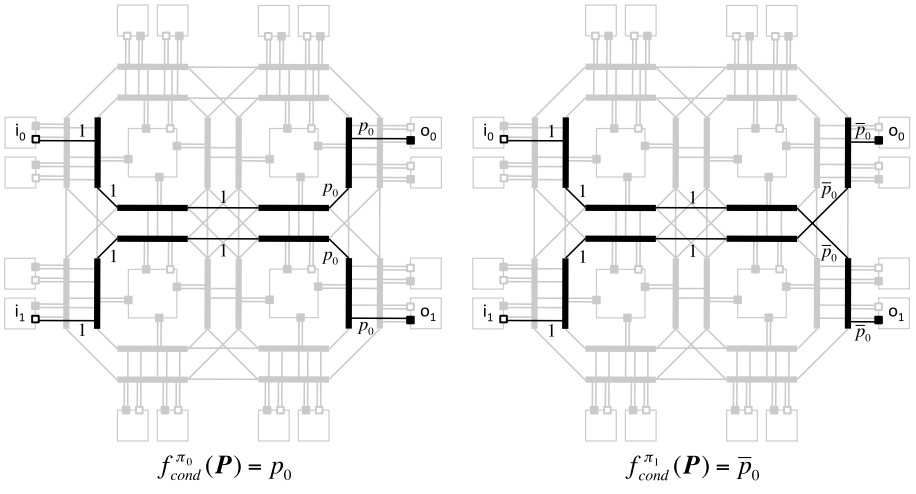


Fig. 3. Routing of the two connection patterns of a 4-way switch, their pattern conditions and the tuning functions

the union of the routing trees. The edges that are crucial to the routing of the TCON are annotated with their tuning function (Fig. 3) assuming the switches are controlled active high.

4.4 The Algorithm

The only problem left is finding a set of disjoint routing graphs, one for each of the TCONs in the tunable circuit.

First, we describe a heuristic subroutine that searches a minimum cost routing graph for a given TCON. Each vertex v in the resource graph has an associated cost c_v. The cost of a routing graph is the sum of the costs of its vertices. Second, we explain how to use this subroutine to find a set of disjoint routing graphs given a set of TCONs.

The TCON Router. We will route a TCON by calculating a routing tree for each of the nets in the TCON. The union of all these routing trees is the routing graph of the TCON. We know that nets in a routing pattern coincide and thus have to be disjoint. However, two nets that are part of different patterns, never coincide and can thus share routing resources. We use this last property to minimize the routing cost of a TCON by maximizing the overlap among patterns.

The pseudo code of our proposed heuristic algorithm is shown in Fig. 5. The algorithm contains two nested for loops. The outer loop loops over all patterns of the TCON. The inner loop loops over all nets in the current pattern and routes them using a net router. A net router is a heuristic that searches a minimum cost routing tree for a given net. We use the net router described in [7].

```
while shared resources exist:
  for each tcon τ do:
    τ.ripUpRouting()
    RG_τ = routeTcon(τ)
    for each vertex v in RG_τ:
      v.updateSharingCost()
  for each vertex v in C do
    v.updateHistoryCost()
```

Fig. 4. Main loop (Negotiated Congestion) of the TROUTE algorithm

```
function routeTcon(tcon τ)
  RG_τ = null graph
  for each pattern π in tcon τ:
    RG_π = null graph
    for each net ν in pattern π:
      RT_ν = routeNet(net)
      for each vertex v in RT_ν:
        v.inPattern = true
        v.inTcon    = true
      RG_π = RG_π ∪ RT_ν
    for each vertex v in RG_π:
      v.inPattern = false
    RG_τ = RG_τ ∪ RG_π
  for each vertex v in RG_τ:
    v.inTcon = false
  return RG_τ
```

Fig. 5. Pseudo code for the TCON router

In order to forbid resources sharing for nets within one pattern and allow resource sharing for nets in different patterns we manipulate the cost of the vertices within the TCON router. Therefore, we keep track of two extra flags for each vertex in the resource graph: *inTcon* and *inPattern*. The *inTcon* flag marks those resources that are used by already routed patterns of the TCON. The *inPattern* flag marks those resources that are used by already routed nets in the current pattern. These flags are used to calculate the manipulated cost of a resource c'_v, as is shown in equation 2.

$$c'_v = \begin{cases} \infty \text{ if } inPattern \\ 0 \quad \text{if } inTcon \wedge \overline{inPattern} \\ c_v \text{ otherwise} \end{cases} \tag{2}$$

There are three cases. The first case ensures that a resource that is already used in the current pattern cannot be used to route an other net in the current pattern. The second case stimulates resource sharing when a resource is already in use by the TCON, but not by the current pattern. It does this by making the cost of these resources equal to zero. The third case is the default case.

Negotiated Congestion. The TROUTE algorithm uses a mechanism called negotiated congestion to calculate a set of disjoint routing graphs for a given tunable circuit. The pseudo code of TROUTE is shown in Fig. 4. The algorithm iteratively rips up and reroutes (*routeTcon*) each of the TCONs until their routing graphs are disjoint. Or in other words, there are no shared resources.

In negotiated congestion, the individual routing problems are coupled by updating the vertex costs c_v during the routing process (*updateSharingCost* and *updateHistoryCost*). Our algorithm calculates and updates the vertex cost in exactly the same way as the routability-driven router described in [1].

Table 1. Properties of nine multi stage Clos network implementations. The numbers between brackets are relative compared to the *Tcon* implementation of the same size.

Impl. size	type	Area LUTs		wires		Speed logic depth		Routing $t_{route}[s]$		W_m	Architecture *cols*	*rows*	*W*
	Conv	202	(12.63)	2131	(9.19)	5	(5.00)	7.96	(18.51)	6	20	20	7
16	*Tlut*	48	(3.00)	526	(2.26)	3	(3.00)	1.06	(2.47)	4	10	10	5
	Tcon	16	(1.00)	232	(1.00)	1	(1.00)	0.43	(1.00)	5	8	8	6
	Conv	1016	(7.94)	13613	(4.56)	9	(4.50)	294.73	(29.21)	6	47	47	7
64	*Tlut*	320	(2.50)	3511	(1.17)	5	(2.50)	24.71	(2.45)	8	23	23	10
	Tcon	128	(1.00)	2987	(1.00)	2	(1.00)	10.09	(1.00)	9	18	18	11
	Conv	6760	(8.80)	97994	(5.49)	12	(4.00)	15415.51	(25.09)	9	114	114	11
256	*Tlut*	1792	(2.33)	25353	(1.42)	7	(2.33)	1234.66	(2.01)	13	53	53	16
	Tcon	768	(1.00)	17853	(1.00)	3	(1.00)	614.30	(1.00)	14	39	39	17

5 Experiments and Results

The TROUTE algorithm was implemented based on a Java version of the VPR (Versatile Place and Route) [1] routability-driven router, which we implemented. We use a simple FPGA architecture[3] with logic blocks containing one 4-LUT and one flip-flop. The wire segments in the interconnection network only span one logic block. The architecture is specified by three parameters: the number of logic element columns (*cols*), the number of logic element rows (*rows*) and the number of wires in a routing channel (W).

We validate TROUTE on Multistage Interconnect Networks that are known as Clos Networks [4]. Our Clos network uses 4×4 crossbar switches as building blocks. We use 4×4 switches because these can be efficiently implemented using four 4-input TLUTs or four TCONs. We compare three network types called: *Conv*, *Tlut* and *Tcon* each for three sizes 16×16 (3 stages), 64×64 (5 stages) and 256×256 (7 stages). *Conv* uses signals to control the crossbar switches while *Tlut* and *Tcon* use reconfiguration. *Tlut* only uses reconfiguration of LUT truth tables while *Tcon* uses both reconfiguration of LUTs and reconfiguration of routing. In *Tlut* all the switches are implemented with 4 TLUTs while in *Tcon* the switches in the even stages are implemented using TLUTs and the switches in the odd stages are implemented using TCONs.

We implemented the nine networks and measured: the number of LUTs, the number of wires, the logic depth, the routing time and the minimum channel width (W_m). Table 1 shows the results. The table also shows the parameters of the FPGA architecture. As suggested in [1], we ensure low-stress place and route by choosing the number of LUTs in the FPGA architecture 20% larger than the number of LUTs in the circuit and the number of wires per channel 20% larger than W_m, the minimum channel width.

[3] A description of this architecture is provided with the VPR tool suite in `4lut_sanitized.arch`.

The wire utilization of the implementations will be influenced by the placement of the inputs of the network. If the inputs and outputs are placed far apart more wires will be needed than when they are placed close together. To normalize this influence we connect each input and output to a LUT that is connected to no other signals. This way the placer is free to place the inputs and outputs to minimize the number of wire resources. These extra LUTs are not accounted for in the LUT count of Table 1, because they are not part of the actual Clos network.

The routing of the *Conv* and *Tlut* implementations is done with the VPR routability-driven router. Their placement is done using the VPR routability-driven placer with default settings. The routing of the *Tcon* implementations is done using TROUTE. The placement is done using an adapted version of the VPR routability-driven placer, called TPLACE (beyond the scope of this paper).

The *Tcon* networks save up to a factor 8.8 in the number of LUTs compared to the *Conv* networks and up to a factor of 3 compared to the *Tlut* networks. As a measure for the clock speed we used the number of LUTs in the longest path (logic depth). When using the *Tcon* implementation, we can reduce the logic depth with up to a factor of 5 compared to the *Conv* implementation and a factor of 3 compared to the *Tlut* implementation.

The table also shows that up to a factor 5.49 can be saved in the number of wires compared to the *Conv* networks and up to a factor 2.26 compared to the *Tlut* networks. This last result might be counterintuitive since TCONs are more complex to route than nets. However, switching from *Conv* to *Tlut* to *Tcon* decreases the number of nets/TCONs and the number of LUTs. Less nets/TCONs connecting less LUTs that can be placed closer together thus results in less wires used. Because the LUTs get placed closer together W_m goes up, but it stays far from the channel widths used in commercial FPGAs.

Table 1 also shows the routing time needed for each implementation. All these experiments are done using an Intel Core 2 processor running at 2.13 GHz with 2 GiB of memory running the Java HotSpot^{TM} 64-Bit Server VM. Using the *Tcon* networks we can save a factor of 18.51 up to 29.21 in the routing time compared to the *Conv* networks and a factor of 2.01 to 2.47 compared to the *Tlut* networks. This gain in routing time is due to the decrease in routing complexity as is explained in the previous paragraph.

6 Conclusions

In this paper we introduced a two staged FPGA tool flow that enables fast generation of FPGA configurations. The generic stage maps a parameterizable HDL design to a parameterizable configuration that expresses both the truth table bits and the routing bits as Boolean functions of the parameter inputs. We also provided a detailed description of TROUTE, the routing algorithm used in the generic stage of our tool flow.

We used TROUTE to implement reconfigurable Multistage Interconnection Networks similar to the implementations in [6,8]. Since our design is done at the abstract level of tunable circuits, while theirs is done at the architectural

level, our method greatly reduces the design effort. We have also shown that our implementations greatly improve area (LUTs: 8.80×, wires: 5.49×), logic depth (4×) and even routing time (25.09×) compared to a conventional non-reconfigurable implementation. These numbers are for a 256×256 Clos network.

References

1. Betz, V., Rose, J., Marquardt, A. (eds.): Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, Norwell (1999)
2. Bruneel, K., Stroobandt, D.: Automatic generation of run-time parameterizable configurations. In: Proceedings of the International Conference on Field Programmable Logic and Applications. Kirchhoff Institute for Physics (2008)
3. Bruneel, K., Stroobandt, D.: Reconfigurability-aware structural mapping for LUT-based FPGAs. In: 2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig). IEEE, Los Alamitos (2008)
4. Clos, C.: A study of non-blocking switching networks. The Bell System Technical Journal XXXII, 406–424 (1953)
5. Hauck, S., Dehon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann, San Francisco (2007)
6. Lysaght, P., Levi, D.: Of gates and wires. In: Parallel and Distributed Processing Symposium, International, vol. 4, p. 132a (2004)
7. McMurchie, L., Ebeling, C.: Pathfinder: A negotiation-based performance-driven router for FPGAs. In: FPGA, pp. 111–117 (1995)
8. Young, S., Alfke, P., Fewer, C., McMillan, S., Blodget, B., Levi, D.: A high i/o reconfigurable crossbar switch. In: FCCM 2003: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE Computer Society, Los Alamitos (2003)

Space and Time Sharing of Reconfigurable Hardware for Accelerated Parallel Processing*

Esam El-Araby, Vikram K. Narayana, and Tarek El-Ghazawi

NSF Center for High-Performance Reconfigurable Computing (CHREC),
The George Washington University, Washington, DC 20052, USA
esam@gwmail.gwu.edu, {vikram,tarek}@gwu.edu

Abstract. High-Performance Reconfigurable Computers (HPRCs) are parallel machines consisting of FPGAs and microprocessors, with the FPGAs used as co-processors. The execution of parallel applications on such systems has mainly followed the Single-Program Multiple-Data (SPMD) model; however, overall system resources are often underutilized because of the asymmetric distribution of the reconfigurable (co-)processors relative to the (main) processors. Furthermore, with the introduction of HPRCs containing multi/many-core technologies, underutilization of system resources becomes more obvious especially for multi-tasking and multi-user usage. To address the asymmetry problem, we propose a resource virtualization solution based on Partial Run-Time Reconfiguration (PRTR). The proposed technique allows space, time, and/or space-time sharing of the reconfigurable (co-)processors among the (main) processors and thus increasing the overall system utilization. We show the effectiveness of the proposed concepts through a stochastic execution model verified with experimental implementations on the Cray XD1 platform. The results demonstrate favorable performance as well as scalability characteristics.

Keywords: Dynamic Partial Reconfiguration, Hardware Virtualization, High Performance Computing, Reconfigurable Computing.

1 Introduction

Recent years have witnessed the introduction of stand-alone general purpose Reconfigurable Computers (RCs) as well as parallel reconfigurable supercomputers called High-Performance Reconfigurable Computers (HPRCs). Examples of such supercomputers are the SRC-7 and SRC-6 [1], the SGI Altix/RASC [2] and the Cray $XT5_h$ and Cray XD1 [3]. These systems are capable of delivering high performance as well as maintaining flexibility, due to the use of FPGAs. The FPGAs are mainly used as co-processing element(s) (CPE) to the main processing element(s) (MPE) in order to accelerate critical functions in hardware. Several efforts have proved the significant performance speedups obtained by these systems for many different applications [4]. Applications for HPRCs are mainly developed using the Single-Program

* This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. IIP-0706352.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 219–230, 2010.

Multiple-Data (SPMD) programming model, which is the most common style of parallel programming used in HPC platforms. In SPMD [5], the participating processors simultaneously execute the same program at independent points, operating on different parts of the input data. Either shared memory or message passing techniques such as MPI may be adopted in order to deploy tasks and execute them in parallel [5], [6]. However, the use of SPMD programming paradigms for HPRCs can be challenging, due to the heterogeneity of the processing elements. This is primarily due to the fact that in HPRCs, the reconfigurable processors act as co-processing element(s) (CPE) to the main host processing element(s) (MPE). In particular, when the ratio of MPEs, CPEs, and their communication channels differs from unity, SPMD programs, which generally assume a unity ratio, might underutilize some of the system processing resources, for example microprocessors [4].

In this work, we propose to space, time, and/or space-time share the reconfigurable resources among the underutilized MPEs, namely microprocessor(s) and/or processor-cores by providing a virtual SPMD view and thus improving the overall system utilization for multi-user environments. In other words, the pool of reconfigurable resources will be virtually increased to maintain the symmetric view of SPMD, i.e. unity ratio among the MPEs, CPEs, and their communication channels. The implementation of these concepts is based on Partial Run-Time Reconfiguration (PRTR) from a practical perspective. We will provide a formal stochastic analysis of the execution model supported by experimental work. The execution model considers multi-user HPRCs equipped with multi-processor/multi-core technology. Our work utilizes PRTR on one of the current HPRC systems, Cray XD1. The results show near-linear scalability behavior for compute intensive applications.

This paper is organized such that section 2 provides a discussion of related work in context of run-time reconfiguration and hardware virtualization. Section 3 describes the space, time, and space-time techniques for sharing reconfigurable resources. Section 3 also includes our analytical model and explains the formulation steps of this model. Section 4 shows both the theoretical and the experimental results. Finally, Section 5 summarizes and concludes the paper with our findings.

2 Related Work

In this work, the primary objective is to share the reconfigurable resources (or CPEs) in HPRCs among all system microprocessors and/or processor-cores (or MPEs) in an SPMD view, irrespective of the system physical limitations/configuration, thereby providing support for true multi-user environments. In other words, regardless of the number of main processing elements (MPEs) and the co-processing elements (CPEs) in the system, we will try to provide a virtual 1:1 correspondence between MPEs and CPEs. To achieve the desired objective, we will leverage previous work and concepts that have been used for solving similar and related problems, namely hardware virtualization. For example, we use the concept of virtual FPGA (VFPGA) proposed in [7].

Many of the proposed solutions in previous research [8], [9], are based on the strategies used in Operating Systems to support virtual memory – dynamic loading, partitioning, overlaying, segmentation, and paging, etc. All of these techniques strive to provide applications with the view of a larger FPGA, by virtually increasing the FPGA logic capacity. This concept of “virtual hardware” requires the use of special

capabilities of the FPGAs, namely, Full Run-Time Reconfiguration (FRTR) and/or Partial Run-Time Reconfiguration (PRTR) [10],[11]. However, all of these proposed techniques are targeted towards embedded systems, with typically a single main processing element (MPE) and only one reconfigurable co-processing element (CPE). The multiplicity and imbalanced heterogeneity of the processing elements, common to HPRCs, is absent in embedded platforms. Furthermore, HPRC systems impose architectural constraints such as a shared configuration interface for the CPEs, as well as shared communication interfaces between the MPEs and CPEs. The unique nature of HPRCs adds a significant complexity to the virtualization problem, and therefore calls for a formal approach in order to solve it. Towards this end, we utilize and build on the techniques and methodologies introduced in [10],[11] by providing a virtualization infrastructure that allows space, time, and/or space-time sharing of the reconfigurable processors. Furthermore, we generalize the execution model based on stochastic Markov chains and queueing networks. The new model includes HPRCs equipped with multi-processor/multi-core technology utilizing the proposed virtualization infrastructure in a true multi-user environment.

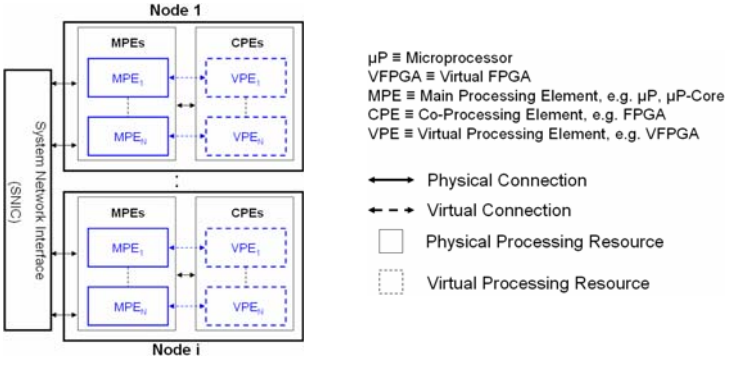


Fig. 1. Architectural assumptions (SPMD view of reconfigurable resources on HPRCs)

3 Techniques for Sharing Reconfigurable Resources

Our methodology is based on the concept of "Computing in Time - Computing in Space" [12] for space, time, and/or space-time sharing of the reconfigurable resources. We first develop a formal analysis of the execution model based on our methodology, following an approach similar to what has been proposed in [10],[11]. In our analysis, we assume a multi-processor/multi-core HPRC architecture with asymmetric heterogeneity at the node level [4]. All nodes are identical and in general each node is assumed to include a number of main processing elements (MPEs), e.g. microprocessors or processor cores, and a number of co-processing elements (CPEs), e.g. FPGAs. The number of MPEs, N_{MPE}, is not necessarily equal to the number of CPEs, N_{CPE}. In current HPRC systems N_{MPE} is typically larger than N_{CPE}, namely $N_{MPE} > N_{CPE}$. Additionally, each CPE is to be partitioned into a number of virtual processing elements (VPEs), N_{VPE}, such that each VPE is associated to a

corresponding MPE maintaining a One-to-One correspondence among MPEs and their dedicated VPEs resulting in a balanced and symmetric distribution of system resources. In other words, the physical reconfigurable resources (FPGAs) will be virtualized and split into multiple virtual FPGAs (VFPGAs) such that $N_{VFPGA} = N_{VPE}$ in order to accommodate for the symmetry requirement of the SPMD execution. Each VFPGA will be located in a separate partially reconfigured region (PRR) on the physical FPGA. Finally, the number of necessary VPEs, N_{VPE}, for providing /guaranteeing the SPMD behavior can be given by equation (1) as follows:

$$N_{VPE} = \left\lceil \frac{N_{MPE}}{N_{CPE}} \right\rceil \times N_{CPE} \tag{1}$$

As the number of VPEs increases, the size of each VPE reduces; the task granularity α_{task} will determine the maximum number of VPEs, as seen below by rewriting (1):

$$\begin{aligned} &N_{regions} = N_{VFPGA} = N_{VPE} = \min\left(N_{VPE}^{cores}, N_{VPE}^{tasks}\right) \times N_{CPE} \ , \ where \\ &N_{VPE}^{cores} \equiv \left\lceil \frac{N_{MPE}}{N_{CPE}} \right\rceil, \ N_{VPE}^{tasks} \equiv \left\lfloor \frac{1-\alpha_{static}}{\alpha_{task}} \right\rfloor, \ \alpha_{static} \le 1 \ , \ \alpha_{task} \le 1 \\ &\alpha_{static} \equiv static \quad region \quad FPGA \quad resource \quad utilization \\ &\alpha_{task} \equiv task \quad granularity \ (task \quad FPGA \quad resource \quad utilization) \end{aligned} \tag{2}$$

Based on equation (2), space-time scheduling of task execution on VPEs is needed when $N_{VPE}^{tasks} < N_{VPE}^{cores}$ while space-only scheduling is needed when $N_{VPE}^{tasks} \ge N_{VPE}^{cores}$. In other words the execution of tasks needs to be performed through both space and time schedules when the task granularity is the governing bound on the number of VPEs while only space schedules are needed when there is a sufficient number of VPEs; at least equal to the number of MPEs. In later discussions, we will refer to equation (2) as the SPMD condition.

The usage model is SPMD in which the system receives some applications as input. These applications require on average a few independent hardware functions (tasks) that need to be executed on dedicated reconfigurable resources. The execution cycle for any task on an HPRC consists of the computation time, the total data input time, output time and the configuration time [10],[11], represented by T_{comp}, T_{in}, T_{out}, and T_{config} respectively. The I/O time T_{in} and T_{out} represent the time necessary to transfer data between the microprocessor and the FPGA. The baseline for our analysis is FRTR, where reconfigurable resources (CPEs) cannot be space shared among the node microprocessors/processor-cores (MPEs). We will focus our discussions on applications that are broken down into hardware tasks only. In addition, we assume that each task is fully characterized by its time requirement, $T_{task} = T_{in} + T_{comp} + T_{out}$.

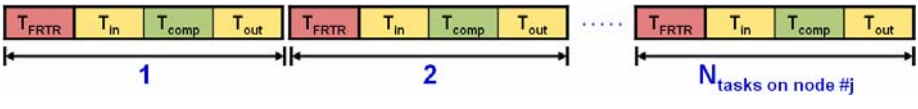


Fig. 2. Typical task execution per node using FRTR on a multi-user HPRC system

The execution model of FRTR on each node, see Fig. 2, is sequential among tasks and is independent from their owners (users). This is because the reconfigurable resource (CPE), assuming one per node, is not space sharable among the node MPEs rendering some MPEs unused. Considering different tasks to have similar average execution characteristics albeit different in functionalities, total execution time for the case of FRTR can be derived as follows:

$$
\begin{aligned}
&T_{task} = T_{in} + T_{comp} + T_{out} \\
&T_{task}^{FRTR} = T_{config}^{FRTR} + T_{task} = T_{FRTR} + T_{in} + T_{comp} + T_{out} \;,\; T_{user_i,node_j}^{FRTR} = \sum_{k=1}^{N_{tasks_{i,j}}} T_{task}^{FRTR} = N_{tasks_{i,j}} \cdot T_{task}^{FRTR} \\
&T_{node_j}^{FRTR} = \sum_{i=1}^{N_{users}} T_{user_i,node_j}^{FRTR} = \sum_{i=1}^{N_{users}} N_{tasks_{i,j}} T_{task}^{FRTR} = T_{task}^{FRTR} \sum_{i=1}^{N_{users}} N_{tasks_{i,j}} \\
&\Rightarrow T_{node_j}^{FRTR} = N_{tasks_{node_j}} T_{task}^{FRTR} \;,\; where \\
&T_{config}^{FRTR} \equiv T_{FRTR} \equiv Average\ \ full\ config.\ time\ of\ any\ task\ generated\ by\ any\ user\ on\ node\ \#\ j
\end{aligned}
\tag{3}
$$

$T_{in} \equiv$ *Average input transfer time of any task generated by any user on node # j*

$T_{comp} \equiv$ *Average computation time of any task generated by any user on node # j*

$T_{out} \equiv$ *Average output transfer time of any task generated by any user on node # j*

$T_{task} \equiv$ *Average execution time of any task generated by any user on node # j*

$T_{task}^{FRTR} \equiv$ *Average execution time of any task generated by any user on node # j for FRTR*

$T_{user_i,node_j}^{FRTR} \equiv$ *Total execution time of all tasks generated by user #i on node # j for FRTR*

$T_{node_j}^{FRTR} \equiv$ *Total execution time of all tasks generated by all users on node # j for FRTR*

3.1 Queuing Analysis and Modeling

The execution model of our proposed virtualization technique and sharing mechanism can be viewed as a combination of three traffic (queueing) processes, namely entry, computation, and exit processes, see Fig. 3(a). The entry process is when tasks at the beginning of their execution life-cycle request configuration and data transfer from the MPEs into the VPEs/VFPGAs. The exit process is when tasks at the end of their execution life-cycle request data transfer from the VFPGA back to the MPEs. The computation process represents the actual processing performed by tasks on their VFPGAs. In our model tasks can continue their computations in parallel while others are entering into and/or exiting from the system. In other words, we are considering that VFPGA reconfiguration, data transfers (in and out) and computations can be overlapped, sharing the I/O channel among all the MPEs in the node.

We base our analysis of the execution model on Markov processes. In particular, we will utilize the mathematical formulation of discrete-parameter (discrete-time) Markov chains [13]. Markov chains are described in general using a state diagram in which each state represents a case when the system contains a certain number of customers (tasks in our case). The system transitions from one state S_i to another state S_j with a probability p_{ij} known as the one-step transition probability, see Fig. 3(b). The

probability distribution of a Markov chain is completely determined by the one-step transition probability matrix, $P=[p_{ij}]$, and the initial-state probability vector [13], see equation (4). Equation (5) shows some important and useful properties of P.

$$\vec{p}(n) = P^T \cdot \vec{p}(n-1) = \left(P^T\right)^n \cdot \vec{p}(0) = S \cdot \Lambda^n \cdot S^{-1} \cdot \vec{p}(0) \quad , \quad where$$

$$\vec{p}(n) = \begin{bmatrix} p_0(n) \\ p_1(n) \\ p_2(n) \\ . \\ p_k(n) \\ . \\ p_{N_{tasks}}(n) \end{bmatrix}, \quad \vec{p}(0) = \begin{bmatrix} p_0(0) \\ p_1(0) \\ p_2(0) \\ . \\ p_k(0) \\ . \\ p_{N_{tasks}}(0) \end{bmatrix}, \quad P = \begin{array}{c} \\ 0 \\ 1 \\ . \\ i \\ . \\ N_{tasks} \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & . & j & . \end{array} \; N_{tasks} \\ \begin{bmatrix} p_{00} & p_{01} & . & p_{0j} & . & p_{0N_{tasks}} \\ p_{10} & p_{11} & . & p_{1j} & . & p_{1N_{tasks}} \\ . & . & . & . & . & . \\ p_{i0} & p_{i1} & . & p_{ij} & . & p_{iN_{tasks}} \\ . & . & . & . & . & . \\ p_{N_{tasks}0} & p_{N_{tasks}1} & . & p_{N_{tasks}j} & . & p_{N_{tasks}N_{tasks}} \end{bmatrix} \end{array} \tag{4}$$

$$\vec{p}(n) \equiv state \;\; probability \;\; vector \;\; after \;\; n \;\; transitions$$

$$p_k(n) \equiv probability \;\; of \;\; the \;\; system \;\; being \;\; in \;\; state \;\; S_k \;\; (having \;\; k \;\; tasks) \;\; after \;\; n \;\; time \;\; steps$$

$$\vec{p}(0) \equiv initial - state \;\; probability \;\; vector$$

$$P \equiv Transition \;\; probability \;\; matrix$$

$$P^T \equiv Transposed \;\; Transition \;\; probability \;\; matrix$$

$$S \equiv Matrix \;\; of \;\; Eigenvectors \;\; of \;\; P^T$$

$$\Lambda \equiv Matrix \;\; of \;\; Eigenvalues \;\; of \;\; P^T$$

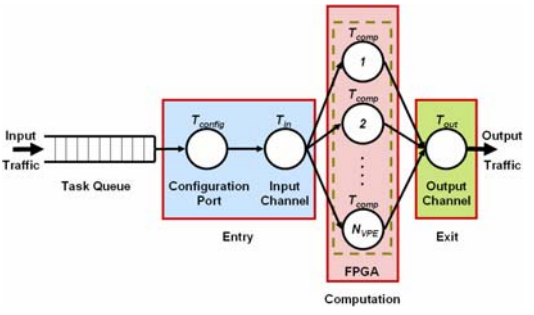
$$\left.\begin{array}{l} 0 \le p_{ij} \le 1, \quad \sum\limits_{j=0}^{N_{tasks}} p_{ij} = 1, \quad \sum\limits_{k=0}^{N_{tasks}} p_k(n) = 1 \\ p_{ij} = \begin{cases} r_{ij} \cdot \Delta t = \dfrac{r_{ij}}{r_{sampling}} \quad , \quad j \ne i \\ 1 - \sum\limits_{\substack{k=0 \\ k \ne i}}^{N_{tasks}} p_{ik} = 1 - \sum\limits_{\substack{k=0 \\ k \ne i}}^{N_{tasks}} \dfrac{r_{ik}}{r_{sampling}} \quad , \quad j = i \end{cases} \\ r_{sampling} \ge \max\limits_{i=0}^{N_{tasks}} \left(\sum\limits_{\substack{k=0 \\ k \ne i}}^{N_{tasks}} r_{ik} \right) \equiv \max \left(\sum\limits_{\substack{k=1 \\ k \ne 0}}^{N_{tasks}} r_{0k}, \sum\limits_{\substack{k=0 \\ k \ne 1}}^{N_{tasks}} r_{1k}, \; \ldots \; , \sum\limits_{\substack{k=0 \\ k \ne N_{tasks}}}^{N_{tasks}} r_{N_{tasks}k} \right) \end{array}\right\} , \; where \tag{5}$$

$$r_{ij} \equiv Transition \;\; rate \;\; from \;\; state \;\; S_i \;\; to \;\; state \;\; S_j$$

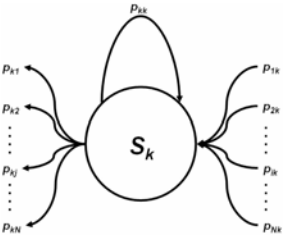
$$\Delta t \equiv Time \;\; step \;\; duration$$

$$r_{sampling} \equiv \frac{1}{\Delta t} \equiv Time \;\; sampling \;\; rate$$

The total execution time on node j can be determined/defined by the time step at which the instantaneous state probability vector becomes very close, within a certain error threshold ε, to its final steady state value. In other words, we define the total execution time as the minimum time step that is necessary for the system to reach as close as possible its final steady state where behavior transients become



(3a) Traffic processes



(3b) Discrete-time Markov chains state diagram

Fig. 3. Queueing execution model

insignificant to a certain error threshold ε. This argument can be described as follows by equation (6):

$$\left.\begin{aligned} & T_{node_j}^{PRTR} = n_{\min} \cdot \Delta t = \frac{n_{\min}}{r_{sampling}} \\ & \lim_{n\to\infty} \left\| \vec{p}(n) - \vec{p}(n_{\min}) \right\| \equiv \left\| \vec{p}(\infty) - \vec{p}(n_{\min}) \right\| \le \varepsilon \\ & \Rightarrow \sqrt{\sum_{i=0}^{N_{tasks}} \left(p_i(\infty) - p_i(n_{\min}) \right)^2} \le \varepsilon \end{aligned}\right\}, \ where \tag{6}$$

$T_{node_j}^{PRTR} \equiv Total\ execution\ time\ of\ all\ tasks\ generated\ by\ all\ users\ on\ node\ \#\ j\ for\ PRTR$

$n_{\min} \equiv Minimum\ number\ of\ time\ steps\ necessary\ for\ close-to-steady-state\ behavior$

$\vec{p}(\infty) \equiv Steady\ state\ probability\ vector$

$\varepsilon \equiv Error\ threshold$

Taking into consideration that the final execution time on the system is determined by the longest execution time among all nodes, namely the slowest (critical) node, the performance gain (speedup) of PRTR in reference to FRTR can be expressed as follows by combining equations (3) and (6):

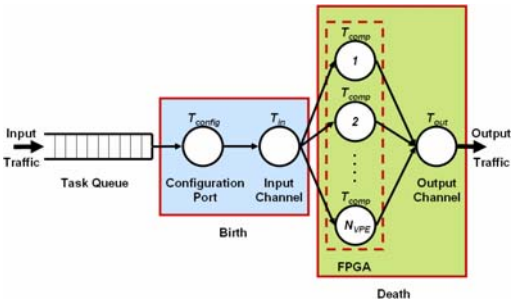
$$\begin{aligned} & T_{total}^{FRTR} = \underset{j=1}{\overset{N_{nodes}}{MAX}} \left(T_{node_j}^{FRTR} \right) = T_{node_j}^{FRTR}, \ T_{total}^{PRTR} = \underset{j=1}{\overset{N_{nodes}}{MAX}} \left(T_{node_j}^{PRTR} \right) = T_{node_j}^{PRTR}, \ S \equiv \frac{T_{total}^{FRTR}}{T_{total}^{PRTR}} \\ & \Rightarrow S = \frac{N_{tasks_{node_j}} \left(T_{FRTR} + T_{in} + T_{comp} + T_{out} \right)}{T_{node_j}^{PRTR}}, \ where \end{aligned} \tag{7}$$

$T_{total}^{FRTR} \equiv Total\ average\ execution\ time\ of\ all\ tasks\ generated\ by\ all\ users\ on\ all\ nodes\ for\ FRTR$

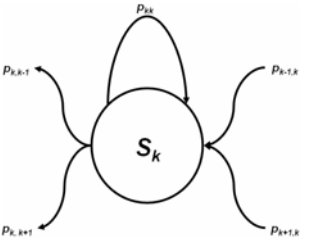
$T_{total}^{PRTR} \equiv Total\ average\ execution\ time\ of\ all\ tasks\ generated\ by\ all\ users\ on\ all\ nodes\ for\ PRTR$

$S \equiv Speedup\ or\ performance\ gain\ of\ PRTR\ relative\ to\ FRTR$

Due to the fact that typical HPRC architectures are designed with a single configuration port and a single communication channel between MPEs and CPEs, we



(4a) Traffic processes



(4b) Birth-Death state model

Fig. 4. Simplified execution model

will use a special class of Markov chains that is typically used to describe queueing systems. More specifically, we will simplify our model as a birth-death process in which transitions are allowed between only neighboring states. The simplified execution model is shown in Fig. 4. Equation (8) describes the simplified model.

$$\vec{p}(0) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ . \\ 0 \\ . \\ 0 \end{bmatrix}, \quad P = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ . \\ i \\ . \\ N_{tasks} \end{array} \begin{array}{c} \begin{array}{ccccccccc} 0 & 1 & 2 & 3 & . & i-1 & i & i+1 & . & N_{tasks} \end{array} \\ \begin{bmatrix} p_{00} & p_{01} & 0 & 0 & . & 0 & 0 & 0 & . & 0 \\ p_{10} & p_{11} & p_{12} & 0 & . & 0 & 0 & 0 & . & 0 \\ 0 & p_{21} & p_{22} & p_{23} & . & 0 & 0 & 0 & . & 0 \\ 0 & 0 & p_{32} & p_{33} & . & 0 & 0 & 0 & . & 0 \\ . & . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & . & p_{i,i-1} & p_{ii} & p_{i,i+1} & . & 0 \\ . & . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & . & 0 & 0 & 0 & . & p_{N_{tasks}N_{tasks}} \end{bmatrix} \end{array}, \quad where \tag{8}$$

$$r_{ij} = \begin{cases} \lambda \ , \ 0 \le i < N_{tasks} \ , \ j = i+1 \\ i\mu \ , \ 0 < i < N_{VPE} \ , \ j = i-1 \\ N_{VPE}\mu \ , \ N_{VPE} \le i \le N_{tasks} \ , \ j = i-1 \\ 0 \ , \ otherwise \end{cases}$$

$$\lambda = \frac{1}{T_{config} + T_{in}} \ , \ and \ \ \mu = \frac{1}{T_{comp} + N_{VPE} \cdot T_{out}}$$

In order to investigate how scalable our approach, we will introduce what we call the scalability factor, η. The scalability factor, η, can be defined as the normalized speedup. In other words, the speedup achieved by a multiple of MPE-VPE pairs would be normalized with respect to the speedup achieved by one MPE-VPE pair. More specifically, η is defined as the ratio between two values of the speedup, namely $S(N_{VPE})$ and $S(1)$, as a function of N_{VPE}. This expression can be written as shown in equation (9). By taking the limit of equation (9) as the number of VPEs increases indefinitely, namely $N_{VPE} \rightarrow \infty$, the asymptotic scalability behavior can be obtained as given by equation (10).

$$\eta(N_{VPE}) \equiv \frac{S(N_{VPE})}{S(1)}$$

$$S(N_{VPE}) = \frac{N_{tasks_{node\,j}} \left(T_{FRTR} + T_{in} + T_{comp} + T_{out}\right)}{T_{node\,j}^{PRTR}(N_{VPE})} \quad , \quad S(1) = \frac{N_{tasks_{node\,j}} \left(T_{FRTR} + T_{in} + T_{comp} + T_{out}\right)}{T_{node\,j}^{PRTR}(1)}$$

$$\Rightarrow \eta(N_{VPE}) = \frac{T_{node\,j}^{PRTR}(1)}{T_{node\,j}^{PRTR}(N_{VPE})} \tag{9}$$

$$\Rightarrow \eta_{\infty} \equiv \lim_{N_{VPE} \to \infty} \eta(N_{VPE}) = \frac{T_{node\,j}^{PRTR}(1)}{T_{node\,j}^{PRTR}(\infty)} \quad , \quad where \tag{10}$$

$\eta(N_{VPE}) \equiv Scalability\ \ factor\ \ as\ \ a\ \ function\ \ of\ \ the\ \ number\ \ of\ \ node\ \ VPEs,\ \ N_{VPE}$

$\eta_{\infty} \equiv Asymptotic\ \ Scalability\ \ factor\ \ as\ \ the\ \ number\ \ of\ \ VPEs\ \ increases\ \ indefinitely$

4 Results

Our experiments have been performed on one of the current HPRC systems, Cray XD1 [3]. The Cray XD1 is a multi-chassis system. Each chassis contains up to six nodes (blades). Each blade consists of two 64-bit AMD Opteron processors at 2.4 GHz, one Rapid Array Processor (RAP) that handles the communication, an optional second RAP, and an optional Application Accelerator Processor (AAP). The AAP is a Xilinx Virtex-II Pro XC2VP50-7 FPGA with a local memory of 16MB QDR-II SRAM [3].

To verify the proposed virtualization techniques and the execution model, a set of experiments were conducted, starting with an application that carries out image feature extraction. In the chosen application, high frequency noise components were first removed from the images using two different algorithms, followed by some processing to extract the object edges of interest. Specifically, a sequence of image processing functions were executed, namely median filtering followed by Sobel edge detection, and smoothing filtering also followed by Sobel edge detection. The final images were then transferred back to the microprocessor memory for some quality checks.

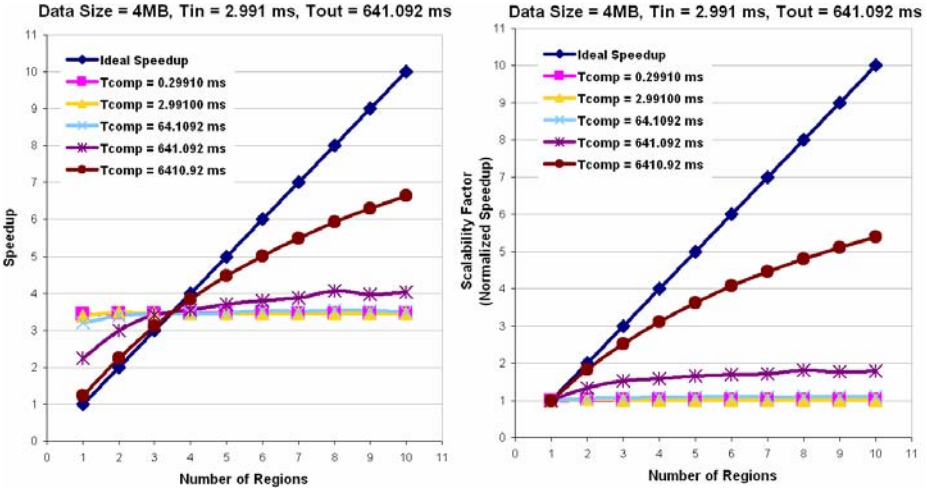
Table 1. Selected scenarios for Cray XD1

	Case 1 (T_{comp}<T_{in}<T_{out})	**Case 2** (T_{comp}=T_{in}<T_{out})	**Case 3** (T_{in}<T_{comp}<T_{out})	**Case 4** (T_{in}<T_{comp}=T_{out})	**Case 5** (T_{in}< T_{out}<T_{comp})
T_{comp} **(msec)**	0.299	2.991	64.109	641.092	6410.92

It may be noted that the SPMD condition as described by equation (2) suggests that the maximum number of PRRs should at least equal the number of microprocessors (MPEs) per node. For Cray XD1, the number of MPEs per node is two. We therefore conducted an initial set of experiments using dual VFPGAs (VPEs). In order to evaluate the proposed execution model for a larger number of cases, we added some features to the virtual infrastructure on Cray XD1 to emulate scenarios for a larger

number of VFPGAs (PRRs). The emulation-based virtual infrastructure accepts a minimum set of parameters for XD1 since it is running on the machine itself. These parameters include the number of VFPGAs and different computation times to emulate different tasks, etc. Five scenarios were emulated to validate the model and the proposed infrastructure as shown in Table 1. These scenarios were selected to investigate different classes of applications starting from the least computational intensive, namely I/O intensive, in case 1 to the most computational intensive applications in case 5, see Table 1. A large (infinite) amount of task traffic was submitted to be executed on a variable number of VPEs from 1 to 10 VFPGAs.

Results for the described scenarios were obtained from actual runs on Cray XD1, and compared against the proposed execution model presented in Section 3. The measured results were found to be in good agreement with the mathematical model. Fig. 5 shows some of these experimental findings for the scenarios listed in Table 1, as a speedup over the conventional execution based on FRTR. The parameters collected from our experiments are T_{FRTR}= 1678.040 ms, T_{PRTR}= 19.771 ms, T_{in}=2.991 ms, and T_{out}= 641.092 ms. Equation (7) suggests that the speedup value should be 3.49, which is consistent with the value measured and shown in Fig. 5(a).



(5a) Speedup achieved using multiple PRRs (VFPGAs) as a function of $N_{VPE} = N_{regions}$

(5b) Scalability factor, η

Fig. 5. Performance of applications using virtual resources

It is worth mentioning that for the measured parameters on Cray XD1 there is a region in Fig. 5(a) where the measured speedup is not upper bounded by the total number of processing elements, N_{MPE}. The upper bound is rather dictated by the ratio between T_{FRTR} and T_{PRTR}. This is true to a certain point, see Fig. 5(a), beyond which the situation reverses and the speedup would be upper bounded by the total number of processing elements, N_{VPE}. This is due to the fact that for a small number of VPEs the savings in the total execution time is not because of the parallel execution of tasks but

rather because of the savings in (re)configuration overhead. On the other hand, for a large number of VPEs the savings in the total execution time because of the parallel execution of tasks become more significant than the savings in (re)configuration overhead.

Finally, the scalability, as defined by equation (9), of our approach is shown in Fig. 5(b). In general, HPC applications with constant overhead show a similar scalability behavior to the one shown in Fig. 5(b). Such behavior is typically due to communication overhead between the system nodes. In our case, the overhead is due to (re)configuration and data transfer back and forth between the MPEs and VPEs, see equation (10). As shown in Fig. 5(b), when the task computation time, T_{comp}, becomes much larger than the associated overhead, the execution speedup, using our techniques, approaches linear behavior. In other words, the execution of highly compute intensive applications using our virtualization techniques becomes linearly scalable, which is a typical behavior on HPC supercomputers.

5 Conclusion

In this paper we presented an effort of virtualizing and space, time, and/or space-time sharing of reconfigurable resources based on Partial Run-Time Reconfiguration (PRTR) for High-Performance Reconfigurable Computing (HPRC) systems configured with multi-processor/multi-core technologies. We investigated the performance potential of our proposed virtualization techniques on HPRCs from both theoretical and practical perspectives. In doing so, we derived a formal stochastic model of multi-user SPMD execution on HPRC systems relative to the baseline of Full Run-Time Reconfiguration (FRTR). The model provided us with theoretical expectations which served as a frame of reference against which we projected our experimental results. In addition, it helped us gain in-depth insight about the boundaries and/or conditions for possibilities of performance gain using PRTR for resource sharing and virtualization. In achieving this objective, our approach was based on leveraging previous work and concepts that were introduced for solving similar and related problems.

In conducting the experimental work, we utilized one of the current HPRC systems, Cray XD1. We also discussed the requirements and setups for PRTR-based resource virtualization on Cray XD1. The experimental results showed good agreement with the analytical model expectations. Sharing reconfigurable resources among the underutilized microprocessors/processor-cores by providing a virtual SPMD view allows improving the overall system versatility, resources utilization, and application performance in multi-user environments. The approach we followed for Cray XD1 has been proven to be scalable and general to be applied to any of the available HPRC systems.

References

1. SRC Computers, Inc.: SRC Carte™ C Programming Environment v2.2 Guide SRC-007-18 (2006)
2. Silicon Graphics Inc.: Reconfigurable Application-Specific Computing User's Guide 007-4718-005 (2007)

3. Cray Inc.: Cray XD1™ FPGA Development S-6400-14 (2006)
4. El-Ghazawi, T., El-Araby, E., Huang, M., Gaj, K., Kindratenko, V., Buell, D.A.: The Promise of High-Performance Reconfigurable Computing. IEEE Computer 41(2), 69–76 (2008)
5. NIST: Single Program Multiple Data, http://www.nist.gov/dads/HTML/singleprogrm.html
6. Darema, F.: SPMD Model: Past, Present and Future. In: Cotronis, Y., Dongarra, J. (eds.) PVM/MPI 2001. LNCS, vol. 2131, p. 1. Springer, Heidelberg (2001)
7. Fornaciari, W., Piuri, V.: General Methodologies to Virtualize FPGAs in HW/SW Systems. In: Midwest Symposium on Circuits and Systems, pp. 90–93 (1998)
8. Li, Z., Hauck, S.: Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation. In: ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 187–195 (2002)
9. Li, Z., Compton, K., Hauck, S.: Configuration Caching Management Techniques for Reconfigurable Computing. In: IEEE Symposium on FPGAs for Custom Computing Machines, pp. 87–96 (2000)
10. El-Araby, E., Gonzalez, I., El-Ghazawi, T.: Exploiting Partial Runtime Reconfiguration for High-Performance Reconfigurable Computing. ACM Transactions on Reconfigurable Technology and Systems 1(4), 21:1–21:23 (2009)
11. El-Araby, E., Gonzalez, I., El-Ghazawi, T.: Virtualizing and Sharing Reconfigurable Resources in High-Performance Reconfigurable Computing Systems. In: 2nd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA 2008), pp. 1–8 (2008)
12. Siemers, C.: Reconfigurable Computing between Classifications and Metrics - The Approach of Space/Time-Scheduling. In: Grünbacher, H., Hartenstein, R.W. (eds.) FPL 2000. LNCS, vol. 1896, pp. 769–772. Springer, Heidelberg (2000)
13. Lipsky, L.R.: Queueing Theory: A Linear Algebraic Approach. Macmillan, New York (1992)

Routing-Aware Application Mapping Considering Steiner Points for Coarse-Grained Reconfigurable Architecture

Ganghee Lee[1], Seokhyun Lee[1], Kiyoung Choi[1], and Nikil Dutt[2]

[1] Department of Electrical Engineering and Computer Science
Seoul National University, Seoul, Korea
berean97@snu.ac.kr, shmir@poppy.snu.ac.kr, kchoi@snu.ac.kr
[2] Department of Computer Science
University of California, Irvine, Irvine, CA, USA
dutt@ics.uci.edu

Abstract. Coarse-grained reconfigurable architectures have drawn increasing attention due to their performance and flexibility. While many coarse-grained reconfigurable architectures have demonstrated impressive performance improvements, their effectiveness heavily depends on the quality of the compilers and/or mappers. However, this mapping process is difficult since it requires the solution of multiple problems simultaneously: compilation of the application and configuration of the architecture while maximally exploiting the parallelism in both the application and the architecture. Utilization of routing resources also adds to the complexity of the mapping process. In this paper, we introduce routing-aware mapping algorithms for coarse-grained reconfiguration architecture. In particular, we consider Steiner point routing, since it gives better results than spanning tree based routing. After presenting an optimal formulation using integer linear programming (that doesn't scale), we present a fast heuristic mapping algorithm. Our experimental result on randomly generated examples shows that our algorithm considering Steiner point routing gives 10% better performance result than the one using spanning tree routing. And our heuristic algorithm finds optimal solutions for 96% of the cases on the average within a few seconds. We also convey similar results on a suite of benchmarks collected from Livermore loops, Mediabench, and DSPStone benchmarks.

Keywords: Coarse-grained reconfigurable architecture, high-level synthesis, mapping, routing.

1 Introduction

With the increasing requirements for more flexibility and higher performance in embedded systems design, reconfigurable computing is becoming more and more popular. Typically, the coarse-grained reconfigurable architectures (CGRAs) consist of a reconfigurable array of processing elements (PEs) and a host processor (RISC or VLIW architecture). The computation intensive kernels of the applications – typically

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 231–243, 2010.

loops – are mapped to the reconfigurable array while the remaining code is executed by the processor. However it is not easy to map an application to the reconfigurable array because of the high complexity of the problem that requires compiling the application on a dynamically reconfigurable parallel architecture, with additional complexity of dealing with complex routing resources. The problem of mapping an application onto a CGRA to minimize the number of resources giving best performance has been shown to be NP-complete [13]. Few automatic mapping/ compiling/synthesis tools have been developed to exploit the parallelism found in the applications and extensive computation resources of CGRAs. Some research [1] uses GUI-based design tools to manually generate a mapping, which would have difficulty in handling big designs. Some researchers [2][3] only focus on instruction-level parallelism, failing to fully utilize the resources in CGRAs, which can be achieved by exploiting loop-level parallelism. More recent efforts [4]-[9] introduce compilers to exploit the parallelism in the CGRA to better utilize the resources of CGRA. However, they neither guarantee optimal solutions nor show how close the solutions are to the optimal. Furthermore, some researchers [4][5][9] use shared registers to solve mapping problem. Although these shared registers simplify the mapping process, they may increase the critical path delay and could be eliminated if routing resources were considered explicitly during the mapping process. In [6], they introduce edge-centric modulo scheduling, which is similar to our approach in the sense of routing-awareness. However, their routing approach is based on spanning tree, thus it may give worse result than the one extended to Steiner tree.

In this paper, we first present a novel integer linear programming (ILP) formulation considering Steiner points to optimally map applications on the CGRA. Since this formulation also considers Steiner points, it yields better solutions than the one using only spanning tree. However, ILP exhibits long run-times for large problem sizes and is therefore unsuitable for design space exploration. In this paper, we also present a fast heuristic mapping algorithm for CGRA that is routing aware and incorporates Steiner points.

2 Coarse-Grained Reconfigurable Architecture

Our target architecture consists of a reconfigurable computing module (RCM) for executing kernel code segments and a general purpose processor for controlling the RCM, which are connected with a shared bus. The RCM used in our platform consists of an array of PEs, several sets of frame buffers, and a configuration cache memory [10]. Fig. 1 shows our CGRA and internal structure of the PEs. It is connected with the nearest neighboring PEs-top, bottom, left, and right. The size of the array can be optimized to a specific application domain [10]. In Fig. 1, for example, the architecture contains a 4x4 reconfigurable array of PEs. The area-critical functional units (such as multipliers) are located outside the PEs and shared among a set of PEs [10]. Each area-critical functional unit is pipelined to curtail the critical path delay, and its execution is initiated by scheduling the area-critical operation on one of the PEs that share this area-critical resource. Thus each PE can be dynamically reconfigured either to perform arithmetic and logical operations with its own ALU in one clock cycle, or to perform multiplication or division operations using the corresponding shared functional unit in several clock cycles with pipelining.

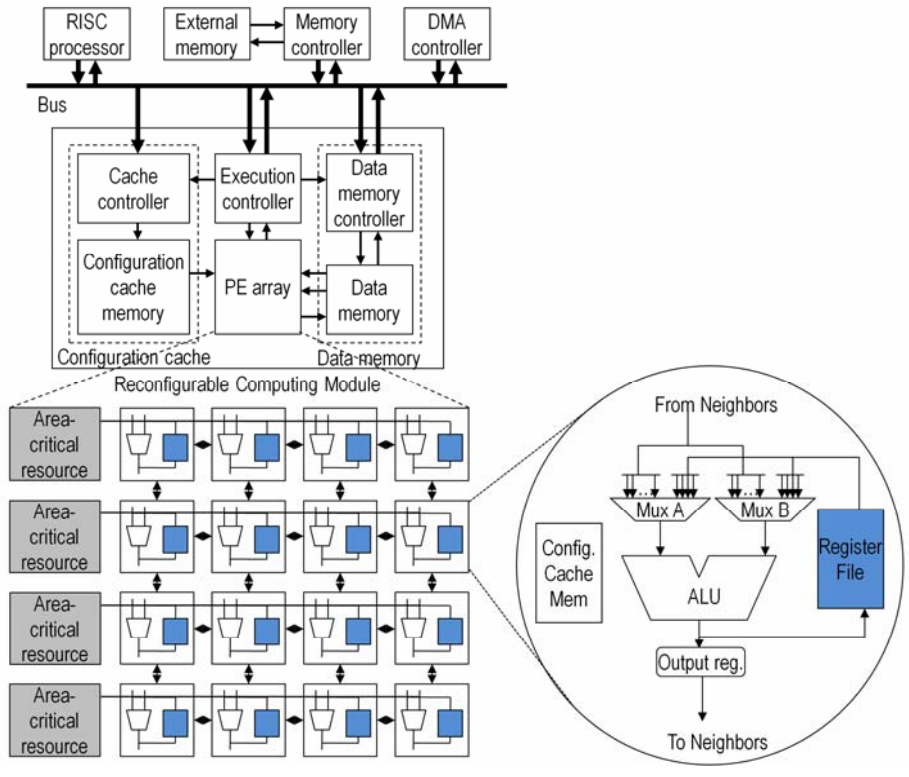


Fig. 1. Coarse-grained reconfigurable architecture

(a) Spanning tree (b) Steiner tree

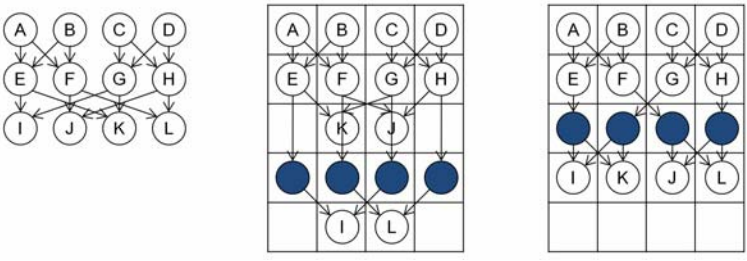
Fig. 2. Spanning tree vs. Steiner tree

The data memory in Fig. 1 is used for storing data that can be accessed by the PEs. There are two sets of memory, each of which consists of three banks: one connected to the write bus and the other two connected to the read buses. These read/write buses are also shared by the PEs like the shared functional units. Two sets of memory are used for double buffering.

The configuration cache is composed of an array of memory elements having the same size as the array of PEs, that is M (number of PEs in a column) by N (number of PEs in a row) array of Cache Elements (CEs). Note that the area-critical resources are

activated through individual PEs that share these resources in each row and thus need not be explicitly modeled as PEs. Each CE has *H* layers so that each PE can be reconfigured independently of other PEs.

When mapping kernels onto the reconfigurable architecture, we use temporal mapping technique [11] to execute one iteration of a loop with one column of PEs by changing the configurations dynamically. Other columns are used for executing other iterations of the loop to achieve loop-level pipelining.



(a) Data flow graph (b) Routing with spanning tree (c) Routing with Steiner tree

Fig. 3. Motivational example that Steiner tree based routing gives better result

3 Motivation

3.1 Spanning Tree vs. Steiner Tree

Fig. 2 shows the difference between spanning tree and Steiner tree. The shortest Steiner tree problem is as follows: given a set of vertices, interconnect them by a set of edges such that the sum of the lengths of all edges is minimized. The difference from the shortest spanning tree problem is that extra intermediate vertices (Steiner points or Steiner vertices) and edges may be added in order to reduce the sum of the lengths of all edges.

3.2 Motivational Example

Fig. 3 shows a motivational example illustrating how a Steiner tree gives a better solution than that obtained by considering only spanning tree for the mapping problem. Here, we assume temporal mapping onto a column of four PEs, which has mesh connectivity. Fig. 3(a) shows the data flow graph of a butterfly addition example. A vertex represents an operation to execute, and an edge represents data-dependency. The previous approaches [4]-[9] use minimum spanning tree for routing, by finding a shortest path in the architecture for every edge in the data flow graph. Fig. 3(b) shows that the total latency obtained by applying spanning tree is 5. However, if we consider Steiner tree, we can reduce the latency down to 4 as shown in Fig. 3(c).

4 Problem Formulation

4.1 Notation and Definition

The main objective is to map a given loop kernel to the CGRA such that the total latency is minimized while satisfying several constraints. In this section we define the notations that are used throughout this paper, and formulate our problem using integer linear programming (ILP).

Loop kernel. The loop kernel is represented by a control data flow graph (CDFG), $K=(V, E, D)$, where V is the set of vertices, E is the set of data-dependency edges, and D is the set of loop-carried dependency edges. The vertices in V represent the operations in the loop, and an edge $e=(u, v) \in E$ exists for a pair of vertices, $u, v \in V$, iff operation v is data-dependent on operation u, and edge $d=(u, v) \in D$ exists for a pair of vertices, $u, v \in V$, iff operation v has loop-carried dependency on operation u.

CGRA. A column of the CGRA which has M PEs with H configuration cache layers can also be represented by a graph $C=(P, L)$, where P is the set of vertices and L is the set of edges. For a given column of the CGRA, vertex $p_{kl} \in P$, $1 \leq k \leq H$, $1 \leq l \leq M$, represents the *l-th* PE that is configured by the *k-th* layer of the configuration cache. For any pair of vertices $p, q \in P$, an edge $l=(p, q) \in L$ exists, iff there is an interconnection between PEs corresponding to p and q.

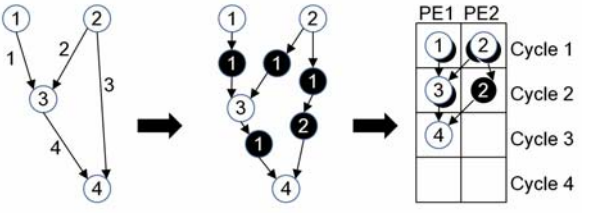
Application mapping. The application mapping problem is formulated as follows. Given a kernel graph $K=(V, E, D)$ and a CGRA graph $C=(P, L)$, find a mapping of K onto C with the objective of minimizing total latency under resource constraints. Note that the problem is not just finding a subgraph of C that is isomorphic to K, since we should also consider the case where a PE sends the output data to the destination indirectly using other PEs as routing resources

4.2 ILP Formulation

The problem of application mapping onto a CGRA has been proven to be NP-complete [13], even in the special case where the application is represented by a complete binary tree and the CGRA consists of a two dimensional grid with just the neighboring connections. In the absence of a polynomial-time exact algorithm, we formulate the problem as an integer linear programming (ILP) problem to find optimal solutions. The novelty of our ILP formulation is that we consider Steiner points routing to yield better results.

Routing vertices and routing PEs. When two vertices are mapped respectively to two PEs that have no interconnections between them, we add extra dummy vertices for data forwarding. Such vertices are called *routing vertices*, and the corresponding PEs are called *routing PEs*. To accommodate the routing PEs, we insert extra routing vertices into each edge of K. Since the exact number of routing PEs required for an edge $e_i \in E$ is not known until the mapping is completed, we insert the maximum possible number of routing vertices. Let R^{ei} be the set of inserted routing vertices into e_i, and let $R= \cup_{\forall i} R^{ei}$ be the set of all routing vertices. An upper bound of the number

of routing PEs for edge e_i is given by T^w_i-1 (i.e. $|R^{ei}| \leq T^w_i$-1), where T^w_i is obtained by subtracting ASAP schedule time of the tail vertex of edge e_i from ALAP schedule time of the head vertex. The worst-case latency of K (which is used in the ALAP scheduling), is obtained by assuming that there is only one PE. Then we take transformation $K \rightarrow K'=(V', E', D)$ by inserting $|R^{ei}|$ vertices into each edge $e_i \in E$. Fig. 4 shows the original kernel K (Fig. 4(a)) transformed into K' by inserting the candidates of routing vertices (dark vertices in Fig. 4(b)) into every edge in K. Now the problem of mapping K to C is transformed to the problem of mapping K' to C. Unlike normal vertices, the candidates of routing vertices for an edge can be mapped to the same PE at the same control step, including the PE on which a normal vertex (head or tail vertex of the edge) is mapped.



(a) Kernel K (b) K' (c) Mapping result

Fig. 4. Example of ILP formulation

Boolean decision variables

- v_{ikl} is 1 if *i-th* vertex $v_i \in V$ is mapped onto $p_{kl} \in P$.
- r_{ijkl} is 1 if *j-th* candidate of routing vertex $r_j \in R^{ei}$ for $e_i \in E$ is mapped onto $p_{kl} \in P$.
- q_{ijkl} is 1 if *j-th* routing vertex $r_j \in R^{ei}$ is mapped onto $p_{kl} \in P$, but no normal vertex is mapped onto p_{kl}, which means p_{kl} is used as an actual routing PE for the routing vertex.

Objective function. The objective to be minimized is the total latency when we map K onto C. For this, we add a sink vertex (v_s) to K and calculate the total latency (T) as follows.

$$T = \alpha + II \cdot (\lceil \beta / F \rceil - 1), \quad \text{when } \beta/F \leq N \tag{1}$$

$$(\alpha + II \cdot (N-1)) \cdot \lceil \beta / F / N \rceil, \quad \text{when } \beta/F > N \tag{2}$$

where $\alpha = \sum_{k}^{H} \sum_{l}^{M} k \cdot v_{skl}$ is the latency of sink vertex, II is the initiation interval calculated from D (set of loop-carried dependency edges), β is the number of iterations, F is the unroll factor, and N is the number of columns in the CGRA.

Constraints. Among many constraints, we only show the constraints related to Steiner point routing in this paper, due to the space limitation.

For $1 \le i \le |V|(or|E|),\ 1 \le j \le |R^e|,\ 1 \le k \le H,\ 1 \le l \le M$,

- Single mapping to a PE at a time:

$$\sum_{i}^{|V|} v_{ikl} \le 1, \forall k, l \tag{3}$$

$$r_{ijkl} + \sum_{i}^{|V|} v_{ijk} \in (V - \{v_q\}) \le 1, \forall k, l, i, j \tag{4}$$

where V-$\{v_q\}$ is the subset of V excluding vertex v_q, which is the head vertex of edge $e_i = (v_p, v_q)$.

- Allow Steiner points:

$$q_{ijkl} + \sum_{i}^{|V|} v_{ikl} + \sum_{i}^{|E|} \sum_{j}^{|R^e|} q_{ijkl} \in (R^{ei} - \{r_p\}) \le 1, \forall k, l, i, j \tag{5}$$

where R^{ei}-$\{r_p\}$ is the subset of R^{ei} excluding routing vertices r_p which has the same predecessor vertex with current q_{ijkl}.

- $\{q\}$ is a subset of $\{r\}$:

$$q_{ijkl} - r_{ijkl} \le 0, \forall k, l, i, j \tag{6}$$

- Routing PE (r=1 but no v is mapped) should set q:

$$q_{ijkl} - r_{ijkl} + \sum_{i}^{|V|} v_{ikl} \ge 0, \forall k, l, i, j \tag{7}$$

5 Design Flow and Mapping

5.1 Overall Design Flow

Fig. 5 describes the overall design flow, which integrates the process of mapping of kernels on the reconfigurable array. The mapping process is based on high-level synthesis (HLS) techniques. Since the HLS techniques and loop-level pipelining fit well with temporal mapping, we focus only on temporal mapping in this paper.

We first partition the application into two parts, one running on the RISC processor and the other running on the RCM. The result of partitioning is two sets of code segments written in C. For the code segments for the RISC processor, we statically schedule them and generate assembly code with a conventional compiler. For the code segments for the RCM (generally loop kernels), we generate a control data flow graph (CDFG) using the SUIF2 [12] parser. During this process, we perform loop unrolling to maximize the utilization of PEs. We then perform HLS to get the schedule and binding information for one column of PEs. Since we have multiple columns, we let each column of the CGRA execute its own iteration of the loop to implement loop pipelining.

5.2 Heuristic Algorithm for Mapping

As shown in Fig. 5, our heuristic mapping algorithm consists of two phases: i) list scheduling to get an initial solution, and ii) quantum-inspired evolutionary algorithm (QEA) [14] to get a more refined solution which is close to the optimum.

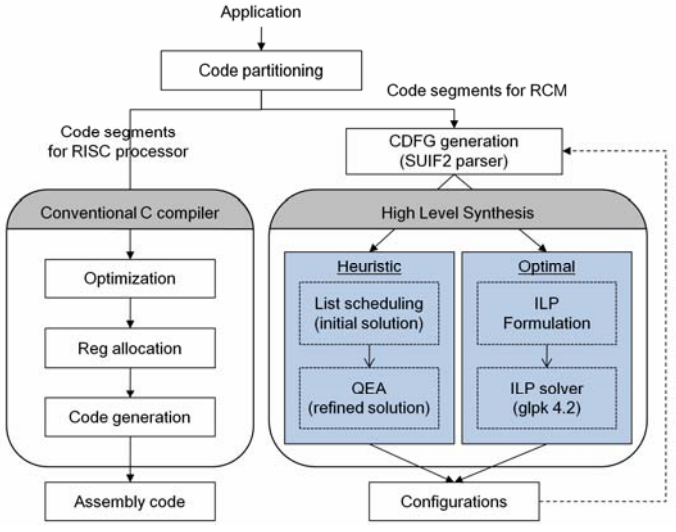


Fig. 5. Overall design flow

List Scheduling. Over the unrolled CDFG, we run list scheduling with the given resource constraint in order to obtain the initial solution (this initial solution is used at the beginning of second phase of QEA). First, we topologically sort the vertices from the sink to the source. Among the vertices in the sorted list, the vertex with the highest priority is scheduled, if all the predecessor vertices are already scheduled and the vertex is reachable from all the scheduled predecessor vertices through the currently available interconnections of the CGRA. When mapping a vertex onto the array of PEs, we consider the following constraints of CGRA: interconnect constraint and shared resource constraint. If there is no direct connection available for implementing a data dependency between two PEs, we try to find a shortest path using the unused remaining PEs as routers. To find the shortest path, we first construct a graph with the remaining PEs (PEs that have not been used yet). The vertices and edges of the graph represent the remaining PEs and their interconnections, respectively. Then we use Dijkstra's algorithm to find a shortest path. Even if we can find a path, we also have to consider the constraint caused by sharing area-critical functional units. For example, if we have only one multiplier shared among the PEs in a row, we cannot use the multiplier again within N (number of PEs in a row) cycles after its use, since the multiplier must be used for pipelining by other columns. In this case, the vertex using the shared resource is scheduled N cycles later with proper routing of data.

QEA. QEA is an evolutionary algorithm that is known to be very efficient compared to other evolutionary algorithms [14]. The fitness function that we use for the QEA is the performance, which is the inverse of the total latency given by the expressions (1) or (2) in the ILP formulation.

The QEA, like genetic algorithms, generates dozens or hundreds of possible cases (through rotation instead of crossover operation) and evaluates each case to choose the best solution. This solution is then used to produce the next generation (refer to

[14] for the details). The iterative improvement continues until it finds a solution which is equal to the result of ASAP scheduling obtained by no resource constraint, or there is no improvement during some time interval. We stop the iteration if it reaches the ASAP solution since it is a lower bound of optimal solution.

We seed the QEA to start from the list scheduling result and attempt to reduce the total latency. Since the QEA starts with a relatively good initial solution, it tends to reach a better solution sooner than starting with a random seed. As a result of QEA, the schedule and binding of each vertex are determined. Once the schedule and binding are obtained, it tries to find the routing path among the vertices with unused remaining PEs to see if these schedule and binding results violate the interconnect constraint. For the routing, we order the edges to be routed since the previous routing may affect the later routing. The ordering is determined as follows.

- Edges located in the critical path get higher priority.
- Among the edges located in the critical path, edges that have smaller slack (shorter distance) get higher priority.
- If a set of edges have the same tail vertex, then the set of edges becomes a group and the priority of this group is determined by the highest priority among the group members. This grouping is necessary for finding a Steiner tree.

According to the above priority, we make a list of candidate edges and find a shortest path for each edge in that order. For finding a Steiner tree, we try to find a path for each outgoing edge individually, and if some paths use the same routing PE, it becomes a Steiner point.

Although this approach may not always find an optimal path, it gives more chances to find a better solution. Indeed, our experimental results in the next section show that our approach finds optimal solutions for 96 percent of the randomly generated examples on the average.

6 Experiments

6.1 Experimental Setup

To demonstrate the effectiveness of our approach, we perform three different experiments: i) experiment with an illustrative example to show that Steiner tree gives a better solution than spanning tree, ii) experiment with a random set of CDFG examples, and iii) experiment with a collection of standard benchmarks from the Livermore loops, Mediabench, and DSPStone benchmark suites.

For the first experiment, we consider mesh connectivity (lack of routing resources better reveals the effect of considering Steiner tree) and for the rest, we consider mesh-plus connectivity. In the mesh-plus connectivity of 4 PEs in a column, interconnects of one hop distance are added to the mesh interconnect. In the mesh-plus connectivity of 8 PEs in a column, left 4 PEs (or right 4 PEs) are fully connected, and pair-wise interconnect is added between the two groups of 4 PEs. Especially for the second experiment, we have devised a random kernel CDFG generator. Our CDFG generator randomly generates 100 CDFGs for each value of node cardinalities from 5 to 20 nodes (1600 in total). We assume the number of PEs in a column is 4 for

randomly generated CDFGs containing 5 to 15 nodes. And for the case of 16 to 20 nodes, we assume the number of PEs in a column to be 8. All experiments are done on Pentium4 2.4GHz dual processor machine with 4GB RAM. We use glpk4.2 [15] for solving the ILP formulation.

6.2 Experimental Result

Illustrative Example. Table 1 shows the experimental result of the example shown in Fig. 3. We assume 4 PEs in a column which has mesh connectivity. As expected, the approach using Steiner tree gives better solution than using spanning tree. Both ILP and heuristic approaches give optimal solutions. However, the heuristic approach is three orders of magnitude faster than the ILP approach.

Randomly Generated CDFGs. Fig. 6 shows the experimental result of 100 randomly generated examples for each problem size. The X-axis represents the number of nodes that each input application contains, and the left Y-axis shows the log scaled average mapping time and right Y-axis shows the percentage finding optimal solution with the '*Heuristic*' approach for 100 examples. Since '*ILP*' takes a lot of time for large input graphs, we stop the ILP solver after 900 seconds. '*ILP*' is unable to find a solution for the case of 8 PEs in a column within 10 hours, thus we set the optimal solution to be equal to the ASAP result. Note that the ASAP result is a lower bound of the optimal solution. The '*Heuristic*' approach is about three orders of magnitude faster than

Table 1. Experimental result of spanning tree vs. Steiner tree.

		Latency (cycles)	Mapping time (secs)
ILP	Spanning tree	5	1022
	Steiner tree	4	965
Heuristic	Spanning tree	5	≤ 1
	Steiner tree	4	≤ 1

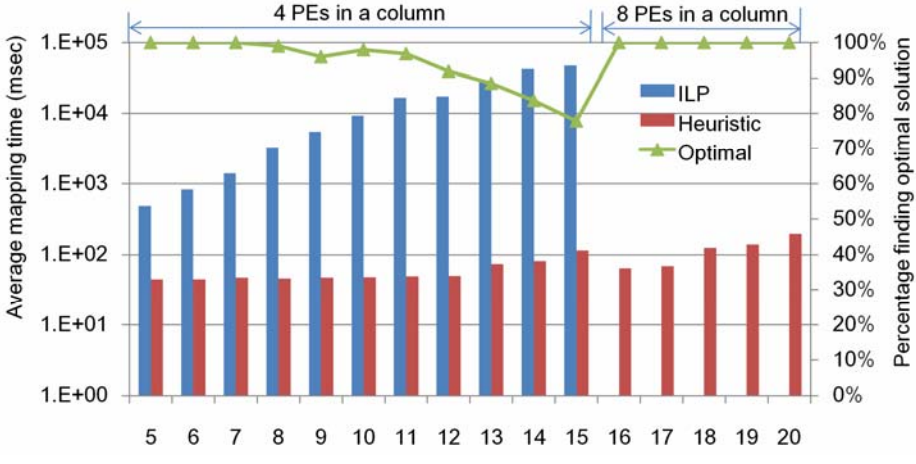


Fig. 6. Experimental result of random examples

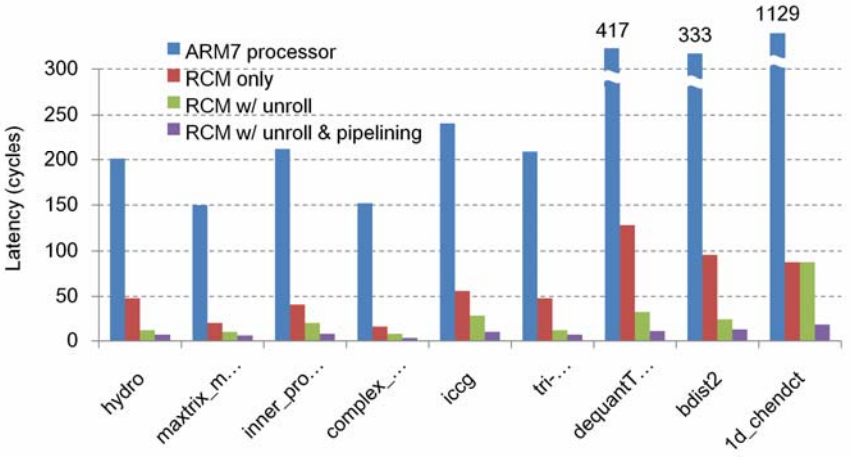


Fig. 7. Latency comparison of standard benchmarks

Table 2. Characteristics and experimental result of benchmark examples

	# of nodes	# of iterations	Unroll factor	RCM w/ Spanning (cycles)	RCM w/ Steiner (cycles)	Optimal (cycles)
hydro	5	8	4	7	7	7
matrix_mult_4x4	7	4	2	6	6	6
inner_product	7	8	2	8	8	8
complex_update	8	4	2	5	5	5
iccg	8	8	2	13	13	13
tri-diagonal elimination	9	8	4	13	13	13
dequantType1 (mpeg4_dec)	10	16	4	15	15	15
bdist2 (mpeg2_enc)	16	8	4	13	13	13
1d_chendct (jpeg_enc)	60	8	1	19	18	18

the '*ILP*' approach, still finding optimal solutions for 96 percent on average. Even in the case of non-optimal solutions, the difference from the optimal solution is just one cycle on average. For the randomly generated test examples, our approach using Steiner point routing gives 10 percent better performance result on average than the one using spanning tree routing.

Standard Benchmark Suites. We experiment with a set of benchmarks collected from the Livermore loops, Mediabench, and DSPStone benchmark suites. We assume 8 PEs in a column with mesh-plus connectivity. Fig. 7 shows the total latency values obtained on the RCM by our approach together with those obtained by software implementation on an ARM7 processor. Table 2 shows their characteristics and comparison between the approaches using spanning tree and Steiner tree. For the small sized benchmark examples (number of nodes ranges from 5 to 16), there is no difference between them. However, as the application size increases (*1d_chendct* example has 60 nodes) routing problem becomes more important. Currently we are working on applying our algorithm to rather big sized practical examples.

7 Conclusion

In this paper, we present approaches to the routing-aware mapping algorithm considering Steiner points for coarse-grained reconfigurable architecture. After outlining an optimal integer linear programming formulation, we present a fast heuristic based on high-level synthesis techniques that uses loop unrolling and pipelining techniques to generate loop parallelized configurations. Experimental results on randomly generated test examples show that our proposed approaches considering Steiner points, give 10 percent better performance than the one using spanning tree. Furthermore our heuristic algorithm finds an optimal solution for 96 percent on average within a few seconds. Experiments on benchmarks also convey similar results.

Acknowledgments. This work was supported by KOSEF under NRL Program Grant (R0A-2008-000-20126-0) funded by MEST, IITA under ITRC support program (IITA-2008-C1090-0804-0009) funded by MKE and Nano IP/SoC Promotion Group under Seoul R&BD Program (10560).

References

1. Chameleon Systems Inc., http://www.chameleonsystems.com
2. Callahan, T.J., Wawrzynek, J.: Instruction-level parallelism for reconfigurable computing. In: Hartenstein, R.W., Keevallik, A. (eds.) FPL 1998. LNCS, vol. 1482, pp. 248–257. Springer, Heidelberg (1998)
3. Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., Amarasinghe, S.P.: Space-time scheduling of instruction level parallelism on a RAW machine. In: Proc. ASPLOSV (1998)
4. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In: Proc. ICFPT (2002)
5. Toi, T., Nakamura, N., Kato, Y., Awashima, T., Wakabayashi, K., Jing, L.: High-level synthesis challenges and solutions for a dynamically reconfigurable processor. In: Proc. ICCAD (2006)
6. Park, H., Fan, K., Mahlke, S.A., Oh, T., Kim, H., Kim, H.: Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: Proc. PACT (2008)
7. Sutter, B.D., Coene, P., Aa, T.V., Mei, B.: Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In: Proc. LCTES (2008)
8. Yoon, J., Shrivastava, A., Park, S., Ahn, M., Paek, Y.: A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architecture. IEEE Trans. Very Large Scale Integration Systems 10 (June 2008)
9. Lee, G., Lee, S., Choi, K.: Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques. In: Proc. ISOCC (2008)
10. Kim, Y., Kiemb, M., Park, C., Jung, J., Choi, K.: Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In: Proc. DATE (2005)
11. Kim, Y., Park, I., Choi, K., Paek, Y.: Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In: Proc. ISLPED (2006)

12. The SUIF compiler system, http://suif.stanford.edu
13. Oliver Shields Jr., C.: Area Efficient Layouts of Binary Trees in Grids, Ph.D thesis, the University of Texas at Dallas, USA (2001)
14. Han, K., Kim, J.: Quantum-inspired evolutionary algorithms with a new termination criterion, Hε gate, and two phase scheme. IEEE Trans. Evolutionary Computation 8 (April 2004)
15. GNU Linear Programming Kit, http://www.gnu.org/software/glpk

Design Automation for Reconfigurable Interconnection Networks

Hongbing Fan[1,⋆], Yu-Liang Wu[2,⋆⋆], and Chak-Chung Cheung[3,⋆⋆⋆]

[1] Wilfrid Laurier University, Waterloo, ON Canada N2L 3C5
hfan@wlu.ca
[2] The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
ylw@cse.cuhk.edu.hk
[3] Department of Electronic Engineering, City University of Hong Kong
cccheung@ieee.org

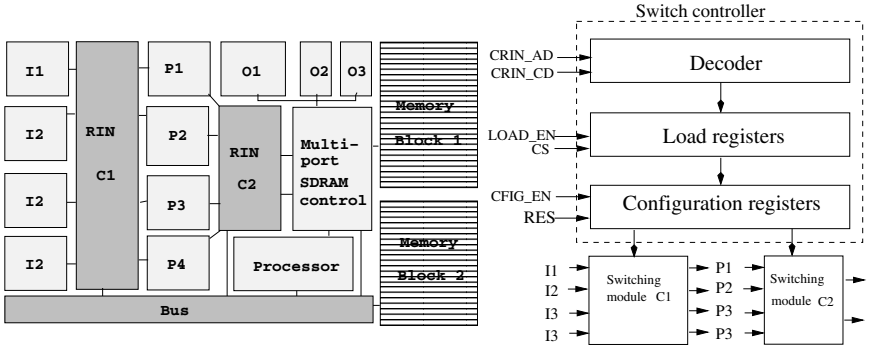
Abstract. A Reconfigurable Interconnection Network (RIN) is a custom designed on-chip switching network yielding routing solutions for a pre-given set of applications. Like FPGA routing networks, the RIN is used to make reconfigurable interconnections among functional blocks. Unlike FPGAs, the network topology of a RIN is irregular as it is designed for a given set of routing requirements and optimized for area, power and delay minimizations. In this paper, we propose an automatic design scheme for RINs, including routing specification formulation, graph modelings, network topology designs, routing algorithms, and MUX-based network circuit implementation. A CAD tool is developed based on the design scheme, which takes a set of routing requirements as input and produces the corresponding RIN network topology and network circuit in HDL format. We present the area costs of various RINs generated by the CAD tool with Altera's Quartus II, and illustrate the RIN design scheme with a reconfigurable multi-stream video system prototype.

1 Introduction

Our study on Reconfigurable Interconnection Network (RIN) was motivated by the design of reconfigurable System-on-Chips (SoCs) for a pre-specified set of applications. The system can be reconfigured to run one application at one time, and another application at another time, or several applications simultaneously. Each application requires a sub-group of available Functional Blocks (FB), which are connected together by a specific interconnection pattern (a routing requirement) determined by the application. We use a RIN as a reconfigurable block to make the various interconnections among the FBs. Since the RIN is used to do the routing for a given set of applications, it can be optimized for resource usage.

⋆ Research partially supported by the NSERC, Canada.
⋆⋆ Research partially supported by RGC Earmarked Grant 2150500 and ITSP Grant 6902308, Hong Kong.
⋆⋆⋆ Research partially supported by the Croucher Foundation, Hong Kong.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 244–256, 2010.



(a) Reconfigurable multi−stream vedio system

(b) Components of RIN.

Fig. 1. RIN and its application in reconfigurable system

This leads us to consider the general RIN design problem. That is, given a set of routing requirements, to design a switching network such that it is guaranteed to route each of the given routing requirements, meanwhile having a low cost on area and power and delay.

Fig.1(a) shows the application of RINs in a reconfigurable multi-stream video system, where the RIN is used to switch stream-video signals going through different processing units. Fig.1(b) shows the block diagram of the RIN, in which the switching module consists of wires and switches that connect input signal ports to output signal ports. The ON/OFF of switches are controlled by control bits. Control bits are stored in the configuration registers, which are loaded through load registers. The load registers get values through address and configuration data path.

The idea of using RINs as a reconfigurable component in a reconfigurable system has been previously discussed. For instance, RINs have been used in Built-In-Self-Testing (BIST) design [9] and fault-tolerance design. In these application instances, a RIN is an application specific functional block satisfying a given Routing Specification (RS). After the design step for the given specification, it is added into the host design, followed by the standard IC design flow for floor plan and routing. On the other hand, the routing networks in FPGAs are reconfigurable interconnection networks. However, FPGA routing networks are designed to do routing for genetic functions and always have regular topology styles such as mesh, row, or tree [8]. A RIN is customarily designed for a given set of routing requirements and often comes with an irregular topology. For on-chip communications, a RIN provides dedicated interconnections among FBs, so it is suitable for stream data communication with low clock frequency.

Even though RINs have been used in many applications, there are no systematic design scheme and design automation tool available. In this paper, we propose a design scheme together with a CAD tool for the design automation of RINs. The scheme includes both hardware and software. The hardware part involves routing specification, network topology design and network circuit

design. The software part deals with the topology generation, circuit generation and routing. The CAD tool takes a set of routing requirements as input and produces the corresponding RIN topology and circuit in HDL format.

The rest of this paper is organized as follows. Section 2 gives the formulations of RS and RIN design problem. Section 3 presents the network topology designs for directed RINs, including 1-stage crossbars, multistage Clos networks and our generalized Clos networks. Section 4 presents the topology designs for undirected RINs. In Section 5, we first give the RIN design automation flow together with the corresponding CAD tools, and then the experimental results done by Altera Quartus II on a variety of RIN circuits, followed by an illustration of applying the tool to generate a RIN for the application of a multi-stream video system.

2 RIN Design Specifications

This section gives the general formulations for routing requirements, routing specification, and the RIN design problem.

2.1 Routing Specifications

The function of a RIN is to make required interconnections of the ports of FBs according to application specifications. The set of FBs contains all the fundamental blocks to be used in the anticipated applications. Each FB contains input/output/inout pins. An application uses a subset of the FBs with specific interconnections of their pins. The term *net* refers to an interconnection request of two or more pins. A net is a 2-pin net (or point-to-point connection) if it connects an output pin (source) to an input pin (sink), and multi-pin net (or one-to-many connection) if it connects an output pin to two or more input pins. An inout pin can function either as an input pin or output pin depending on how it is used. A realization (or routing) of a net is a physical connection of the pins by wires and ON switches. An application corresponds to a set of disjoint nets, which is referred to as a *routing requirement*.

In a traditional ASIC design, there is usually only one routing requirement dedicated for one application. The routing problem is to determine how to layout the wires to realize all the nets of the routing requirement. When multiple application design is considered, a super set of FBs and a set of routing requirements over the pins are given. We consider using a RIN to realize the given routing requirements after the system is fabricated and when the system is in use. As a functional block, the input/ouput/inout pins of the RIN will be connected to the output/input/inout pins of other function blocks. To avoid confusion, we use the term input/output/inout port to refer to the input/output/inout pin when designing a RIN. Then the sets of input, output and inout ports, and the set of routing requirements over the ports forms the routing specification (RS) for RIN design. It can be formulated as follows.

Let A, B and U denote the sets of input, output and inout ports respectively. A net N (or $(t+1)$-pin net) connecting an input/inout port $p_0 \in A \cup U$ to some output/inout ports $p_1, \ldots, p_t \in B \cup U$ is represented as $N = \{p_0, p_1, \ldots, p_t\}$.

A routing requirement consists of a set of disjoint nets, represented as $R = \{N_i : i = 1, \ldots, k\}$, where each N_i is a net and $N_i \cap N_j = \emptyset$ if $i \neq j$. A set of routing requirements is represented as $C = \{R_1, \ldots, R_s\}$, where each $R_j = \{N_{j_i} : i = 1, \ldots, k_j\}$ is a routing requirement for $j = 1, \ldots, s$. Then the RS is formulated as the quadruple (A, B, U, C).

A RIN satisfying RS (A, B, U, C) is a module consisting of wires and switches connecting ports of A, B, U and satisfying that, for each routing requirement $R \in C$, there is ON/OFF assignments of the switches so that all nets in R are realized and two different nets are disjoint.

2.2 RS Simplification

The RS (A, B, U, C) is derived directly from the output, input, and inout ports of other FBs and the set of routing requirements associated with the anticipated applications. It can be simplified without affecting the design of RIN. For example, if a net is contained in every routing requirement, then a fixed routing of the net will be satisfying the requirement. Therefore, the net can be removed from RS. It is proved that the following reduction rules can be used for simplification.

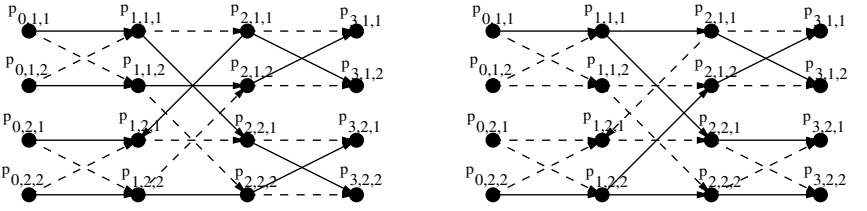
R1. If $a \in A \cup B \cup U$ is not contained in any net, remove a from $A \cup B \cup U$.
R2. Remove all nets N such that $|N| = 1$.
R3. If a net N is contained in every routing requirement of C, then remove N from all the requirements.
R4. If $\{a, a'\}$ is the only net containing a, then remove net $\{a, a'\}$ from C. In this case, a switches joining a to a' must be added to RIN.

Repeatedly applying the above rules to the resulting routing specification until no further reductions can be done, we obtain a reduced RS (A, B, U, C). Further more, we decompose (A, B, U, C) into minimal RSs $(A_i, B_i, U_i, C_i), i = 1, \ldots, k$, where $A_i, B_i, U_i, i = 1, \ldots, k$ are mutually disjoint and a minimal RS is one which can not be decomposed further. We need only consider the problem of designing a RIN for a minimal RS because the RIN for non-minimal RS is the union of RINs of all its minimal RSs.

We only consider the design for two types of RSs, directed and undirected. A directed RS contains no inout ports, i.e., $U = \emptyset$, denoted by (A, B, C). We use directed RINs consisting of wires and unidirectional switches for directed RS. An undirected RS only contains inouts ports, i.e., $A = B = \emptyset$, denoted by (U, C), and we use undirected RIN consisting of wires and bidirectional switches for undirected RS. If $A \cup B \neq \emptyset$ and $U \neq \emptyset$, we treat all ports as inout ports and design an undirected RIN accordingly.

3 Design for Directed RINs

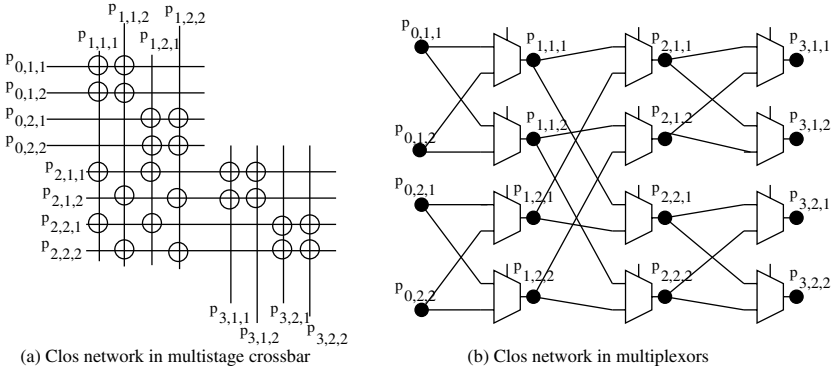
In this section, we propose the topology design and circuit implementation for directed RS (A, B, C). We first give the graph modeling for directed RINs and MUX-based implementation, then propose four candidate topology designs in the tradeoff of different resource costs.



(a) Routing of a 2−pin net routing requirement

(b) Routing of a multi−pin net routing requirement

Fig. 2. Routings of 2-pin and multi-pin routing requirements.



(a) Clos network in multistage crossbar

(b) Clos network in multiplexors

Fig. 3. Implementations of multistage switching networks

3.1 Graph Modeling and Implementations

We model a directed RIN for (A, B, C) by a multistage digraph. A k-stage digraph is a digraph $G = (V_0 \cup \cdots \cup V_k, E)$, where $V_0, \ldots, V_k$ are disjoint node sets, and each edge (or arc) in E is from a node in V_i to a node in V_j with $j > i$. If every edge is joining a node of V_i to a node of V_{i+1}, we call it k-stage bipartite digraph. Particularly, $A = V_0$ are input nodes and $B = V_k$ are output nodes, and nodes in $V_i, 1 \leq i \leq k-1$ are intermediate nodes.

The graph modeling provides an effective way to do routing in the network. The routing of a 2-pin net is a path from the input node to the output node, and the routing of a mult-pin net is a tree with the input node as root and output nodes as leaves. For example, Fig.2(a) shows the 3-stage $(4, 4)$-Clos network and the routing of 2-pin routing requirement $\{\{p_{0,1,1}, p_{3,2,2}\}, \{p_{0,1,2}, p_{3,1,1}\}, \{p_{0,2,1}, p_{3,1,2}\}, \{p_{0,2,2}, p_{3,2,1}\}\}$, and Fig.2(b) the routing of multi-pin routing requirement $\{\{p_{0,1,1}, p_{3,1,2}, p_{3,2,1}\}, \{p_{0,2,2}, p_{3,1,1}, p_{3,2,2}\}\}$. We say a graph G is a topology candidate for the given RS (A, B, C) if G contains a routing for every routing requirement of C.

The graph modeling is effective for the circuit implementation. It can be implemented by a multi-stage crossbar, as shown in Fig.3(a), in which an edge

corresponds to a crosspoint switch and a node corresponds to a wire (either vertical or horizontal). It can also be implemented by a multiplexer network, as shown in Fig.3(b), in which an edge corresponds to a wire and a non-input node corresponds to a multiplexer. We use multiplexer network as the circuit implementation in our design scheme and CAD tool.

3.2 One-Stage Networks

Given a directed RS (A, B, C). We define the *interconnection digraph*, denoted by $IG(A, B, C)$, of (A, B, C) as bipartite digraph (A, B, E), where $(u, v) \in E$ if $u \in A, v \in B$ and u, v are contained in a net of a routing requirement of C. That is, we add an edge joining node u of A to node v of B provided u is connected to v in some of the given routing requirement. The interconnection digraph $IG(A, B, C)$ of (A, B, C) is a topology candidate for (A, B, C). This is because, for any net $N = \{p_0, \ldots, p_t\}$ of a given routing requirement, the routing of N in G can always be done by the edges $(p_0, p_1), \ldots, (p_0, p_t)$. This indicates that $IG(A, B, C)$ is nonblocking for any of the given routing requirements and routing can be done in linear time. Clearly, $IG(A, B, C)$ is a 1-stage switching network (or a partial crossbar), and contains no intermediate nodes. Hence, $IG(A, B, C)$ is the best topology candidate for (A, B, C) in terms of switch delay and intermediate nodes and routing algorithm.

There are situations that we want to do routing for new applications other than the pre-given ones after the RIN is designed. A new application must use a subset of the given FBs, and the interconnection is likely among the FBs which are feasible. We determine a feasible interconnection by checking if a similar type of connection has been given in the pre-given routing requirements. Specifically, if an input node u has a connection to an output node v, and input node x has connection to an output nodes v and y in the pre-given routing requirements, then it is possible to connect u to y in a new application. So, we add edge (u, y) to improve the routing capacity. As a result of repeatedly applying this operation, a complete bipartite graph (or full crossbar) over (A, B) is derived, we call it the *compete interconnection digraph* of (A, B, C), and denote it by $\overline{IG}(A, B, C)$. We see that $\overline{IG}(A, B, C)$ has the capacity of making all possible interconnections of the input, output ports. Therefore, it is the best solution in terms of routing capacity, delay, intermediate nodes and routing algorithm.

3.3 Multistage Clos Networks

Both $IG(A, B, C)$ and $\overline{IG}(A, B, C)$ could be inefficient in terms of switch cost. For instance, when $\overline{IG}(A, B, C)$ is an (N, N)-network, i.e., $|A| = |B| = N$, the switch cost of $\overline{IG}(A, B, C)$ is $O(N^2)$, so $\overline{IG}(A, B, C)$ may not fit in the host design when N is large. In this situation, we use unicast (for 2-pin net) or multicast (for multi-pin net) rearrangeable multistage switching networks to expand the dense subgraphs so as to reduce the switch cost. This is a tradeoff between the switch cost and the costs on delay and intermediate nodes as increasing the number of stages will increase both the switch delay and the number of intermediate nodes.

The tradeoff can significantly reduce the overall area cost as well as the power cost due to the reduction of the number of switches.

Multistage switching networks have been studied extensively in the fields of communication networks and parallel/distribuited computing since the early work of Clos [2]. Various routing capabilities such as strictly/wide-sense nonblocking, rearrangable, unicast, multicast, and broadcast, have been proposed. Huang [7] gave a comprehensive coverage on the mathematical theory of switching networks. Dolev et al. [3] proved that, theoretically, a k-stage rearrangeable broadcast (N, N)-network (of N inputs and N outputs) has a lower bound of $O(N^{1+1/k})$ switches and an upper bound of $O((N \log N)^{1+1/k})$ switches. Practically, Clos networks are often used due to its scalable routing capacities, simple structure and routing algorithms. We use 3-stage Clos network, which can be represented as a multistage bigraph as follows.

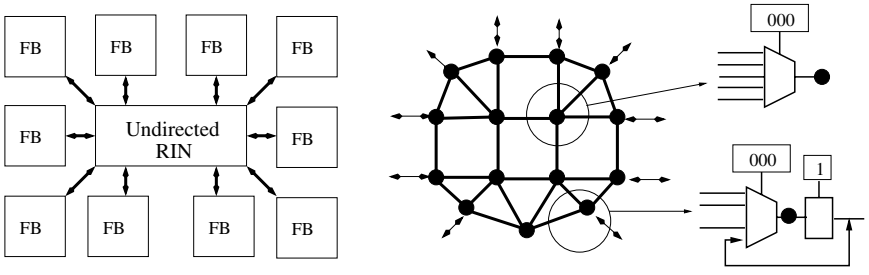
$$\begin{aligned} C(n_1, r_1, m, n_2, r_2) &= (V_0 \cup V_1 \cup V_2 \cup V_3, E_1 \cup E_2 \cup E_3), \\ V_0 &= \{p_{0,j,s} : j = 1, \ldots, r_1, s = 1, \ldots, n_1\}, \\ V_1 &= \{p_{1,j,s} : j = 1, \ldots, r_1, s = 1, \ldots, m\}, \\ V_2 &= \{p_{2,j,s} : j = 1, \ldots, r_2, s = 1, \ldots, m\}, \\ V_3 &= \{p_{3,j,s} : j = 1, \ldots, r_2, s = 1, \ldots, n_2\}, \\ E_1 &= \{(p_{0,j,s}, p_{1,j,t}) : 1 \le s \le n_1, 1 \le t \le m, 1 \le j \le r_1\}, \\ E_2 &= \{(p_{1,j,t}, p_{2,h,t}) : 1 \le t \le m, 1 \le j \le r_1, 1 \le h \le r_2\}, \\ E_3 &= \{(p_{2,j,s}, p_{3,j,t}) : 1 \le s \le n_2, 1 \le t \le m, 1 \le j \le r_2\}. \end{aligned} \tag{1}$$

When the RS (A, B, C) contains only 2-pin nets, we use unicast rearrangeable switching networks. It was known that a 3-stage Clos network $C(n_1, r_1, m, n_2, r_2)$ is unicast rearrangeable if $m \ge \max\{n_1, n_2\}$ and routing can be done efficiently. Particularly, the symmetric 3-stage Clos network $C(n, r, m, n, r)$ is unicast rearrangeable when $m \ge n$. Moreover, by setting $m = n = r = N^{1/2}$, the $C(N^{1/2}, N^{1/2}, N^{1/2})$ is unicast rearrangeable with $3N^{3/2}$ switches and the minimum $2N$ intermediate nodes. Therefore, we use the unicast 3-stage Clos network $C(n_1, r_1, \max\{n_1, n_2\}, n_2, r_2)$ as the next network topology candidate, followed by the 5-stage Clos network candidate, and so on. Note that when expanding $\overline{IG}(A, B, C)$ into Clos network $C(n_1, r_1, m, n_2, r_2)$, we calculate the values for n_1, r_1, m, n_2, r_2 so that the resulting Clos network satisfies the required RS and has the minimum number of switches.

When the RS (A, B, C) contain multi-pin nets, we use multicast or broadcast rearrangeable switching networks. It was known [11] that $C(n_1, r_1, m, n_2, r_2)$ is broadcast rearrangeable and has a linear time routing algorithm when $m \ge 2(n_1 - 1)(\log r_2 / \log\log r_2) + (\log r_1)^{1/2}(n_2 - 1) + 1$. By setting $r_1 = n_1 = r_1 = n_2 = m = O(N^{1/2})$, the derived N-network is broadcast rearrangeable with switch cost $O(N^{3/2} \log N / \log\log N)$. Therefore, with multi-pin RS, we use $C(n_1, r_1, m, n_2, r_2)$ with $m = 2(n_1 - 1)(\log r_2 / \log\log r_2) + (\log r_1)^{1/2}(n_2 - 1) + 1$ as the 3-stage topology candidate.

3.4 Generalized Clos Networks

The multicast rearrangeable Clos networks could be bad on node cost. For example, the number of intermediate nodes in the broadcost 3-stage (N, N)-Clos



(a) Undirected RIN in a reconfigurable system

(b) Graph modeling of undirected RIN and mux implementation of nodes

Fig. 4. The graph representation of undirected RIN

network is $O(N \log N / \log \log N)$. The intermediate node will increase fast when the stage number increase. To avoid the cost explosion of the intermediate nodes, we propose a $(2k+1)$-stage rearrangeable multicast switching network which uses the minimum number of $2kN$ intermediate nodes with a tradeoff of switch cost.

The idea is to add more switches to the middle stage of the unicast Clos network so as to increase the routing capacity. It was known that if the middle stage becomes hyperuniversal [5], then the resulting the 3-stage network is multicast rearrangeable. For any non-negative integer $0 \leq w \leq m$, define:

$$\begin{aligned} E_2(w) &= \{(p_{1,j,h}, p_{2,j',h'}) : |h - h'| \leq w\}, \\ C_w(n_1, r_1, m, n_2, r_2) &= C(n_1, r_1, m, n_2, r_2) + E_2(w). \end{aligned} \tag{2}$$

It was known that when $w = 3r_1 + 3r_2$, $C_w(n_1, r_1, \max\{n_1, n_2\}, n_2, r_2)$ is multicast rearrangeable with the minimum number of intermediate nodes and efficient routing algorithm [6]. Therefore, we choose $C_{3r_1+3r_2}(n_1, r_1, \max\{n_1, n_2\}, n_2, r_2)$ as an alternative topology candidate. Since the middle stage of $C_w(n_1, r_1, \max\{n_1, n_2\}, n_2, r_2)$ contains $\max\{n_1, n_2\}/w$ disjoint (wr_1, wr_2)-full crossbars, we can continue to expand these full crossbars into 3-stage networks to derive a 5-stage multicast rearrangeable (N, N)-network as the next candidate for RIN.

4 Design for Undirected RINs

This section presents candidate topology designs for undirected RS (U, C). The design scheme is similar to that of directed RS.

4.1 Interconnection Graphs

We model an undirected RIN by a simple graph $G = (U \cap V, E)$, where U is the set of inout nodes, V the set of intermediate nodes, and E the set of edges representing switches. Fig.4(a) illustrates a reconfigurable system with an undirected RIN, and (b) the graph representation and MUX implementation. The main problem is to find a topology candidate for (U, C) with a small number

of edges and intermediate nodes as well as short delays. Similar to directed RIN design, we first consider the topology candidates that use no intermediate nodes, and then multistage network solutions.

Given RS (U, C). We call graph $G = (U, E)$ an *interconnection graph* of (U, C) if, for any net N of C, the induced subgraph $G[N]$ is connected. Then an interconnection graph G of (U, C) is a topology candidate of (U, C) because, for any requirement $R = \{N_i : i = 1, \ldots, k\} \in C$, since $G[N_i]$ is connected, there is a spanning tree T_i of $G[N_i]$ for every $i = 1, \ldots, k$. By assigning ON to all switches on $T_1 + \cdots + T_k$, and OFF to other switches of G, thus we obtain a routing for R on G in a linear time. In addition, G is a candidate without intermediate nodes. We use $IG(U, C)$ to denote an interconnection graph of (U, C).

The existence of interconnection graphs is obvious as the complete graph over U, denoted by $\overline{IG}(U, C)$, is an interconnection graph of (U, C). Since $\overline{IG}(U, C)$ has the full routing capacity, so we use $\overline{IG}(U, C)$ as the first candidate. The number of edges in $\overline{IG}(U, C)$ is $O(N^2)$, it may not fit in the host design, so next we try to find an $IG(U, C)$ with the minimum number of edges. Since the problem of finding a minimal interconnection graph is NP-hard [4], we use the greedy algorithm proposed in [4] to find an $IG(U, C)$, and use it as the second candidate.

4.2 Multistage Switching Networks

When both $\overline{IG}(U, C)$ and $IG(U, C)$ have a high cost on switches, we then consider using the 1-sided 3-stage Clos network, which is defined as follows.

$$\begin{aligned} PSN(n, m, r) &= (U_1 \cup \cdots \cup U_r \cup V_1 \cup \cdots \cup V_r, E_1 \cup \cdots \cup E_r \cup E_c) \\ U_j &= \{u_{j,h} : 1 \le h \le n\}, j = 1, \ldots, r, \\ V_j &= \{v_{j,h} : 1 \le h \le m\}, j = 1, \ldots, r, \\ E_j &= \{u_{j,h}v_{j,h'} : 1 \le h \le n, 1 \le h' \le m\}, j = 1, \ldots, r, \\ E_c &= \{v_{j,h}v_{j',h} : 1 \le j, j' \le r, 1 \le h \le m\}, \end{aligned} \tag{3}$$

where $U = U_1 \cup \cdots \cup U_r$ is the set of inout nodes, and $V = V_1 \cup \cdots \cup V_r$ is the set of intermediate nodes. The middle stage (V, E_c) of $PSN(n, m, r)$ forms an (r, m)-switch box. It was known that $PSN(n, m, r)$ is unicast (multicast) rearrangeable [6] if the middle switch box is universal [1] (hyperuniversal [5]).

The routing on $PSN(n, m, r)$ can be done efficiently by three steps. Step 1 induces the routing requirement to a routing requirement on the center switch box, step 2 finds a routing of the routing requirement on the switch box, and step 3 extends the switch box routing to the inout stage. For example, Fig. 5(a) depicts the $PSN(4, 4, 4)$. Let $R = \{N_1, \ldots, N_7\}$ be a routing requirement for $PSN(4, 4, 4)$, $N_1 = \{u_{1,1}, u_{2,1}\}, N_2 = \{u_{1,2}, u_{2,2}, u_{4,3}\}, N_3 = \{u_{1,4}, u_{4,4}\}, N_4 = \{u_{4,2}, u_{2,3}, u_{3,3}\}$, $N_5 = \{u_{1,3}, u_{3,2}\}, N_6 = \{u_{3,1}, u_{2,4}\}$, $N_7 = \{u_{4,1}, u_{3,4}\}$. Then R induces the routing requirement $R' = \{R'_1, \ldots, R'_7\}$ on the center $(4, 4)$-SB, where $R'_1 = \{1, 2\}, R'_2 = \{1, 2, 4\}, R'_3 = \{1, 4\}, R'_4 = \{4, 2, 3\}, R'_5 = \{1, 3\}, R'_6 = \{3, 2\}, R'_7 = \{4, 3\}$. Fig.5(b) shows the routing of R' on the center switch box, and (c) the routing of R on $PSN(4, 4, 4)$ extended from the routing of R'.

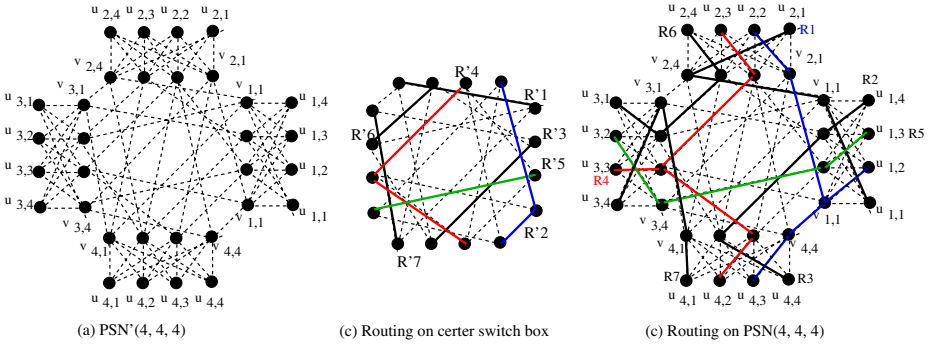


Fig. 5. One-sided 3-stage Clos network and routing

Therefore, for 2-pin net RS, we use a universal switch box (USB) designs given in [10] to substitute the center switch of $PSN(n, m, r)$ to obtain a rearrangeable unicast switching network [12].

For multi-pin net RS, we use the following generalized Clos network. For any integer w with $0 \leq w \leq m$, define:

$$E_w(m,r) = \{v_{j,h}v_{j',h'} : 1 \leq j, j' \leq r, j \neq j'; 1 \leq h, h' \leq m, |h - h'| \leq w\},$$
$$PSN_w(n,m,r) = PSN(n,m,r) + E_w(m,r). \tag{4}$$

It was known that when $w = 3r$, $PSN_w(n, m, r)$ is broadcast rearrangeable with switch cost $O(mr^3)$, and the minimum intermediate node cost $O(mr)$, and an $O(m^2r^5)$ time routing algorithm [6]. Therefore, we use $PSN_w(n, m, r)$ for $w = 1, 2, 3$ as the topology candidates.

5 CAD Tools for RIN Generation

We developed CAD tools for the design automation of RIN based on the design scheme proposed above. The tool consists of following programs.

1. RS extractor: to determine RS (A, B, U, C).
2. RS simplifier: to produce a set of reduced minimal RSs.
3. RIN topology generator: given directed RS (A, B, U, C), to generate a sequence of topology candidates.
4. RIN circuit generator: given an RIN topology graph (either directed or undirected), to generate the circuit in Verilog HDL format.
5. Control circuit generator: to generate control circuits for RINs.
6. Router: to find routing for a given routing requirement.
7. Configuration bit generator: to convert the routing to configuration bit stream.

Fig.6 depicts the RIN automatic design flow with the tool. First of all, it extracts the RS (A, B, U, C). Second, depending on (A, B, U, C), choose to use directed

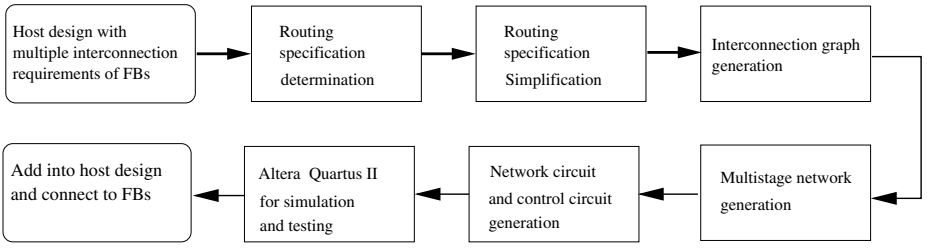


Fig. 6. RIN design automation flow

RIN or undirected RIN. The simple criteria is, if there is not inout in the RS, we use directed RIN; otherwise use undirected RIN. In either case, we use the RIN topology generator to generate a sequence of digraphs (or simple graphs) as topology candidates. Third, for each of the candidates in the sequence, use the RIN circuit generator to generate multiplexer modules for each node of the same degree, then the switching modules using the multiplexors, followed by the switch control module and top level interface modules. After an RIN circuit is generated, we test it with existing CAD tools such as Altera Quartus II. After the testing, we insert it into the host design for system testing. If it is passed, the design cycle is stopped; or otherwise we put the next topology candidate into the design cycle.

To test the RIN topology generator and circuit generator, we first use RIN topology generator to produce a variety of network topology candidates. Then input the topology description file into the RIN circuit generator, which generate the corresponding network circuits. We then compile and simulate the RIN circuits using Altera's Quartus II targeting at Altera's Cyclone FPGA chip. Table 1 shows the experimental results for the directed s-stage (N, N)-RIN (of N input ports and N-output ports) with $N = 8, 32, 128, 512, s = 1, 3, 5$ and s-stage undirected N-RIN (of N inout ports) with $N = 8, 32, 128, 512, s = 1, 3$. All the experimented RINs have data width one. The logic unit number column shows the number of logic units used by the RIN. The decrease percentage is derived by comparing with the corresponding 1-stage network. The number of logic units reflects the area cost. We see that when N is large, increasing the number of stages will significantly reduce the number of logic unit usage.

To verify the application of RINs, we use the tool to generate a RIN for the application of reconfigurable multi-stream video system. After it is generated and passed testing, we add it into the application design and connect the ports of the RIN to the ports of existing modules with the Altera's Quartus II FPGA design KDE. The design has four video inputs and two video output, there are four pre-given applications, each with the video signals going to different processing modules. We store the configuration bits for each application in the memory, and assign call numbers 0, 1, 2, 3 for them respectively. After compiling, we load the design configuration into the Altera FPGA prototyping board. When the system is turned on, we input a call number and then give a configuration enable signal. The system successfully went through the configuration process, which first loads

Table 1. Experimental results on generated directed (N, N)-RINs and undirected N-RINs with Altera Quartus II

		Directed (N, N)-RIN				Undirected N-RIN			
N	stages	switch number	decrease in %	logic unit number	decrease in %	switch number	decrease in %	logic unit number	decrease in %
8	1	64	0%	40	0%	32	0%	52	0%
8	3	64	0%	32	20%	64	-100%	90	-73%
8	5	80	-25%	45	-12%	-	-	-	-
32	1	1024	0%	672	0%	512	0%	546	0%
32	3	512	50%	288	57%	350	32%	661	-21%
32	5	512	50%	304	54%	-	-	-	-
128	1	16384	0%	10880	0%	8192	0%	7258	0%
128	3	4096	75%	2561	76%	1584	80%	3995	45%
128	5	3072	81%	1665	84%	-	-	-	-
512	1	262144	0%	174593	0%	131072	0%	96615	0%
512	3	32768	86%	20993	89%	7392	94%	32383	66%
512	5	19505	92%	12800	92%	-	-	-	-

the corresponding configuration bit stream into load registers associated with each node, then configuration enable signal is automatically applied so that all the control bits are passed from load registers to the configuration registers. After configuration process is done, a feedback signal successfully initiates starting the target application. This experiment proves that the RIN generated by the tool works well with the reconfigurable application system.

6 Conclusions

Application specific reconfigurable interconnection network (RIN) provides an efficient solution for the design of reconfigurable systems. However, the RIN designs in network topology design, circuit generation, and routing is a difficult problem due to variations of routing specifications and irregular network topologies. In this paper, we proposed a RIN design automation scheme and CAD tools, which resolves the RIN topology design by decomposing the network into primitives and then using the existing switching network design results, the circuit design by the automatic generation, and the routing difficulty by a novel topology-aware routing algorithm and configuration bit storage. Our experimental result shows that using RIN for reconfigurable computing can be done effectively with the proposed design automation CAD tools.

References

1. Chang, Y.W., Wong, D.F., Wong, C.K.: Universal Switch Modules for FPGA Design. ACM Transactions on Design Automation of Electronic Systems 1(1), 80–101 (1996)
2. Clos, C.: A Study of Nonblocking Switching Networks. Bell Systems Technical J. 22, 406–424 (1953)

3. Dolev, D., Dwork, C., Pippenger, N., Wigderson, A.: Superconcentrators, Generalizers and Generalized Connectors with Limited Depth (Preliminary Version). In: STOC, pp. 42–51 (1983)
4. Fan, H., Hundt, C., Wu, Y.-L., Ernst, J.: Algorithms and Implementation for Interconnection Graph Problem. In: Yang, B., Du, D.-Z., Wang, C.A. (eds.) COCOA 2008. LNCS, vol. 5165, pp. 201–210. Springer, Heidelberg (2008)
5. Fan, H., Liu, J., Wu, Y.L., Cheung, C.C.: On Optimal Hyper Universal and Rearrageable Switch Box Designs. IEEE Transactions on Computer Aided Designs 22(12), 1637–1649 (2003)
6. Fan, H., Wu, Y.-L.: A New Approach for Rearrangeable Multicast Switching Networks. In: COCOA, pp. 384–394 (2009)
7. Hawang, F.K.: The Mathematical Theory of Nonblocking Switching Networks. Series on Applied Mathematics. World Scientific Publishing Co., Inc., River Edge (2004)
8. Lemieux, G., Lewis, D.: Design of Interconnection Networks for Programmable Logic. Kluwer-Academic Publisher, Boston (2003)
9. Li, L., Chakrabarty, K.: Test set embedding for deterministic bist using a reconfigurable interconnection network. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23, 1289–1305 (2004)
10. Shyu, M., Wu, G.M., Chang, Y.D., Chang, Y.W.: Generic Universal Switch Blocks. IEEE Trans. on Computers, 348–359 (April 2000)
11. Yang, Y., Masson, G.M.: Nonblocking Broadcast Switching Networks. IEEE Trans. Comput. 40(9), 1005–1015 (1991)
12. Yen, M., Chen, S., Lan, S.: A Three-Stage One-Sided Rearrangeable Polygonal Switching Network. IEEE Trans. on Computers 50(11), 1291–1294 (2001)

A Framework for Enabling Fault Tolerance in Reconfigurable Architectures

Kostas Siozios[1], Dimitrios Soudris[1], and Dionisios Pnevmatikatos[2]

[1] School of Elect. & Computer Eng., National Technical University of Athens, Greece
[2] Electronic & Computer Eng. Department, Technical University of Crete, Greece

Abstract. Fault tolerance is a pre-request not only for safety critical systems, but almost for the majority of applications. However, the additional hardware elements impose performance degradation. In this paper we propose a software-supported methodology for protecting reconfigurable architectures against Single Event Upsets (SEUs), even if the target device is not aware about this feature. This methodology initially predicts areas of the target architecture where faults are most possible to occur and then inserts selectively redundancy only there. Based on experimental results, we show that our proposed selectively fault-tolerance results to a better tradeoff between desired level of reliability and area, delay, power overhead.

1 Introduction

SRAM-based Field-Programmable Gate Arrays (FPGAs) are arrays of Configurable Logic Blocks (CLBs) and programmable interconnect resources, surrounded by programmable input/output pads on the periphery. Even though their programmability feature makes them suitable for widely application implementation, a number of design issues have to be tackled. Among others, reliability issue becomes worse as devices have evolved. For instance, as the transistor geometry and core voltages decrease, while the numbers of transistors per chip and the switching frequency increase, the target architectures become more susceptible to incur faults (i.e., flipped bit or a transient within a combinatorial logic path). Consequently, mechanisms that handle faults detection and correction during device operation are required, even for non critical safety systems.

The last ten years many discussions were done about the design of reliable architectures with fault tolerant features. More specifically, the term fault tolerant corresponds to a design which is able to continue operation, possibly at a reduced level, rather than failing completely, when some part of the system fails [1, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14]. These solutions include fabrication process-based techniques (i.e. epitaxial CMOS processes) [10], design-based techniques (i.e. hardware replicas, time redundancy, error detection coding, self-checker techniques) [4], mitigation techniques (i.e. multiple redundancy with voting, error detection and correction coding) [5], and recovery techniques (i.e. reconfiguration scrubbing, partial configuration, rerouting design) [11].

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 257–268, 2010.

Even though fault tolerance is a well known technique, up to now it was mostly studied for ASIC designs. However FPGAs poses new constraints (i.e. higher power density, more logic and interconnection resources, etc), while the existing fault models are not necessarily applicable. To make matters worse, faults in FPGAs can alter the design, not just user data. In addition to that, the FPGA designs utilize only a subset of the fabricated resources, and hence only a subset of the occurred faults may result to faulty operation. Consequently, FPGA-specific mitigation techniques are required, that can provide a reasonable balance among the desired fault prevention, the performance degradation, the power consumption and the area overhead due to the additional hardware.

Up to now there are two approaches for preventing faults occurring on FPGAs. The first of them deals with the design of new hardware elements which are fault tolerant enabled [2, 4, 12, 15]. These resources can either replace existing hardware blocks in FPGAs, or new architectures can be designed to improve robustness. On the other hand, it is possible to use an existing FPGA device and provide the fault tolerance at higher level with CAD tools [2, 3, 4, 8, 13, 14].

Both of these approaches have advantages and disadvantages, which need to be carefully concerned. More specifically, the first approach results to a more complex architecture design, while the derived FPGA provides a static (i.e. defined at design time) fault tolerant mechanism. On the other hand, the implementations belonging on second approach potentially are able to combine the required dependability level, offered by fault tolerant architectures, with the low cost of commodity devices. However, this scenario imposes that the designer is responsible for protecting his/her own design.

In [12] a fault tolerant interconnection structure is discussed, where the faults are corrected by spare routing channels which are not used during place and route (P&R). A similar work is discussed in [13], where a defect map is taken as input to P&R tool and then application's functionalities are not placed in the faulty blocks. In another approach [14], EDA tools take as input a generic defect map (which may be different from the real defect map of the chip) and generate a P&R according to this. A work that deals with a yield enhancement scheme based on the usage of spare interconnect resources in each routing channel in order to tolerate functional faults, is discussed in [15]. The only known commercial approach for supporting fault tolerance in FPGAs can be found in [8]. This implementation inserts two replica blocks for each of the application's logic blocks, which are working in parallel, while the output is derived by comparing their outputs with a majority voting. Table 1 gives a qualitative comparison in terms of supported features for a number of fault tolerant approaches found in relevant references.

In this paper we propose a software supported methodology that improves application's reliability, without inserting excessive amount of redundancy over the entire FPGA architecture. More specifically, we identify sensitive sub-circuits (where faults are most possible to occur) and we apply the proposed fault tolerant technique only at these critical regions rather than inserting redundancy in the

Table 1. Qualitative comparison among fault tolerant approaches

Feature	[8]	[12]	[13]	[14]	[15]	Proposed
Fault tolerant	TMR	Spare routing	Defect map	Defect map	Spare routing	TMR & fault map
Protects	Logic	Routing	Logic	Logic	Routing	Logic
Modifies	HDL	Hardware	HDL	HDL	Hardware	HDL
Applied uniformly	Yes	Yes	No	No	Yes	No
Online fault management	No	No	No	No	No	Yes
Multiple fault tolerant techniques	No	No	Yes	No	No	Yes
Software support	Yes	No	Yes	Yes	No	Yes
Public available	No	No	No	No	No	Yes
Complete framework	Yes	No	No	No	No	Yes

entire device. Such an approach results to better tradeoff between desired fault coverage and area, delay and power consumption.

The main contributions of this paper are summarized, as follows:

1. We introduce a novel methodology for supporting on-line fault detection and correction for FPGA devices.
2. We identify sensitive sub-circuits (where faults are most possible to occur) and we apply the proposed fault tolerant technique only at these points, rather than inserting redundancy in the entire device.
3. We developed a tool that automates the introduction of redundancy into certain portions of an HDL design.
4. The derived application implementations are validated with a new platform simulator.

The rest paper is organized as follows: In section 2 we discuss the motivation of this paper, while section 3 describes the implemented fault tolerant technique. The proposed methodology, as well as its evaluation is discussed in detail in sections 4 and 5, respectively. Finally, conclusions are summarized in section 6.

2 Motivation

The source of errors in ICs can be traced to three main categories: (i) due to internal to the component (i.e. component failure, damage to equipment, cross-talk on wires), (ii) generally external causes (i.e. lightning disturbances, radiation effects, electromagnetic fields), and (iii) either internal or external (i.e. power disturbances, various kinds of electrical noise). Classifying the source of the disturbance is useful in order to minimize its strength, decrease its frequency of occurrence, or change its other characteristics to make it less disturbing to the hardware component.

The first step in order to build a reliable system is to identify possible regions with increased probability of failure. Throughout this paper we study faults

related to power, thermal, as well as random effects. More specifically, the increased switching activity results to higher power consumption and consequently to higher on-chip temperatures. In [17], Black mentioned that the mean time to failure ($MTTF$) of aluminum interconnects exponentially decreases as the temperature (T) of a chip increases. Equation 1 gives the mathematical expression that describes this phenomenon.

$$MTTF \propto \frac{e^{\frac{E_a}{kT}}}{J_{dc}^{n}} \tag{1}$$

where E_a is the enable energy (its value is defined experimentally), J_{dc} denotes the threshold of electromigration current, while n and k are constants. The switching activity of an application is a property which does not depend either to the target platform, or the employed toolset that performs application mapping. However, the employed toolset introduce some constraints regarding the spatial distribution of regions with excessive high (or low) values and consequently with increased (or decreased) probability of failure [18]. Regarding the random faults, they exhibit a distribution of independent (non-correlated) failures.

By combining the spatial variation of these values of the three parameters over the FPGA device, we are able to identify regions with increased failure probability. In order to show that different applications result to different distributions of failure probabilities (even for the same P&R algorithms), Fig.1 plots this variation over an 64×64 FPGA array regarding the *s298* and *frisc* benchmarks, without considering yet any redundancy. In this figure, different colors denote different failure probabilities, while as closer to red color a region is, the higher probability for this region to occur a fault.

Based on these maps, it is evident that the failure probability is not constant across the FPGA or among different applications, since it varies between two arbitrary points (x_1,y_1) and (x_2,y_2) of device. Based on this distribution it is

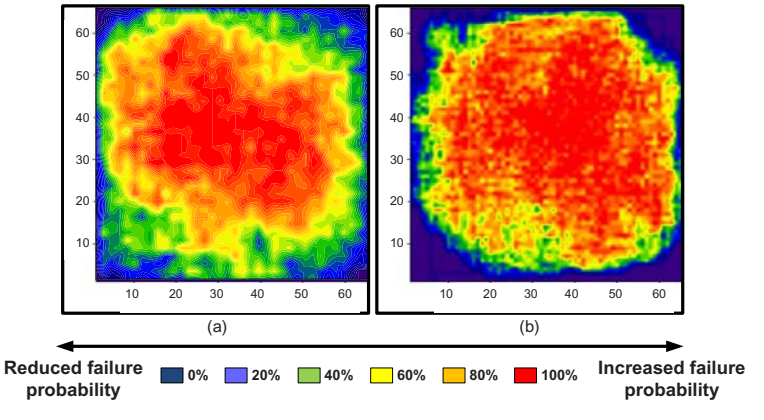


Fig. 1. Spatial distribution of failure probability for (a) *s298* and (b) *frisc* benchmarks

feasible to determine regions on the device with excessive high values of failure probability (regions of importance), where we have to pay effort in order to increase the fault tolerance. Consequently, the challenge, with which a designer is faced up, is to choose only the actually needed redundancy level, considering the associated spatial information from the distribution graph.

A second important conclusion is drawn from Fig.1: although the majority of existing fault tolerant techniques exhibits a homogeneous and regular structure, the actually critical for failure resources provide a $non-homogeneous$ and $irregular$ picture. Consequently, careful analysis of the points of failure must be performed, while the target system implementation needs to combine regions with different density of fault tolerance.

3 Proposed Fault Tolerant Technique

Our target is a generic recent FPGA device similar to the Xilinx Virtex architecture, consisting of an array of configurable logic blocks (CLBs), memories, DSP cores and programmable input and output blocks (placed on its periphery). We assume that the logic blocks are formed by a number of Basic Logic Elements (BLEs), each of which is composed of a set of programmable Look-Up Tables (LUTs), multiplexers, and flip-flops. The communication among these hardware blocks is provided by a hierarchical interconnection network of fast and versatile routing resources. More info regarding the architecture of target FPGA can be found in [19].

In order to provide fault tolerance, we incorporate an R-fold modular redundancy (RMR) technique. Such a mechanism can effectively mask faults if only less than $((R+1)/2)$ replicas are faulty (either on combinational and sequential logic), but the faults present in different register locations and the voter works properly. This approach was first studied by J. Neumann [16], while the only commercial software product [8] for supporting fault tolerance in FPGAs is also based on this technique.

The main advantages of incorporating an RMR-based technique are summarized, as follows: (i) the corrective action is immediate, since the faulty module never affects the circuit, (ii) there is no need for fault detection procedures, and (iii) the conversion of a non-redundant system to a redundant one is easily undertaken without hardware modifications. On the other hand, this approach cannot recover faults occurred on routing fabric. If these faults are also required to be detected and repaired, another technique (usually based on spare routing resources) must be incorporated in conjunction.

At the RMR-based technique, the reconfigurable platform is encoded as a M-of-N system, consisting of N hardware blocks where at least M ($M<=N$) of them are required for proper device operation. Let's assume that different blocks fail with statistically independent order and if a block fails then it remains non-functional (i.e. the faults are not temporal). If $R(t)$ denotes the probability of an individual block to be still operational at time t, then the reliability of a M-of-N architecture corresponds to the probability that at least M blocks are functional

at time t. By assuming that f_p denotes the probability entire reconfigurable architecture to suffer by a common failure, then the system's reliability is defined, as follows:

$$R_{M_o f_N}(t) = [(1 - f_p) \sum_{k=M}^{N} \{(\binom{N}{k} R^i(t))(1 - R(t))^{N-k}\}] \quad (2)$$

where $\binom{N}{k} = \frac{N!}{(N-k)!k!}$. In case a fault affects the entire architecture (i.e., f_p=0), then the FPGA's reliability is calculated based on Eq. 3. Whenever $R(t)$ <0.5, the hardware redundancy become a disadvantage, as compared to a platform without redundancy at all.

$$R_{M_o f_N}(t) = \sum_{k=M}^{N} \{(\binom{N}{k} R^i(t))(1 - R(t))^{N-k}\} \quad (3)$$

The output of this system is derived from a voting mechanism by receiving a number of N=$\{i_1, i_2, ..., i_N\}$ inputs from a M-of-N architecture. A typical voter does a bit-by-bit comparison and then outputs the majority of the N inputs.

In this work we study an instantiation of the RMR principle, where R=3. This instantiation is also known as Triple Modular Redundancy (TMR), similar to the one incorporated by commercial tool [8]. Fig.2(b) shows the extension of a non fault tolerant architecture (Fig.2(a)) in order to support the TMR approach, while the logic description for the employed majority voter is shown in Fig.2(c). Even though voters can be designed as a pre-fabricate circuits, it is usual to implement them in logic fabric (i.e. BLEs) for increased flexibility.

Such a voting scheme can prevent upset effects through the logic at the final output. Consequently, they can be placed in the end of the combinational and sequential logic blocks, creating barriers for the upset effects. On the other hand, they cannot prevent faulty operations when more than $N/2$ of input signals are not valid.

A critical issue that affects the efficiency of the fault tolerant mechanism is the granularity of application's size that replicated and voted. For instance, a

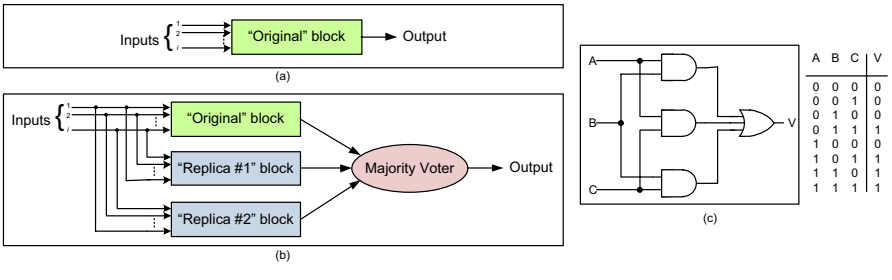


Fig. 2. Logic blocks instantiations (a) without fault tolerant, (b) with the employed TMR redundancy, and (c) the employed majority voter schematic and its truth table.

small size block partition requires a large number of voters that may be too costly in terms of area, performance and power overhead. On the other hand, placing only voters at the last output increases the probability an upset in the routing affecting two (or more) distinct redundant logic blocks to overcome the fault correction.

4 The Proposed Fault Tolerant Framework

In this section we describe the proposed framework for supporting the fault tolerance. The goals of this methodology are to ensure reasonable fault masking at the same time with error detection, diagnosis and recovery, as well as acceptable penalty in delay, power/energy consumption and silicon area.

Up to now, almost the majority of fault tolerant implementations were applied uniformly over the entire device, ignoring about constraints posed from the target application and assuming that the failure probability is uniformly distributed over the device. However, as the fault tolerance is tightly firmed to additional (i.e. spare or redundant) hardware, this assumption may not be an acceptable option, due to the extra area, delay and power overhead. Moreover, for some designs we can afford lower reliability level for a reduced mitigation cost.

As we have already shown in Figure 1, the spatial distribution failure probability does not exhibit a regular picture across the FPGA. Consequently, a uniform fault tolerant design approach cannot be an efficient solution, giving rise to new techniques to achieve the desired circuit reliability at lower (hardware) cost. More specifically, by employing the proposed framework we can determine parts of application with increased failure probability, and then selectively inserts redundancy to them.

Fig.3 shows the proposed framework for supporting fault tolerance in reconfigurable architectures. At this framework rather than modifying the FPGA architecture, we annotate appropriately the application's description with the desired redundancy. Initially, an application profiling is performed to find application's functionalities with increased failure probabilities (i.e. due to increased switching activity). Then, the application is P&R on the target FPGA in order to determine the spatial location where these critical functionalities are physical mapped. By intersecting the placement info for functionalities with their failure probability, it is possible to build a map that shows how this probability is actually distributed over the FPGA. Rather than existing approaches for building TMR-based implementations [8, 12, 13, 14, 15], where redundancy is applied uniformly over the entire architecture, we protect mostly these sensitive areas.

Another critical parameter in this framework is the desired level of fault coverage. This level can be expressed either as the acceptable extra area overhead, or the desired percentage of repaired faults. Among others a more aggressive approach results to higher percentage of repaired faults, but it comes with penalties in additional area, delay and power consumption. In addition to that, in critical regions it is possible to increase the device reliability by selectively incorporating a fault tolerant approach with finer granularity.

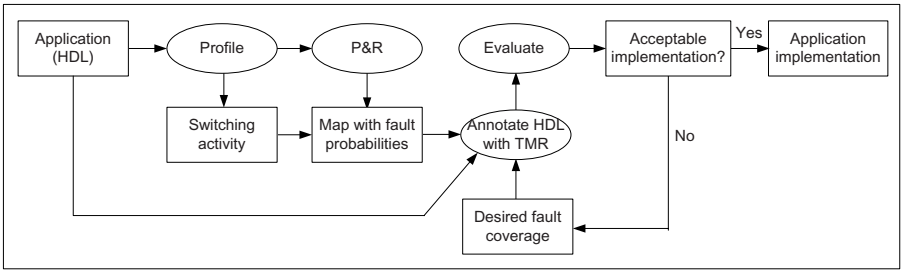


Fig. 3. The proposed fault tolerance framework

Having as input the map with fault probabilities in conjunction to the desired level of fault coverage, it is feasible to annotate application's HDL description with TMR. The focused TMR is a more acceptable solution, as compared to existing uniform implementations [8]. In case additional fault coverage is required, there is another tradeoff between fault coverage and performance/area cost.

In order to automate the proposed framework, we have developed a new tool named *Fault-Free*, which is part from the *MEANDER* tool flow [10]. Apart from this tool, the proposed methodology is transparent to the selected CAD tool chain can be used as an intermediate step between synthesis and P&R.

5 Experimental Results

In this section we provide a number of experimental results that prove the efficiency of the proposed methodology. This evaluation is performed with the usage of 20 biggest MCNC benchmarks [7], while our fault model can study failures occurred due to power, thermal and random effects. Such a fault model follows a distribution of independent (non-correlated) failures. The failure rate during the device's useful life is assumed constant, while there are varying failure rates during the infant mortality (gradually reduction of failure rate) and wear-out phases (gradually increase of failure rate). This approach is acceptable for studying errors in CLBs, SRAM cells, interconnection resources, etc.

Fig.4 plots the sensitive configuration bits by calculating the failure probability of hardware resources for an FPGA array composed by 64×64 slices which implements the *frisc* application [7]. The way this info was extracted and plotted, have already discussed in Fig.1. However, In Fig.4 we depict how this failure probability is updated as we insert gradually more TMR redundancy in application's functionalities mapped onto critical regions. More specifically, we show results about D=30% and D=45% redundant hardware blocks, as compared to implementation without redundancy (i.e. D=0%) and the existing TMR which is applied uniformly over the device (D=100%) [8].

Based on Fig.4, a number of conclusion may be derived. Among others, as we apply more redundancy (i.e. increase D), the failure probability decreases. However, this decrease is not linear, since the improvement in failure probability

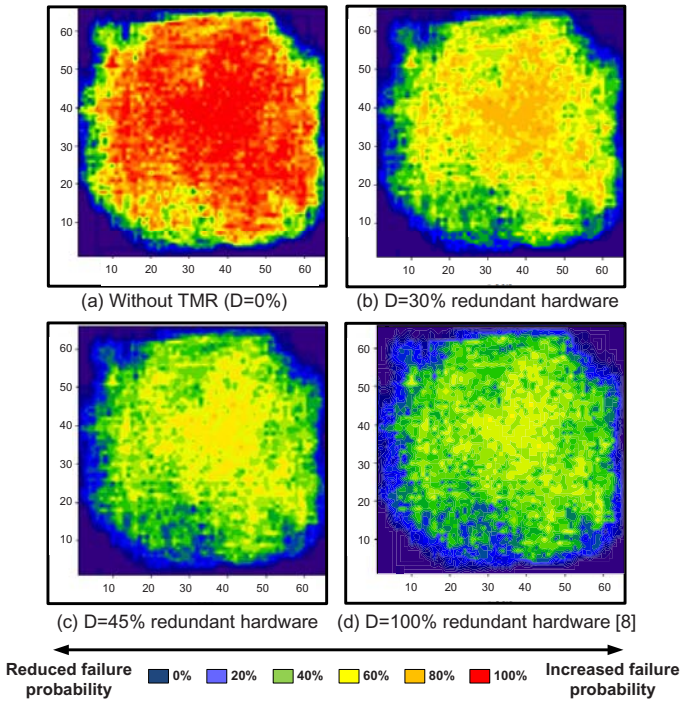


Fig. 4. Resource map with increased failure probability about *frisc* benchmark for different percentages of TMR redundancy

from D=0% to D=30% is much higher as compared to the case where D range from 45% up to 100%. In addition to that, even for the case where TMR is applied uniformly over the FPGA (i.e. D=100%), there is still a failure probability due to errors occurred at the interconnection network.

The efficiency for the alternative TMR-based implementations discussed in this paper can be found in Fig.5, where we study the percentage of repaired over the injected faults. More specifically, in order to make this experiment, for each of the curves we calculate the bitstream file for *frisc* application. This size ranges from B=865 Kbits (D=0%) up to B=1382 Kbits (D =100%), while each of these files remains constant along the corresponding curve. Then, we inject randomly a number of faulty bits in this configuration data. The amount of faulty injected bits, mention with F, ranges from 5% up to 20% of the configuration's file size. We assume that the faults are randomly distributed between logic and interconnection resources. If a single fault occurred at logic block (i.e. CLB) that incorporates TMR mechanism, the fault is repaired. Otherwise (i.e. more than one faults on logic or fault(s) in routing fabric), these faults cannot be eliminated and reported in the Fig.5. The horizontal axis of this figure corresponds to the percentage of injected faulty bits over the size of configuration data, while the vertical one shows the percentage of faulty bits over the total injected bits

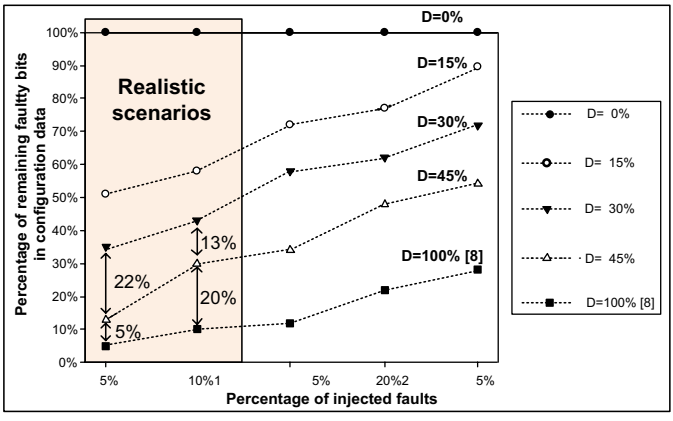


Fig. 5. Percentage of faulty bits for different injection rates regarding the *frisc* application

Table 2. Performance evaluation results for different TMR implementations

Benchmark	# of CLBs	Array	W	Energy×Delay Product (nJ·nsec) No TMR (R=0%)	TMR (R=30%)	TMR (R=45%)	TMR [8] (R=100%)
alu4	1,522	43×43	12	4.77	5.87	6.21	8.21
apex2	1,878	47×47	13	8.30	8.91	9.14	10.3
apex4	1,262	39×39	15	4.13	4.85	5.09	6.3
bigkey	1,817	58×58	7	1.41	1.86	1.92	2.94
clma	8,383	99×99	15	133	148	155	174
des	1,591	68×68	8	16.5	17.1	17.8	19.9
diffeq	1,497	42×42	9	2.21	2.85	3.42	3.77
dsip	1,370	58×58	8	6.29	6.98	7.25	8.19
elliptic	3,604	65×65	12	18.2	20.1	22.3	25.7
ex1010	4,598	73×73	12	37.1	38.6	40.4	44.2
ex5p	1,064	35×35	15	3.86	4.37	5.05	5.97
frisc	3,556	64×64	16	22.0	24.6	25.9	27.4
misex3	1,397	41×41	12	4.35	5.14	5.87	6.59
pdc	4,575	73×73	20	58.0	63.1	65.6	70.3
s298	1,931	47×47	9	12.5	14.58	15.9	19.3
s38417	6,406	87×87	9	24.2	26.99	27.8	29.4
s38514.1	6,293	86×86	10	25.1	27.87	28.9	32.4
seq	1,750	45×45	14	5.47	6.11	6.51	7.97
spla	3,690	65×65	16	25.8	28.56	30.9	38.6
tseng	1,401	41×41	8	0.69	0.93	1.26	1.81
Average	2,979	59×59	12	20.7	22.9	24.1	27.2
Ratio:				1.00	1.11	1.17	1.31

that remain faulty after applying the recovery mechanism. In other words, the vertical axis plots the efficiency to repair faults for the alternative redundant implementations. As a reference point we use the scenario without redundancy, denoted as D=0%. Also, in this graph we highlight the values affecting resonable scenarios about the expected percentages of occurring faults in real-life or exotic (i.e. space, military) applications.

Based on Fig.5 we can conclude that the proposed methodology results to significant error prevention, even for the cases with small area overhead (i.e. D=30%). More specifically, such a redundancy scheme results about to 65% error correction in case where F=5%. In case a more aggressive implementation is assumed (i.e. D=45%), then the error correction is about 88%, as compared to an implementation without redundancy (D=0%). The existing TMR approach [8] results to an additional improvement of 5%, but this impose 55% more redundancy. Also, we have to mention that even when all the functionalities are replicated with TMR (D=100%), there is still a failure probability, since TMR does not prevent faults in routing infrastructure.

One of the main disadvantages of applying fault tolerant techniques is the performance degradation due to additional hardware blocks. Table 2 depicts the number of utilized CLBs, the size of FPGA array, the number of routing wires in each track, as well as the Energy×Delay Product (EDP) for the original (i.e., R=0%), as well as the three TMR instantiations discussed in this paper. For this study we use the 20 biggest MCNC benchmarks [7]. Based on Table 2, the existing way for implementing TMR (R=100%) results to 31% EDP overhead over the initial mapping (R=0%). However, as we shown in Figure 7, such an extreme solution does not lead to significant error prevention, as compared to a case with fewer, but more focused replica blocks. For instance, the HDL annotation that corresponds to D=45%, achieves EDP savings (as compared to D=100%) about 13%, with an almost negligible penalty in error correction (about 5%).

6 Conclusions

A novel framework for exploring and supporting fault tolerance at FPGAs was introduced. This framework is based on the TMR approach, but rather than annotating the whole application's description with redundancy, we replicate only application's functionalities implemented onto regions with increased failure probability. Since our framework does not require dedicated hardware blocks for supporting fault tolerance, it leads to increased flexibility, while it can be also applied to commercial devices as an additional step between synthesis and P&R. Moreover, our approach can provide a tradeoff between the desired reliability level and the extra overhead due to extra hardware resources. Based on experimental results, we shown that the proposed fault tolerant technique is not so much expensive in term of area, delay and energy dissipation, as compared to existing TMR solutions, while it leads to almost similar error correction.

References

1. Lach, J., Mangione-Smith, W., Potkonjak, M.: Efficiently Supporting Fault-Tolerance in FPGAs. In: Int. Symp. on FPGAs, pp. 105–115 (1998)
2. Nikolic, K., Sadek, A., Forshaw, M.: Fault-tolerant techniques for nanocomputers. Nanotechnology 13, 357–362 (2002)
3. Bhaduri, D., Shukla, S.: NANOPRISM: A Tool for Evaluating Granularity vs. In: Reliability Trade-offs in Nano Architectures. GLS-VLSI, pp. 109–112 (2004)
4. Kastensmidt, F., Carro, L., Reis, R.: Fault-Tolerance Techniques for SRAM-Based FPGAs. Springer, Heidelberg (2006)
5. Pratt, B., et al.: Improving FPGA Design Robustness with Partial TMR. In: International Reliability Physics Symposium, pp. 226–232 (2006)
6. Sterpone, L., et al.: On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications. In: DATE Conference, pp. 336–341 (2008)
7. Yang, S.: Logic Synthesis and Optimization Benchmarks, Tech. Report (1991)
8. Xilinx TMR Tool, `http://www.xilinx.com/quickstart/tmrq.htm`
9. Ziegler, J., et al.: IBM Experiments in Soft Fails in Computer Electronics (1978-1994). IBM Journal of Research and Development 40(1), 3–18 (1996)
10. 2D MEANDER Framework, `http://proteas.microlab.ntua.gr`
11. Colinge, J.: Silicon-on-Insulator Technology: Overview and Device Physics. In: IEEE Nuclear Space Radiation Effects Conference 2001 (2001)
12. Yu, J., et al.: Defect Tolerant FPGA Switch Block and Connection Block with Fine Grain Redundancy for Yield Enhancement. In: FPGA, pp. 255–262 (2005)
13. Jain, R., Mukherjee, A., Paul, K.: Defect Aware Design Paradigm for Reconfigurable Architectures. In: Computer Society Annual Symposium on VLSI, pp. 91–96 (2006)
14. Doumar, A., Ito, H.: Defect and Fault tolerance FPGAs by shifting the configuration data. In: IEEE Symposium on Defect and Fault-tolerance, pp. 377–385 (1999)
15. Camregher, N., et al.: Analysis of Yield Loss due to Random Photolithographic Defects in the Interconnect Structure of FPGAs. In: FPGA, pp. 138–148 (2005)
16. Neumann, J.: Probabilistic logics and the synthesis of reliable organisms from unreliable components. In: Automata Studies, pp. 43–98 (1956)
17. Black, J.: Electromigration - A Brief Survey and Some Recent Results. IEEE Transaction on Electron Devices ED-16, 338–347 (1974)
18. Siozios, K., Soudris, D.: A Power-Aware Placement and Routing Algorithm Targeting 3D FPGAs. Journal of Low-Power Electronics 4(3), 275–289 (2008)
19. Soudris, D., et al.: AMDREL: A Novel Low-Energy FPGA Architecture and Supporting CAD Tool Design Flow. In: Fine and Coarse-Grain Reconfigurable Systems, pp. 152–180 (2007)

QUAD – A Memory Access Pattern Analyser

S. Arash Ostadzadeh, Roel J. Meeuws, Carlo Galuzzi, and Koen Bertels*

Computer Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Delft, The Netherlands
{S.A.Ostadzadeh,R.J.Meeuws,C.Galuzzi,K.L.M.Bertels}@tudelft.nl

Abstract. In this paper, we present the Quantitative Usage Analysis of Data (QUAD) tool, a sophisticated memory access tracing tool that provides a comprehensive quantitative analysis of memory access patterns of an application with the primary goal of detecting actual data dependencies at function-level. As improvements in processing performance continue to outpace improvements in memory performance, tools to understand memory access behaviors are inevitably vital for optimizing the execution of data-intensive applications on heterogeneous architectures. The tool, first in its kind, is described in detail and the benefit and the qualities of the presented tool are described on a real case study, the *x264* benchmarking application.

1 Introduction

With the increased proliferation of Chip Multiprocessors (CMP), there is a compelling need for utility tools to facilitate the application development process, tuning and optimization. This requirement becomes more critical with the introduction of hybrid architectures incorporating reconfigurable devices [1]. Since the applications form the basis of such designs, the need to tune the underlying architecture for extracting maximum performance from the software code has become imperative [2]. As the size of reconfigurable fabrics increases, mapping an entire application onto a reconfigurable device does not seem elusive anymore. Traditionally, FPGAs contained the logic and some temporary memory for a kernel, or even the logic for the whole application, but never the logic and memory for an entire application [3]. Although this in itself is an important step towards decreasing the processor-memory gap, there is an inevitable demand for tools that can help users to have a clear understanding of the memory requirements of an application. *To accomplish this goal, a thorough analysis of the memory access behavior of an application is vital*. This demand proves to be even more crucial considering the fact that the main obstacle limiting the performance of reconfigurable systems is the memory latency [4].

Even in the best case scenario, when all the memory addresses can fit on a single chip, the latency to access memory locations from different parts imposes a substantial delay. In [3], the memory access behavior of an application (*g721_e*) whose logic and

* This research has been funded by the hArtes project EU-IST-035143, the Morpheus project EU-IST-027342 and the Rcosy Progress project DES-6392.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 269–281, 2010.

memory was entirely mapped onto a reconfigurable device, is discussed. The idea of mobile memory is also investigated to dynamically move the memory closer to the location where it is accessed. It justifies the motivation to lessen the distance between the accessors and the memory locations.

Inspecting the behavior of an application in general, and the actual pattern of memory accesses in particular, is an essential aspect of carrying out effective optimizations for the application development of reconfigurable systems. As a result, many research initiatives are emerging that target support tools for application behavior analysis from different perspectives.

The main contributions of this paper are the following:

- the description of an efficient tool, QUAD, to provide information that can be used in addressing memory-related bottlenecks in reconfigurable computing systems
- the detection of actual data dependency between functions compared to conventional data dependency discovered by similar memory access analysers
- the validation of the proposed tool in a real case study

The rest of the paper is organized as follows. Section 2 gives an overview of the related research. In Section 3, we present an overview of the Delft Workbench (DWB) and the profiling framework which includes QUAD as a dynamic memory access profiler. Section 4 introduces QUAD and describes some of the design and implementation issues. In Section 5, a real application case study is examined. Finally, Section 6 provides concluding remarks and an outline of the future research.

2 Related Research and Problem Definition

Profilers are tools that allow users to analyze the run-time behavior of an application in order to identify the types of performance optimizations that can be applied to the application and/or the target architecture. Generally, profiling refers to a technique for measuring where programs consume resources, including CPU time and memory. General profiling tools such as *gprof* [5], can provide function-level execution statistics for the identification of application hot-spots. *However, they do not distinguish between computation time and memory access time*. As a result, they can not be employed to locate potential system bottlenecks regarding memory-related problems.

In [6,7], the authors provide target independent software performance estimations. However, they lack a thorough memory access analysis, which is vital in tuning performance optimizations in hybrid reconfigurable architectures. [8] describes an approach for evaluating the performance and memory access patterns of multimedia applications through profiling. The tool is only utilized for algorithmic complexity evaluation and its accuracy in performance estimation is not investigated. The way an algorithm interacts with memory has a large impact on performance. More precisely, the memory reference behavior of an application, at the most basic level, depends on the intrinsic nature of the application. However, the developer still has considerable flexibility in manipulating the algorithm, data structures and program structure to change the memory reference patterns [9].

Most existing memory access analysis tools only focus on detecting memory bottlenecks, or faults/bugs/leaks and provide no detailed information regarding the inherent

data dependencies in a program's memory reference behavior [10,11]. One of the early simple tools developed for understanding memory access patterns of Fortran programs is presented in [12]. The tool instruments a program and produces a flat trace file of all memory accesses which can be visualized later. Similarly, a tool set is presented in [13] to reveal the pattern of memory references. It generates a set of histograms for each memory access in a program regarding the strides of references.

In [14], the authors present a quantitative approach to analyze parallelization opportunities in programs with irregular memory access patterns. Applications are classified into three categories with low, medium and high dependence densities. Similar to our work, Embla [15] allows the user to discover the data dependencies in a sequential program, thereby exposing opportunities for parallelization. Embla performs a dynamic analysis and records dependencies as they arise during program execution. *However, in this work, we intend to discover the* ***actual data dependency****, which is different from the conventional data dependency referred to in Embla and other similar tools.* By definition, **data dependency** is a situation in which a program segment (instruction, block, function, etc.) refers to the data of a preceding segment. Actual data dependency arises when a function consumes data that is produced by another function earlier. In other words, the common argument passing by the caller function to the callee regarding data distribution does not necessarily imply that the data will be used later in the called function. *Furthermore, in our approach the usual restriction of data dependency detection based on hierarchies of function calls (commonly depicted with call graphs) is relaxed as we merely trace the journey of bytes through memory addresses and do not rely on the control dependencies of tasks to detect potential data dependencies.*

In this paper, we present QUAD (Quantitative Usage Analysis of Data), a memory access tracing tool that provides a comprehensive quantitative analysis of the memory access patterns of an application with the primary goal of detecting actual data dependencies at function-level. To the best of our knowledge, this is the first tool that addresses the actual data dependency detection and abstracts away from the properties of data dependency detection of an application on a particular architecture. In addition, previous research into data dependency detection has mainly focused on the discovery of parallelization opportunities. However, we do not necessarily target parallel application development. Even though QUAD can be employed to spot coarse-grained parallelism opportunities in an application, it practically provides a more general-purpose framework that can be utilized in various reconfigurable systems' optimizations by estimating effective memory access related parameters, e.g. the amount of unique memory addresses used in data communication between two cooperating functions. QUAD can also be used to estimate how many memory references are executed locally compared to the amount of references that have to go to the main memory.

Accessing memory locations sequentially, or within predefined strides, can be considered very efficient on cache-based computing systems. This information can be useful in the presence of memory hierarchies on a system and can be a source of performance improvement. Frequently, it is possible for the programmer to restructure the data or code to achieve better memory reference behavior. Inefficient use of memory remains a significant challenge for application developers. QUAD can also be utilized to diagnose memory inefficiencies by reporting useful statistics, such as boundaries of

memory references within functions, detection of unused data in memory, etc. QUAD can be easily ported to different architectures as long as there exists a primitive tool set that can provide the basic memory read/write instrumentation capabilities, like BIT [16] for instrumenting java byte codes.

The main features of the tool proposed in this paper, are listed in the following.

- QUAD detects the actual data dependency at function-level in an application, which involves a higher degree of accuracy compared to the conventional data dependency detected by other similar tools.
- QUAD does not require any modification of the binaries and it has no compiler dependence other than debug information. It also abstracts away from the properties of a particular architecture.

3 Profiling Framework in DWB

This work has been carried out in the context of the Delft WorkBench (DWB) [17] and the hArtes [18] projects. The DWB is a semi-automatic tool platform for integrated hardware/software co-design targeting heterogeneous computing system containing reconfigurable components. It aims to be a comprehensive platform supporting development at all levels starting from profiling and partitioning to synthesis and compilation. Conversely, the closely related hArtes project targets the same heterogeneous systems. However, it also takes into account digital signal processing hardware and provides its own heterogeneous platform.

The Delft Workbench focuses on four main steps within the entire heterogeneous system design.

- *Code Profiling and Cost Modeling* - focuses on identifying application hot-spots and on estimating implementation costs for different components [19].
- *Graph Transformations* - aim to use the profiling information and estimates for clustering tasks, partitioning tasks over components, optimizing tasks, or restructuring the task graph [20,21,22].
- *VHDL Generation* - When tasks need to be implemented on Reconfigurable components, this tool allows to automatically translate the high level language descriptions into VHDL [23].
- *Retargetable Compiler* - schedules and combines all the implemented parts of the application and it generates the executable binary [24].

The work in this paper mainly revolves around code profiling. Figure 1 depicts in detail the profiling framework envisioned by the DWB platform. We distinguish between static and dynamic profiling paths. Static profiling can provide estimates in a small amount of time, whereas dynamic profiling provides accurate measurements of several aspects like execution time and memory access behavior. The dynamic profiling path focuses on run-time behavior of an application and, therefore, is not as fast as the static profiling. Furthermore, the dynamic profiling requires representative input data in order to provide relevant measurements. *gprof* is used to identify hot-spots and frequently executed functions, while QUAD is concerned with tracing and revealing the pattern of

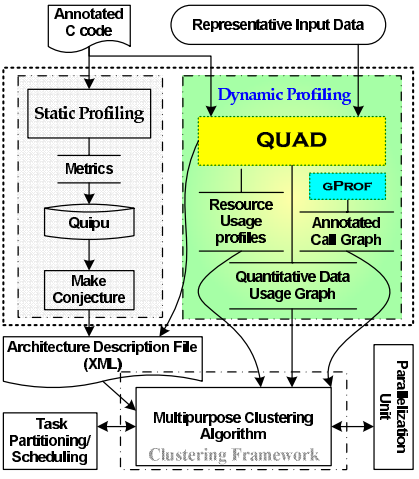


Fig. 1. Profiling Framework within DWB

memory references with the primary aim to detect actual data dependencies between functions. In this paper, we only focus on the representation and implementation details necessary for the description of QUAD.

4 QUAD Design and Implementation

4.1 Pin

QUAD is a Dynamic Binary Analysis (DBA) tool which analyzes an application at the machine code level as it runs. DBA tools can be built from scratch or be implemented using a Dynamic Binary Instrumentation (DBI) framework. Instrumentation is a technique for inserting extra code into an application to observe its behavior. This process can be performed at various stages either in the source code, or at compile-time, or at post-link time, or at run-time. QUAD is implemented as a tool using the Pin [25] runtime binary instrumentation system. By using Pin, we have the benefit of working transparently with unmodified Linux, Windows and MacOS binaries on Intel ARM, IA32, 64-bit x86, and Itanium architectures. Thanks to the instrumentation transparency, Pin preserves the original application behavior. The application uses the same addresses (both instruction and data) and the same values (both register and memory) as it would in an uninstrumented execution. This transparency, vital for correctness, results in more relevant information collected by the instrumentation. Dynamic instrumentation is particularly beneficial for this type of tools. It captures the execution of arbitrary shared libraries in addition to the main program and it has no dependence on the instrumented application's compiler. Requiring only a binary and being compiler-independent does not imply that the source code is not needed for program revisions. Instead, it provides flexibility for the tool to be language-independent and it can be used with any compiler toolchain that produces a common binary format. Furthermore, it does not require the

user to modify the build environment to recompile the application with special profiling flags.

Since QUAD relies on dynamic instrumentation and it is compiler-independent, detecting the producers/consumers of the data being stored/loaded via memory addresses must be done in the absence of any kind of control/data flow or call graphs. As a consequence, the detection is based only on the dynamic execution of the program. In order to provide some degree of flexibility, QUAD also implements and maintains its own call graph during the execution of a program.

4.2 QUAD Overview

QUAD has been designed as a base system to provide useful quantitative information about the data dependence between any pair of cooperating functions in an application. Data dependence is estimated in the sense of producer/consumer binding. More precisely, QUAD reports which function is consuming the data produced by another function. The exact amount of data transfer and the number of Unique Memory Addresses (UMA) used in the transfer process are calculated. Based on the efficient Memory Access Tracing (MAT) module implemented in QUAD, which tracks every single access (read/write) to a memory location, a variety of statistics related to the memory access behavior of an application can be measured, e.g. the ratio of local to global memory accesses in a particular function call.

Figure 2 illustrates the architectural overview of QUAD along with the components in Pin. At the highest level, there is a Virtual Machine (VM), a code cache, and an instrumentation API. The main component inside QUAD is the MAT module, which is responsible for building and maintaining dynamic trie [26] data structures to provide relevant memory access information as fast as possible. The trie data structure acts as a shadow memory for each byte accessed within the address space of an application.

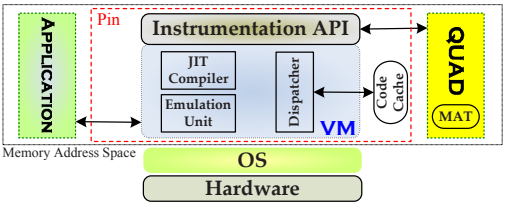


Fig. 2. Architectural overview of QUAD

The VM consists of a Just-In-Time (JIT) compiler, an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments the application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering (leaving) the VM from (to) the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for system calls which require special handling from the VM. Since Pin does not reside in the kernel of the operating system, it can only capture user-level code. As Figure 2 shows, three binary programs are present when an instrumented program is

running: the application, Pin, and QUAD. Pin is the engine that instruments the application. QUAD contains the instrumentation and analysis routines and it is linked with a library that allows QUAD to communicate with Pin.

4.3 QUAD Implementation

The interfaces to most run-time binary instrumentation systems are API calls that allow developers to hook in their instrumentation routines. In Pin, the API call to *INS_AddInstrumentationFunction()* allows a user to instrument programs based on a single instruction while the *RTN_AddInstrumentFunction()* provides instrumentation capability at routine granularity. QUAD uses these two API routines to set up calls to the instrumentation routines *Instruction()* and *UpdateCurrentFunctionName()*. These two instrumentation routines, in turn, call the two main analysis routines *RecordMemRef()* and *EnterFunc()* which are responsible for updating tracing information of memory references and maintaining an internal call graph respectively. Figure 3 illustrates an implementation overview of QUAD. The detailed algorithms associated with each module are not included in this paper for brevity. Nevertheless, in the following we provide some description highlights.

The initialization process in the main module includes Pin system initialization, command line options parsing, internal call graph initialization, and some output XML file preprocessing. The *Instruction()* instrumentation routine sets up the call to *RecordMemRef()* routines every time an instruction that references memory is executed. When Pin starts the execution of an application, the JIT calls *Instruction()* to insert new instructions into the code cache. If the instruction references memory (read or write), QUAD inserts a call to *RecordMemRef()* before the instruction, passing the Instruction Pointer (IP), Effective Address (EA) for the memory operation, a flag indicating whether it is a read or write operation, number of bytes read or written, and a flag showing whether or not the instruction is a prefetch. The analysis routine returns immediately upon detection of a prefetch state for an instruction. *INS_InsertPredicatedCall()* injects the analysis routine and ensures that the analysis routine is invoked only if the memory instruction is predicated true. There is also a separate *RecordMemRef()* analysis routine for the case that we are interested to trace local memory references within the stack region. In this case, the value of stack pointer is also passed to the analysis routine for further investigation. *Instruction()* also monitors the *ret* instruction to leave a function and upon detection calls a different analysis routine that updates the internal call graph if necessary.

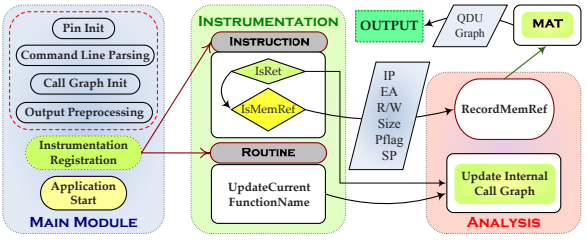


Fig. 3. Implementation overview of QUAD

The main objective of *RecordMemRef()* is to identify the function responsible for the current memory reference and to pass the required information to the MAT module. The instrumentation at the routine granularity in QUAD is responsible for pushing the name of the currently called function onto an internal call stack. Note that the respective pop operation is later performed upon detection of the *ret* instruction. QUAD needs to maintain its own call graph because a user may not be interested to dive into library routines or routines that are not included in the main binary image file. In these cases, QUAD assumes the most recent caller routine from the main binary as the one responsible for issuing memory references.

4.4 Memory Access Tracing (MAT) Module

In order to spot and to extract memory reference information during the execution of an application, an efficient memory access tracing module is implemented. The tracing process utilizes *trie* data structures for fast storage and retrieval. MAT defines trie structures with base 16 that is representative of memory addresses in hexadecimal format. Each hexadecimal digit in a 32-bit memory address corresponds to one level in the trie data structure, leaving 8 levels deep in the hierarchy for complete address tracing. The trie data structure is designed to grow dynamically on demand for reducing memory usage overhead as much as possible. This means that if a particular memory address is fed to MAT for the very first time, the levels required to trace that particular address are created in the trie. Hence, no space is allocated for unused memory addresses. The saving is considerable because the complete data structure is expected to be gigantic and may result in memory overflow in some systems.

The memory reference recording process is accomplished in two distinct phases. In the first phase, we trace an 8-level trie for a particular memory address. For each memory reference three different arguments are specified: memory address, function ID, and read/write flag. In case of a write access, the corresponding shadow memory address in the trie is labeled with the caller (producer) function ID. When a read flag is detected, the function ID responsible for the most recent write in the memory address is retrieved and passed along with the consumer function ID to the second phase where a data communication record is created.

The memory reference information gathered by QUAD during the execution of an application are reported in two separate formats. The producer/consumer binding information is saved in a text file using standard portable XML format. This makes it easy for third-party applications to import data for further interpretation and processing. The actual data dependency bindings between functions is also provided in the form of a graph data structure, which is called Quantitative Data Usage (QDU) graph.

5 Case Study

We used *x264* [27] as a benchmark to test QUAD in a series of experiments. The goal is to have an initial understanding of the application behavior regarding the data communication patterns, memory usage, and memory requirements. *The information provided by QUAD can be used later in HW/SW partitioning and mapping as well as to hint application developers how to revise and optimize the code for a specific architecture.*

x264 is a free library for encoding H.264/AVC video streams. The version used in this work is a modified *x264 r654* encoder tailored to the MOLEN [28] paradigm taking into account the restrictions in terms of coding rules accepted by the DWARV hardware compiler [23].

5.1 Experimental Setup

All the experiments were executed on an Intel 64-bit Core 2 Quad CPU Q9550 @ 2.83GHz with the main memory of 8GB, running Linux kernel v2.6.18-164.6.1.el5. The *x264* source code was compiled with *gcc* v3.4.6 and with the profiling option enabled. We need to use *gprof* as an auxiliary tool to interpret the data and to make some conclusions. The standard command line options used to run the 64-bit compiled version of *x264* was the following:

1. *–no-ssim* - to disable the computing of structural similarity (SSIM) index;
2. *rate control -q1* - to indicate almost lossless compression;
3. *–no-asm* - to disable all stream processing optimizations based on CPU capabilities.

The 64-bit version of QUAD was used with the following command line options:

1. *ignore_stack_access* - to ignore all the memory accesses to the stack region. This gives a clear view of the data transferred via non-stack region.
2. *use_monitor_list* - to include only some critically potential functions in the report files, due to the high complexity and the size of the *x264* application.

akiyo_qcif was used as the input data file for encoding. It is a raw YUV 4:2:0 file with the resolution of 176x144 pixels containing 300 frames. The output was in raw byte stream format.

5.2 Experimental Analysis

x264 contains over two hundreds functions. The set of functions to be called are determined based on different options selected by the user or by the input/output file specifications. On the basis of the computation-intensive kernels identified in the flat profile provided by *gprof*, we chose a number of functions (or series of functions) for further inspection. The main criterion adopted here was the suitability for the DWARV compilation tool. Table 1 presents part of the flat profile.

Table 1. Flat profile for *x264*

function name	% time	self seconds	calls	total ms/call	self ms/call
pixel_satd_wxh	34.51	0.49	1361024	0	0
x264_cabac_encode_decision	8.45	0.12	10808084	0	0
get_ref	7.75	0.11	1165182	0	0
block_residual_write_cabac	7.04	0.1	400643	0	0
x264_pixel_sad_x4_16x16	5.63	0.08	88506	0	0
x264_frame_filter	5.63	0.08	2700	0.03	0.03
x264_pixel_sad_x4_8x8	3.52	0.05	243588	0	0
refine_subpel	2.82	0.04	151014	0	0
motion_compensation_chroma	2.11	0.03	442213	0	0

% time is the percentage of the total execution time of the program used by the function; ***self seconds*** is the number of seconds accounted for by the function alone; ***calls*** is the number of times a function is invoked; ***total ms/call*** is the average number of milliseconds spent in the function and its descendants per call; ***self ms/call*** is the average number of milliseconds spent in the function per call.

Table 2. Summary of data produced and consumed by **satd**- and **sad**-related kernels

function name	IN	IN UMA	OUT	OUT UMA
pixel_satd_wxh	326310528	137425	91578266	5126
x264_pixel_satd_16x16	40607660	1523	26133254	1233
x264_pixel_satd_16x8	5795852	1009	2849136	722
x264_pixel_satd_4x4	34342432	2745	18099806	2318
x264_pixel_satd_4x8	3091448	1953	1610194	1644
x264_pixel_satd_8x16	6248933	1053	3152620	650
x264_pixel_satd_8x4	3019905	1937	1568928	1594
x264_pixel_satd_8x8	91709500	3307	46203830	3049

function name	IN	IN UMA	OUT	OUT UMA
x264_pixel_sad_16x16	59965275	103924	5704610	624
x264_pixel_sad_16x8	9159212	55626	1736556	480
x264_pixel_sad_4x4	0	0	0	0
x264_pixel_sad_4x8	0	0	0	0
x264_pixel_sad_8x16	8924130	53174	1709404	566
x264_pixel_sad_8x4	0	0	0	0
x264_pixel_sad_8x8	53730341	89155	10838624	664

IN represents the total number of bytes read by the function; ***IN UMA*** indicates the total number of unique memory addresses used in reading; ***OUT*** represents the total number of bytes read by any function in the application from memory locations that the specified function has written to those locations earlier; ***OUT UMA*** indicates the total number of unique memory addresses used in writing.

As presented in the Table 1, **pixel_satd_wxh** is the main kernel of the application accounting for 35% of the total execution time. It was initially selected as the main candidate kernel for hardware mapping along with **sad**-related functions. Although **x264_cabac_encode_decision** is the most frequently called function, each call has a smaller contribution compared to **pixel_satd_wxh**. As a result, the overall contribution of **x264_cabac_encode_decision** drops considerably. There are several **satd**-related functions defined in the form of Macros corresponding to various block sizes. These macros, when expanded, create different functions calling the main **pixel_satd_wxh** making it a very critical function on the execution path. Table 2 summarizes the results of memory access tracing for **satd**- and **sad**-related functions. As expected, **pixel_satd_wxh** is the top consumer on the list (in total more than 300MB) as all the other **satd**-related functions call this kernel to perform their primary tasks.

It is worth noting that some **sad**-related functions (the ones with 4 rows and/or columns) do not exhibit any data transfers, which is an indication that they are not called. This depends on the input file characteristics and options used. Although the kernels are intensely reading (writing) data from (to) memory, the number of unique memory addresses used in the data transfer is limited (MBs data transfer vs. KBs locations). This indicates the possibility of allocating memory buffers, e.g. on FPGA BRAMs to gain better performance. QUAD can also provide a detailed map of the used memory addresses for the examination of mapping opportunities on a target architecture. The auxiliary functions communicating with kernels are recognized and presented in the QDU graph. These auxiliary functions can be sources of further investigation. For example, one might investigate mapping tightly coupled functions on FPGA and creating a buffer to facilitate the data transfer, or merging auxiliary function(s) with the primary kernel to cut off data transfers between functions. In case of **pixel_satd_wxh**, **mc_copy_w16** is tightly coupled with the main kernel and it is responsible for producing approximately 130 MB of data (75k UMA). Further inspection of **mc_copy_w16** reveals that it belongs to the *motion compensation* library and merely calls the built-in *memcpy* routine of the C language library in a loop in order to create a block of pixels from a flat set of pixels with a predefined stride. It seems feasible to rewrite the routine from scratch and to combine it with the kernel.

sad-related routines are also defined in the form of Macros corresponding to various block sizes. Unlike **satd**-related functions, these macros, when expanded, create different functions with separate bodies. In order to evaluate the impact of an identical kernel routine for the **sad**-related functions, we created a new function called **pixel_sad_wxh** and revised all the **sad**-related functions to call this critical kernel. It is a more likely

Table 3. Flat profile for the revised *x264* (non-instrumented and QUAD-instrumented binaries)

function name	% time	self seconds	calls	rank	% time(+QUAD)	self seconds	rank(+QUAD)
pixel_sad_wxh	33.45	0.5	2646060	1	22.94	546.53	1
pixel_satd_wxh	24.32	0.36	1361024	2	7.61	181.18	4
x264_frame_filter	8.11	0.12	2700	3	9.96	237.21	3
get_ref	7.43	0.11	1165182	4	7.49	178.41	5
motion_compensation_chroma	5.41	0.08	442213	5	3.25	77.53	7
block_residual_write_cabac	4.73	0.07	400643	6	2.64	62.96	8
x264_cabac_encode_decision	2.03	0.03	10808084	8	14.62	348.32	2
x264_macroblock_cache_load	2.03	0.03	29700	9	1.49	35.39	12
x264_cabac_encode_bypass	1.35	0.02	2234007	10	4.84	115.29	6

Table 4. Data produced/consumed by **pixel_satd_wxh** & **sad**-related functions in the revised *x264*

function name	IN	IN UMA	OUT	OUT UMA
pixel_sad_wxh	816414788	245781	100154164	2976
pixel_satd_wxh	326389008	137389	91580364	5108
x264_pixel_sad_16x16	16169821	885	7275066	614
x264_pixel_sad_16x8	4588984	793	2122520	510
x264_pixel_sad_8x16	4480742	843	2066828	552
x264_pixel_sad_8x8	39174841	1011	17382666	732

candidate for implementation on FPGA devices. Table 3 depicts part of the flat profile for *x264* after the introduction of the new **pixel_sad_wxh** kernel. **pixel_sad_wxh** now gets the dominant position with the contribution of about 33.5% to the whole application's execution time. Note that it is also called nearly double of the times compared to the second dominant kernel, **pixel_satd_wxh**. The *gprof* flat profile of the QUAD-instrumented binary is also provided. The considerable increase in the self-seconds contribution of each kernel is due to the overhead introduced by the QUAD instrumentation code routines. However, the ranking provided in this respect is somehow more representative of real execution time regarding the data communication between functions via non-local memory. It is due to the fact that we do not take into consideration stack-region memory accesses and only upon detection of a non-local memory access, a time-consuming routine to parse the trace trie is called.

Table 4 summarizes the results of memory access tracing for **pixel_satd_wxh** and **sad**-related kernels in the revised version of *x264*. As expected, the communication load of **pixel_sad_wxh** dominates the former main kernel **pixel_satd_wxh**. However, there is a substantial increase in the total amount of bytes consumed by this new kernel (about 800MB) compared to the collective number of bytes consumed by the **sad**-related functions in the original version. This is due to the fact that the **sad**-related functions have to pass extra arguments to the new kernel. The new kernel uses the extra information to distinguish between different **sad**-related functions. The number of bytes consumed in **pixel_sad_wxh** can be further reduced by a revision of the code to minimize this overload. The total number of bytes produced and consumed beside the unique memory addresses used inside individual **sad**-related functions are significantly reduced since the load is shifted to the new **pixel_sad_wxh** kernel.

Including the local memory accesses in the tracing would also reveal notable observations. By including stack region accesses, **pixel_satd_wxh** becomes the dominant kernel once again (22.15% of the whole contribution). This indicates that if there is no

intention to map the local temporary memory into the hardware and fetching data from external memory is expensive, there is a high probability that mapping **pixel_satd_wxh** onto hardware is preferable compared to **pixel_sad_wxh**.

6 Conclusions

The gap between processors and memory performance will be a major challenge for the optimization of memory-bound applications on hybrid reconfigurable systems. This demands the development of utility tools to help users in tuning applications for maximal performance gain of these systems. In this paper, we have presented QUAD, a tool that provides a comprehensive quantitative analysis of the memory access patterns of an application. QUAD can be employed in detecting coarse-grained parallelism opportunities as well as providing information about the memory requirements of an application. The information is particularly useful in buffer size estimation for local memory reallocation to store data in case of mapping kernels onto reconfigurable devices that initially cause memory bandwidth problems. QUAD has been tested on a real *x264* benchmarking application and a detailed discussion was presented based on the extracted statistics. In the future work, we are planning to utilize the information provided by the tool for task clustering in heterogeneous reconfigurable systems.

References

1. Kwok, T.O., Kwok, Y.K.: On the design, control, and use of a reconfigurable heterogeneous multi-core system-on-a-chip. In: Proc. of PDP, pp. 1–11 (2008)
2. Kempf, T., Karuri, K., Wallentowitz, S., Ascheid, G., Leupers, R., Meyr, H.: A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In: Proc. of DATE, pp. 468–473 (2006)
3. Yan, R., Goldstein, S.C.: Mobile memory: Improving memory locality in very large reconfigurable fabrics. In: Proc. of FCCM, pp. 195–204 (2002)
4. Hauck, S., Dehon, A.: Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon). Morgan Kaufmann, San Francisco (2007)
5. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. 17(6), 120–126 (1982)
6. Giusto, P., Martin, G., Harcourt, E.: Reliable estimation of execution time of embedded software. In: Proc. of DATE, pp. 580–589 (2001)
7. Bammi, J.R., Kruijtzer, W., Lavagno, L., Harcourt, E., Lazarescu, M.T.: Software performance estimation strategies in a system-level design tool. In: Proc. of CODES, pp. 82–86 (2000)
8. Ravasi, M., Mattavelli, M.: High-level algorithmic complexity evaluation for system design. J. Syst. Archit. 48(13-15), 403–427 (2003)
9. Martonosi, M., Gupta, A., Anderson, T.: Memspy: analyzing memory system bottlenecks in programs. In: Proc. of Sigmetrics/Performance, pp. 1–12 (1992)
10. Venkataramani, G., Doudalis, I., Solihin, Y., Prvulovic, M.: Memtracker: An accelerator for memory debugging and monitoring. ACM Trans. Archit. Code Optim. 6(2), 1–33 (2009)
11. Choudhury, A.N.M.I., Potter, K.C., Parker, S.G.: Interactive visualization for memory reference traces. Comput. Graph. Forum 27(3), 815–822 (2008)

12. Brewer, O., Dongarra, J., Sorensen, D.: Tools to aid in the analysis of memory access patterns for FORTRAN Programs. Parallel Computing 9(1), 25–35 (1988)
13. Balle, S., Steely Jr., S.: Memory Access Profiling Tools for Alpha-based Architectures. In: Kågström, B., Elmroth, E., Waśniewski, J., Dongarra, J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 28–37. Springer, Heidelberg (1998)
14. von Praun, C., Bordawekar, R., Cascaval, C.: Modeling optimistic concurrency using quantitative dependence analysis. In: Proc. of PPoPP, pp. 185–196 (2008)
15. Faxén, K., Popov, K., Janson, S., Albertsson, L.: Embla–Data Dependence Profiling for Parallel Programming. In: Proc. of CISIS, pp. 780–785 (2008)
16. Lee, H.B., Zorn, B.G.: Bit: a tool for instrumenting java bytecodes. In: Proc. of USITS, pp. 7–16 (1997)
17. Bertels, K., Vassiliadis, S., Panainte, E.M., Yankova, Y.D., Galuzzi, C., Chaves, R., Kuzmanov, G.: Developing applications for polymorphic processors: the delft workbench. Technical report, CE Group (2006)
18. Bertels, K., Kuzmanov, G., Panainte, E.M., Gaydadjiev, G.N., Yankova, Y.D., Sima, V., Sigdel, K., Meeuws, R.J., Vassiliadis, S.: Hartes toolchain early evaluation: Profiling, Compilation and HDL generation. In: Proc. of FPL, pp. 402–408 (2007)
19. Meeuws, R.J., Sigdel, K., Yankova, Y.D., Bertels, K.: High level quantitative interconnect estimation for early design space exploration. In: Proc. ICFPT, pp. 317–320 (2008)
20. Ostadzadeh, S.A., Meeuws, R.J., Sigdel, K., Bertels, K.: A clustering framework for task partitioning based on function-level data usage analysis. In: Proc. of FPGA, p. 279 (2009)
21. Ostadzadeh, S.A., Meeuws, R.J., Sigdel, K., Bertels, K.: A Multipurpose Clustering Algorithm for Task Partitioning in Multicore Reconfigurable Systems. In: Proc. of CISIS, pp. 663–668 (2009)
22. Galuzzi, C., Bertels, K.: A framework for the automatic generation of instruction-set extensions for reconfigurable architectures. In: Woods, R., Compton, K., Bouganis, C., Diniz, P.C. (eds.) ARC 2008. LNCS, vol. 4943, pp. 280–286. Springer, Heidelberg (2008)
23. Yankova, Y.D., Kuzmanov, G., Bertels, K., Gaydadjiev, G.N., Lu, Y., Vassiliadis, S.: Dwarv: Delftworkbench automated reconfigurable VHDL generator. In: Proc. of the FPL, pp. 697–701 (2007)
24. Panainte, E.M., Bertels, K., Vassiliadis, S.: The molen compiler for reconfigurable processors. ACM TECS 6(1), 6 (2007)
25. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proc. of PLDI, pp. 190–200 (2005)
26. Fredkin, E.: Trie memory. ACM Commun. 3(9), 490–499 (1960)
27. x264, `http://www.videolan.org/developers/x264.html`
28. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Trans. on Computers 53(11), 1363–1375 (2004)

Hierarchical Loop Partitioning for Rapid Generation of Runtime Configurations

Siew-Kei Lam[1], Yun Deng[1], Jian Hu[2], Xilong Zhou[2], and Thambipillai Srikanthan[1]

[1] Centre for High Performance Embedded Systems,
Nanyang Technological University,
50 Nanyang Avenue, Singapore
{assklam,dengyun,astsrikan}@ntu.edu.sg
[2] School of Software and Microelectronics,
Peking University, P.R. China

Abstract. Runtime reconfiguration provides an efficient means to reduce the hardware cost, while satisfying the performance, flexibility and power requirements of embedded systems. The growing complexity of the applications necessitates methods that can rapidly identify a suitable set of configurations by splitting the computational structures into temporal partitions in order to evaluate the benefits of runtime reconfiguration early in the design cycle. In this paper, we present a hierarchical loop partitioning strategy that reduces the complexity of the search space for determining the runtime custom instruction configurations for reconfigurable processors. Experimental results show that the proposed partitioning strategy can lead to an average and maximum performance gain (in terms of clock cycle savings) of over 14% and 31% respectively when compared to a recently reported technique. In addition, when compared to the existing technique, the proposed partitioning method has significantly lower runtime in many of the cases considered.

Keywords: FPGA, Runtime reconfiguration, temporal partitioning.

1 Introduction

Emerging embedded applications for portable battery operated devices (e.g. mobile phones, PDAs, mobile gaming devices, etc.) necessitates computing platforms that are capable of meeting the increasing performance demands at low cost and low energy budget. At the same time, these computing platforms must maintain a high degree of flexibility to meet the shrinking time-to-market window. To this end, the instruction set extension capability of reconfigurable processors (e.g. [1]-[3]) provides an attractive means to meet these stringent requirements of embedded systems. A reconfigurable processor consists of a microprocessor core that is coupled with a RFU (Reconfigurable Functional Unit), which facilitates critical parts of the application to be implemented in hardware e.g. FPGA (Field Programmable Gate Array) in the form of custom instructions.

Runtime reconfiguration offers the potential to realize cost efficient systems that can still lead to high performance by changing the configuration of a small

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 282–293, 2010.

reconfigurable hardware at runtime. The two main drawbacks that discourage the use of runtime reconfiguration in embedded real-time systems is the large reconfiguration overhead in commercial FPGA architectures, and the lack of supporting tools and methodologies.

In this paper, we present a framework that rapidly generates custom instruction configurations for a given application and evaluate the benefits of runtime reconfiguration on a reconfigurable processor with an area-constrained RFU.

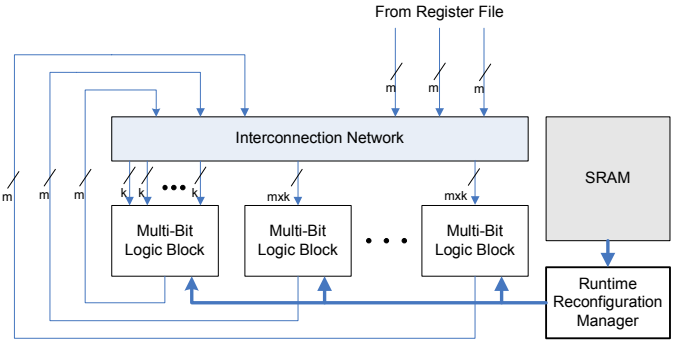


Fig. 1. Target architecture

The target RFU model in Figure 1, which is described in detail in [4], consists of a set of multi-bit logic blocks that is organized around an interconnection network. Each multi-bit logic block incorporates programmable fine-grained logic elements that are similar to those found in commercial FPGA architectures. However, unlike commercial architectures, the logic elements within each multi-bit block shares the same configuration memory, which leads to reduce runtime reconfiguration overhead. In this paper, we assume that the full reconfiguration model is adopted, i.e. each reconfiguration will result in new configurations loaded onto all the logic blocks in the RFU. If the computation resource requirement of the custom instructions exceeds the number of available logic blocks in the RFU, then the custom instructions are mapped to different configurations. At runtime, a reconfiguration manager automatically loads the required configurations onto the logic blocks for computing the custom instructions. The proposed runtime management scheme relies on the dynamic execution profile to replace the functionality of the logic blocks with the goal of minimizing the overall reconfiguration overhead (see [4]).

1.1 Related Work

The potential benefits of instruction set extension have led to numerous amount of research work that focuses on generating custom instructions from a given application (see [5] for a list of references). In order to select custom instructions for different runtime configurations, temporal partitioning must be performed to divide the design into mutually exclusive configurations such that the computational resource requirement of each configuration is less than or equal to the reconfigurable resource capacity of the RFU.

The task partitioning algorithm presented in [6] for minimizing the communication cost is achieved in two steps. In the first step, an initial partition is obtained by using a network flow based algorithm to produce a set of feasible mean cuts. In the second step, a scheduling technique is employed to select an optimal global solution.

The work in [7] employs ILP (Integer Linear Programming) to achieve near-optimal latency designs for temporal partitioning of the application task graph. A loop transformation strategy was used to maximize the throughput while minimizing the runtime reconfiguration overhead.

The framework presented in [8] automatically partitions loops to a target platform consisting of a processor, RFU and memory hierarchy. A hierarchical loop clustering strategy is used to partition a loop into smaller clusters in order to perform optimal hardware-software partitioning of the loop clusters. The loop clustering strategy traverses the hierarchical loop graph in a top-down fashion and recursively clusters the nested loops until the sizes of all the clusters are within a pre-defined limit. The current framework in [8] does not allow multiple loops in a single configuration.

The work in [9] describes an architecture-aware temporal partitioning strategy for mapping custom instructions on an adaptive extensive processor, which incorporates coarse-grained functional units. The strategy partitions and modifies custom instructions that violate the RFU constraints, in order to map them onto the RFU.

Recently, a framework was presented in [10] to select custom instruction versions to be mapped onto appropriate configurations. A custom instruction version consists of a set of custom instructions from a particular loop that satisfy the area constraint of the RFU. The partitioning scheme consists of temporal partitioning of frequently executed application loops with custom instructions into one or more configurations, and spatial partitioning to select an appropriate custom instruction version for each loop within a configuration. The temporal partitioning problem has been modeled as a k-way graph partitioning problem, and spatial partitioning is resolved using dynamic programming. The framework assumes the availability of the custom instruction versions and their corresponding hardware area-time measures. This necessitates time-consuming hardware implementation of the custom instructions prior to the partitioning process if no high level estimation strategy is in place. In the worst case, the temporal partitioning algorithm in [10] iterates l times, where l is the number of hot loops.

1.2 Our Work

This paper presents a framework that rapidly partitions loops, which constitute the most frequently executed segments of embedded applications, into configurations and selects profitable custom instructions in the respective configurations. This enables the benefits of runtime reconfiguration to be evaluated early in the design cycle without undergoing time consuming hardware implementation. Unlike methods in [8] and [10], the proposed strategy takes into consideration the nested loop paths, and is not restricted to only hot loops which will vary with the input data. This can potentially lead to higher performance gain by leveraging on the target architecture's capability to perform dynamic execution profiling for determining suitable configurations to be loaded onto the RFU at runtime. Unlike the framework in [10], our work does not generate custom instruction versions and their corresponding hardware area-time

measures prior to the partitioning process. Instead, we employ an efficient strategy that rapidly estimates the hardware area-time information of the custom instructions. In addition, the proposed framework relies on a hierarchical loop partitioning strategy that is similar to [8] for rapid partitioning of the application loops with custom instructions into one or more configurations. The proposed strategy utilizes efficient heuristics that takes into account the reconfiguration cost for partitioning loops into configurations. Finally, custom instructions for the respective configurations are then selected using a greedy algorithm. It is worth mentioning that the proposed method can be adopted in commercial FPGA tools as the target RFU model incorporates programmable logic elements that are similar to those found in commercial FPGA architectures.

2 Proposed Framework

Figure 2 gives an overview of the proposed framework. We have relied on the Trimaran compiler infrastructure [11] to generate the IR (Intermediate Representation) of the applications in the form of DFG (Data-Flow Graph). The IR serves as input to the Custom Instruction Generation stage to identify a set of custom instruction candidates. Details of the Custom Instruction Generation stage can be found in [5].

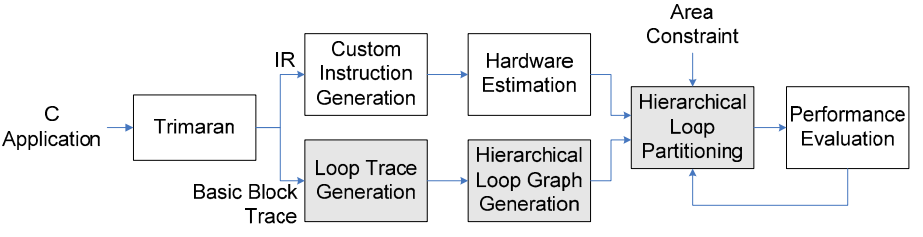


Fig. 2. Proposed framework

The hardware area-time information of the custom instruction candidates are then rapidly estimated without undergoing time-consuming hardware implementation. This step estimates the area costs and critical path delays of the custom instruction candidates when they are implemented on the multi-bit logic blocks of the RFU. In this paper, we target programmable logic elements similar to those found in the Xilinx Virtex device [12]. It is noteworthy that the hardware area-time results using the proposed estimation technique have been shown to be within 8% of those obtained using hardware synthesis. In addition, the hardware estimation can be achieved in the order of milliseconds. The details of the hardware estimation process can be found in [5].

A hierarchical loop graph is then generated to enable rapid temporal partitioning of loops using the proposed hierarchical loop partitioning strategy. Note that the partitioning strategy relies on the hardware estimation results to obtain a profitable set of custom instruction configurations.

Finally, performance evaluation is performed using Trimaran's simulation environment, which converts the IR into executable codes and emulates the execution on a virtual HPL-PD processor [11].

3 Hierarchical Loop Generation

The proposed partitioning strategy relies on a HLG (Hierarchical Loop Graph) representation of the application in order to reduce the complexity of the search space for determining the runtime configurations. Figure 3 shows the HLG representation of a CFG (Control Flow Graph). Each node in the HLG (except for the root node *R*) represents a unique loop in the CFG. For example loop L_1 consists of the nested loop path with basic blocks 1, 2, 3, 5, 7 and 8. A directed edge between two nodes *s*, *d* in the HLG, where *s*, *d* are at different HLG levels (*s* is not the root node), signifies that *d* is a nested loop of loop *s*. The nodes in the HLG are also associated with a value f_x, which denotes the execution frequency of the corresponding loop.

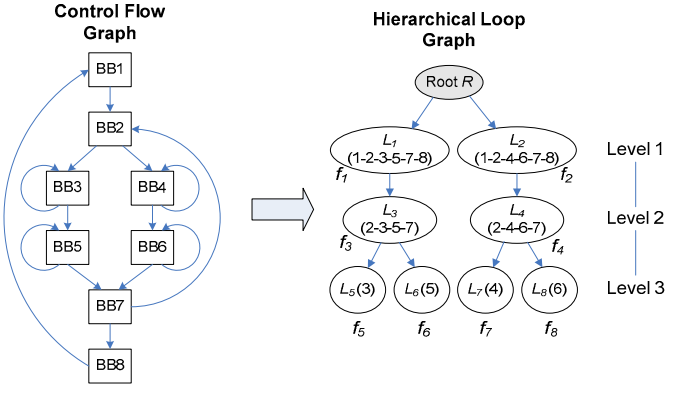


Fig. 3. Hierarchical Loop Graph of CFG

Due to the difficulty in identifying loops across function boundaries in the Trimaran CFG, our current framework constructs the HLG from the application loop trace, which is derived from the basic block trace (see Figure 2). The basic block trace records the dynamic execution flow of the application basic blocks. The loop trace can be derived from the basic block trace by reading each basic block entry in the trace file and checking if it is part of a new loop or existing loop. A loop is identified when a particular segment of a trace consists of duplicated basic blocks. New or existing loops can be determined by maintaining a history of loops that have been identified so far. Finally, information pertaining to how the loops are nested within one another and the execution frequency of each unique loop are determined from the loop trace to construct the HLG.

4 Hierarchical Loop Partitioning Strategy

The proposed hierarchical loop partitioning consists of two main tasks: 1) temporally partition the application loops based on the HLG into one or more configurations, such that the overall performance gain of runtime reconfiguration is maximized, and 2) select profitable custom instructions from the loops in each configuration. The final output of the algorithm is a set of configurations and the selected custom instructions in each configuration.

4.1 Temporal Partitioning

Figure 4 illustrates the proposed temporal partitioning strategy, where L_x is a node in the HLG and C_y denotes a configuration. The partitioning algorithm aims to reduce the search space by giving preference to loops at the lower levels in the HLG and considering the higher levels (nested loops) only when they are necessary. Figure 4(a) shows an example of the Level 1 nodes in the HLG. The dummy root node, which is the parent of the Level 1 nodes, is included for programming consistency.

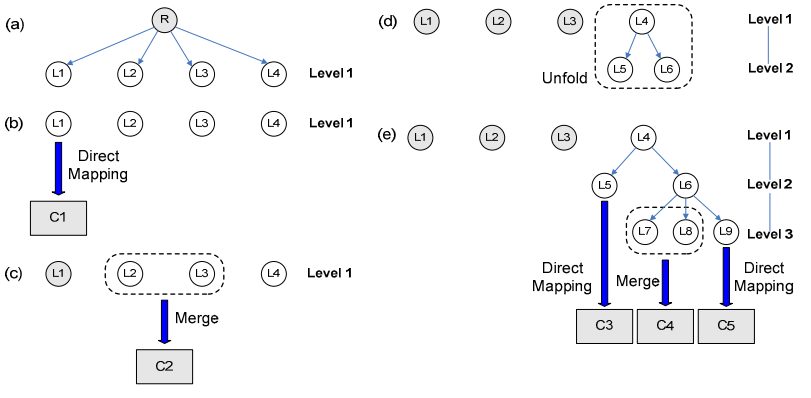


Fig. 4. Example of proposed temporal partitioning strategy

The partitioning strategy evaluates the first loop in Level 1 of the HLG (i.e. L_1) and found that it can directly map onto a configuration (see Figure 4(b)). The loop can be directly mapped onto the configuration if the logic requirement of the custom instructions in that loop can efficiently utilize the resource capacity of the configuration. The partitioning algorithm then moves on to evaluate the next loop in Level 1 without the need to consider the nested loops of L_1. In the event that the custom instructions of a loop under-utilizes the resource capacity of the configuration, it is considered for merging with the next loop on the same level. This is shown in Figure 4(c) where L_2 and L_3 are merged into a single configuration. Note that the nested loops of L_2 and L_3 are not evaluated further. However, if the logic requirement of the custom instructions in the loop (e.g. L_4 in Figure 4(d)) is larger that the resource capacity of a configuration, an unfolding process may take place to allow the nested loops to be evaluated. It can be observed that only the immediate nested loops (or child nodes in the next

higher level) are unfolded at a time. The evaluation process for direct mapping, merging and unfolding is then performed on these nested loops. For example in Figure 4(e), it can be observed that the nested loop L_5 can be directly mapped to a configuration, while the nested loop L_6 is further unfolded to the next higher level. In subsequent iterations, some of the nested loops of L_6 are merged into a configuration (i.e. L_7 and L_8), while others are directly mapped onto a configuration L_9). The process is repeated until all the loops in Level 1 of the HLG have been evaluated.

Figure 5 shows the algorithm for temporal partitioning. The temporal partitioning strategy consists of three main processes: 1) unfolding, 2) merging, and 3) direct mapping. Heuristics are employed to determine which of these three processes that the loop will be subjected to (i.e. line 3, 7 and 11). These heuristics are based on comparing the estimated area of all profitable custom instruction candidates in loop x, with a constant that is a product of the resource capacity of the configuration (i.e. A) and a pre-defined factor (i.e. **u** or **m**). The output of the temporal partitioning algorithm is a configuration set C, where each configuration in the set is associated with one or more loops in the HLG.

Let's first define a profitable custom instruction candidate i in loop x as one that satisfy the constraint in (1), where g_i^x is the gain of the profitable custom instruction candidate and t_{lb} is the reconfiguration time of a single multi-bit logic block, in terms of number of software clock cycles. a_i is the estimated area of i that is obtained using the method discussed in [5]. g_i^x is calculated as shown in (2), where f_x is the execution frequency of loop x, and n_i^x is the number of software clock cycles for i. We assume that each operation in the Trimaran IR takes one software execution clock cycle.

$$\frac{g_i^x}{a_i} > 2 \times t_{lb} \tag{1}$$

$$g_i^x = f_x \times n_i^x \tag{2}$$

The constraint in (1) ensures that only custom instruction candidates that can still lead to notable performance gain after taking into account the runtime reconfiguration overhead are considered as profitable custom instructions. Line 2 in Figure 5 shows the heuristic used to evaluate if the current loop x needs to be unfolded to the next higher level in the HLG. The unfolding process can only take place when the following conditions are satisfied. Firstly, given a configuration area A, the estimated area of all the profitable custom instruction candidates ($A_x = \sum_{i}^{n\_pro} a_i$, where n_pro is the number of profitable custom instructions in loop x) must be larger than **u** x A, where **u** is a pre-defined unfolding factor ($1 \le u < 2$).

Secondly, in order for the loop to be unfolded, it must contain nested loops (see line 18). If these conditions are met, the unfolding process recursively calls the temporal partitioning algorithm, where the immediate nested loops of loop x will be unfolded (line 19). If the first condition is met but the second condition is violated, loop x is directly mapped onto a single configuration (line 21).

```
Procedure Hierarchical_Loop_Partitioning (HLG, Ci)
Input:  Hierarchical Loop Graph (HLG),
        custom instruction candidates (Ci)
Result: Selected custom instructions for each
        configuration (Sel_CI)

1. configuration set C = empty;
2. Temporal_Partitioning (root);
3. Sel_CI = Cusom_Instruction_Selection (C, Ci);
4. return;

Procedure Temporal_Partiion (node)
Inputs: HLG, configuration set (C), configuration area (A),
        area of profitable custom instructions for each
        loop x in HLG (Ax)
Output: Set of configurations C

1.  for each loop x that is a child of node
2.      if Ax > u*A then
3.          Unfold_Loop (x);
4.      else if Ax < m*A then
5.          Put loop x into stack for merging;
6.      else if m*A < Ax < u*A then
7.          Direct_Map (x);
8.      end if
9.  end for
10. if stack is not empty then
11.     Merge_Loop (stack);
12. end if
13. Initialize new configuration c and insert node in c;
14. if Compute_Effective_Gain (c) >
        Compute_Effective_Gain (∀ci ∈ C with nested
        loops of node) then
15.     Replace ci with c in C;
16. end if
17. return;

Procedure Unfold_Loop (x)
18. if loop x has nested loops in HLG then
19.     Temporal_Partition (x);
20. else
21.     Direct_Map (x);
22. end if
23. return;

Procedure Direct_Map (x)
24. Initialize new configuration c and insert loop x in c;
25. if Compute_Effective_Gain (c) > 0 then
26.     Insert c as a new configuration in C
27. end if
28. return;

Procedure Merge_Loop (stack)
29. Initialize new configuration c with unutilized area Ac = A;
30. for each loop x in stack
31.     if Ac > Ax then
32.         Include loop x in configuration c;
33.         Ac = Ac – Ax;
34.     else
35.         if Compute_Effective_Gain (c) > 0 then
36.             Insert c as a new configuration in C;
37.         end if
38.         Initialize new configuration c with unutilized
            area Ac = A - Ax;
39.         Include loop x in new configuration c;
40.     end if
41. end for
42. if c != { } && Compute_Effective_Gain (c) > 0 then
43.     Insert c as a new configuration in C;
44. end if
45. return;
```

Fig. 5. Algorithm for temporal partitioning

The heuristic that is used to consider if loop x should be merged with other loops in the same nested level of the HLG is shown in Line 4 of Figure 5. In particular, if the estimated area of all the profitable custom instruction candidates in loop x is less than **m** x A, where **m** is a pre-defined merge factor ($m < 1$), then the loop will be pushed into a stack to be considered for merging (Line 5). Note that these loops have been inserted into the stack as each of them will under utilize the logic capacity of the RFU.

When all the loops in a particular level have been considered, the loops that are inserted in the stack are partitioned into configurations. The merging process (line 29-44) employs a greedy approach to merge the loops in the stack until the total estimated area of the profitable custom instruction candidates in the merged loops exceed the given configuration area A (line 31-33). When the current configuration cannot accommodate a loop in the stack due to the area constraint, a new configuration is created for the loop (line 38-39). A configuration of merged loops is considered as a valid configuration only if the effective gain of the configuration is larger than 0 (line 35 and 42).

The effective gain of configuration c (G_c), that is calculated using the Compute_Effective_Gain function, is the total gain of all the profitable custom instruction

candidates in the loops of c that have been greedily selected by taking into account the runtime reconfiguration overhead. G_c is calculated as shown in (3), where n_rtr_c is the number of times configuration c will be reconfigured in the application and n_ml is the number of loops that have been merged in configuration c. n_rtr_c can be determined from the loop trace. t_{config} is the overhead to reconfigure all the logic blocks (i.e. $t_{config} = n\_lb \times t_{lb}$, where n_lb is the number of logic blocks). The merging process terminates when all the loops in the stack has been considered for merging.

$$G_c = \sum_{x}^{n\_ml} \sum_{i} g_i^x - (t_{config} \times n\_rtr_c) \tag{3}$$

The heuristic in Line 6 of Figure 5 is used to determine if loop x can be directly-mapped to a single configuration. In particular, loop x can be directly mapped to a single configuration if the estimated area of all the profitable custom instruction candidates in loop x is 1) larger than **m** x A, and 2) less than **u** x A. It can be observed that similar to the merging process, a configuration is valid only if its effective gain is larger than 0 (line 25-26).

Note that the algorithm may eventually discard the configurations resulting from the unfolding process if it does not lead to higher performance gain (lines 14-15).

4.2 Custom Instruction Selection

In the temporal partitioning process, we have identified a set of configurations and their associated loops. The next step in the proposed partitioning strategy is to select custom instructions for each of the configurations. This is achieved by employing a greedy algorithm to select custom instructions with the highest gain in each configuration such that the total area required by the selected custom instruction does not exceed the logic capacity of the RFU. The gain of the custom instructions in the loops associated with the configuration is first estimated using (2) and sorted in decreasing gain. The greedy algorithm then selects the custom instructions by giving preference to those with the highest gain in the sorted list such that the area of the selected custom instructions does not exceed the logic capacity. This process is repeated for all the configurations.

5 Experimental Results

In order to evaluate the benefits of the proposed partitioning strategy, we have employed six widely-used embedded benchmarks from [13]-[15]. We compare the proposed hierarchical partitioning strategy (denoted as *Hierarchical*) with a recently reported iterative method [10] (denoted as *Iterative*). Both methods perform temporal partitioning of the application loops and selection of custom instructions from the temporal partitions. For the Hierarchical method, we have empirically determined suitable values of **u** and **m** to be 1.2 and 0.6 respectively. We also assume that t_{lb} = 3K clock cycles (similar to the configuration time of one hardware unit in [10]). We have employed the full basic block trace for all the applications considered except for mpeg2 enc, where a partial basic block trace file is used to generate the loop trace and

HLG as the original basic block trace file is very large. Similar to [10], we assume that the hardware area constraint is about 20-30% of the maximum hardware area that is required to accommodate all the custom instructions such that runtime reconfiguration is not necessary.

Table 1 compares the performance gain of the two methods. The performance gain is measured in terms of the clock cycle savings that resulted from instruction customization after taking into account the runtime reconfiguration cost. It can be observed that Hierarchical outperforms Iterative in all cases. This is due to the fact that Hierarchical takes into consideration the nested loop paths and is not restricted to hot loops. Hierarchical achieves an average and maximum performance gain of over 14% and 31% respectively when compared to Iterative.

Table 1. Comparison of performance gain

Clock Cycle Savings (K Cycles)			
	Iterative	**Hierarchical**	**%Gain**
Adpcm Enc	2030	2056	1.28
Cjpeg	1993	2310	15.93
Mpeg2 Enc	2917	3629	24.41
Virterbi00	173640	184695	6.37
Epic	998	1053	5.51
Sha	44309	58458	31.93

Table 2. Comparison of partitioning runtime

Partitioning Runtime (seconds)		
	Iterative	**Hierarchical**
Adpcm Enc	1.43	0.01
Cjpeg	6.61	0.05
Mpeg2 Enc	6.62	0.10
Virterbi00	0.29	0.32
Epic	0.27	0.27
Sha	0.04	0.05

Table 2 compares the partitioning runtime between the two methods. For the proposed Hierarchical method, the runtime is measured for the tasks described in Section 4. Similarly, the runtime for the Iterative method is only measured for temporal and spatial partitioning. The time taken to generate the custom instruction versions and the corresponding hardware area-time measures in [10] is not considered. Both methods rely on the custom instruction generation process discussed in [5], and hence the time taken to identify the custom instructions is not considered in the comparison.

It can be observed that the runtime of Hierarchical is significantly lower than Iterative in most cases (i.e. Adpcm Enc, Cjpeg and Virterbi00), and comparable in the remaining cases (the difference in runtime in these remaining cases is less than 0.04s).

This is due to the fact that the proposed strategy significantly reduces the search space of the loop partitioning process by evaluating the nested loops only when the profitable custom instructions in the corresponding larger loops cannot be mapped entirely onto the RFU area.

6 Conclusion

We have presented a framework for reconfigurable processors that employs a hierarchical partitioning strategy which aims to maximize the performance of custom instruction realization through runtime reconfiguration, while minimizing the reconfiguration overhead. The proposed hierarchical partitioning strategy heuristically determines whether the loops in the HLG can be directly mapped to a configuration, merged with other loops or unfolded to the nested loops. Nested loops are only evaluated when the profitable custom instructions in larger loops cannot be mapped entirely onto a restricted RFU area. This strategy significantly reduces the search space of the partitioning process, resulting in rapid identification of temporal partitions. A greedy algorithm is then used to select custom instructions in each temporal partition. Experimental results show that the proposed hierarchical partitioning strategy can lead to a higher performance gain than a recently reported iterative partitioning approach. In addition, the partitioning runtime of the proposed partitioning strategy is significantly lesser than the iterative method in many of the cases considered. This enables rapid design exploration for maximizing the utilization of reconfigurable space through runtime reconfiguration. Future work includes devising a method to generate the HLG from the application CFG and profiling information from Trimaran, as generating the HLG from the basic block trace is not feasible when the trace file is too large.

References

1. Altera: NIOS II Processors, http://www.altera.com/products/ip/processors/nios2/ni2-index.html
2. Xilinx Platform FPGAs, http://www.xilinx.com
3. Video/Imaging Design Line: Analysis: Stretch's Second-Gen Configurable Processor (2007), http://www.videsignline.com/howto/videoprocessing/201311209
4. Lam, S.K., Huang, F., Srikanthan, T., Wu, J.: Run-Time Management of Custom Instructions on a Partially Reconfigurable Architecture. In: IEEE International Conference on Electronic Design (2008)
5. Lam, S.K., Srikanthan, T.: Rapid Design of Area-Efficient Custom Instructions for Reconfigurable Embedded Processing. Journal of Systems Architecture 55(1), 1–14 (2009)
6. Jiang, Y.C., Wang, J.F.: Temporal Partitioning Data Flow Graphs for Dynamically Reconfigurable Computing. IEEE Transactions on Very Very Large Scale Systems 15(12), 1351–1361 (2007)
7. Kaul, M., Vemuri, R., Govindarajan, S., Ouaiss, I.: An Automated Temporal Partitioning and Loop Fission Approach for FPGA based Reconfigurable Synthesis of DSP Applications. In: Design Automation Conference, pp. 616–622 (1999)

8. Li, Y., Callahan, T., Darnell, E., Harr, O., Kurkure, U., Stockwood, J.: Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In: Design Automation Conference, pp. 507–512 (2000)
9. Mehdipour, F., Noori, H., Zamani, M.S., Murakami, K., Sedighi, M., Inoue, K.: An Integrated Temporal Partitioning and Mapping Framework for Handling Custom Instructions on a Reconfigurable Functional Unit. In: Asia-Pacific Computer Systems Architecture Conference, pp. 219–230 (2006)
10. Huynh, H.P., Sim, J.E., Mitra, T.: An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization. Design Automation for Embedded Systems (2008)
11. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism, `http://www.trimaran.org`
12. Xilinx Data Sheet: Virtex 2.5V FPGA Detailed Functional Description, DS003-2, Version 2.8.1 (2002)
13. The Embedded Microprocessor Benchmark Consortium: `http://eembc.org`
14. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 330–335 (1997)
15. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: IEEE International Workshop on Workload Characterization, pp. 3–14 (2001)

Reconfigurable Computing and Task Scheduling for Active Storage Service Processing

Yu Zhang and Dan Feng

Wuhan National Lab for Optoelectronics, School of Computer Science,
Huazhong University of Science and Technology
Wuhan, China
yuzhang13@gmail.com, dfeng@hust.edu.cn

Abstract. Active storage is a promising solution for data processing. Many existing active storage systems utilize spare CPU processing power on the storage controllers to execute active service. The software approach is very flexible for various applications, but the performance is quite low for computational-intensive service tasks. This paper presents a low power SoC-based hardware solution for high performance active storage. The basic idea is incorporating flexible reconfigurable accelerators in storage controllers for efficient active service processing. In order to reduce the reconfiguration latency, we also proposed hybrid configuration prefetching and configuration caching algorithms according to the task pattern. In the experiments, we presented results on applications such as data security, data compression and image processing with our FPGA prototype for an active storage processor.

Keywords: reconfigurable computing, active storage, configuration prefetching, configuration caching.

1 Introduction

Driven by Moore's law, the computer chip has evolved from a simple integrated circuit to a SoC (System on Chip) with millions of transistors. The benefits for such ongoing integration are smaller form factor, lower power and higher performance. Storage controllers such as RAID, NAS and SAN storage controllers are the key hardware components for data center and enterprise information systems. It becomes a trend that more newly designed storage controller chips have utilized SoC technology. Examples are Intel Tolapai EP80579 SoCs (13-20w) for NAS applications, and D-Link iSNP8000 SoCs (less than 66w) for iSCSI storage.

Active storage means the storage system can process data by itself besides merely storing data. Moving computation close to data has several advantages: 1) reduce server overhead, and reduce data traffic between the storage system and the host server; 2) increase performance by running applications on multiple storage devices in parallel; 3) achieve large power savings [1]. Research works on active storage system include CMU's Active Disks for data mining and multimedia processing [2]; Intel

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 294–305, 2010.

lab's Diamond system for interactive search through non-indexed data [3]; and Texas A&M University's MVSS for data security and data processing [4].

The previous research efforts on active storage are based on a premise that modern storage architectures have progressed to a point that there is the real possibility of utilizing unused processing power of the storage devices. If the storage controller uses high performance general purpose processor, it makes sense. While using less powerful embedded processors on SoCs, it's quite difficult for the storage system to perform I/O and computational-intensive tasks like data encryption/decryption and multimedia processing with good efficiency.

Two solutions can be used to incorporate more computation power into SoCs: multi-processors and dedicate hardware accelerators. But neither way is very suitable for the active storage controllers that require both computation power and considerable flexibility for various applications.

For MPSoC, 1) The architecture must be tailored for the application to meet the constraints such as power, performance and cost, so it's hard to define a general MPSoC architecture for a wide range of active service; 2) Some functions require operations that do not map well onto a CPU's data operations. For instance, bit-level operations are difficult to perform efficiently on some processors; 3) highly responsive input and output operations may be best performed by an accelerator with an attached I/O unit [5].

The hardware accelerator has good performance, but the flexibility is very poor. We cannot modify or update any service function after the active storage system is running. We will also meet problems when a large number of accelerators need to be integrated into the SoC because of limited expansion capability.

Reconfigurable computing refers to systems incorporating some form of hardware programmability. Dynamic reconfiguration, also known as run-time reconfiguration, uses a dynamic allocation scheme that re-allocates hardware at run-time. It can increase system performance by using highly-optimized circuits that are loaded and unloaded dynamically during the operation of the system. Reconfigurable computing provides the flexibility of software processors and the performance of dedicated hardware processing engines [6].

The basic idea of this paper is to integrate reconfigurable accelerators into the storage SoC controller for high performance active storage. Our hardware system can be used in all existing active storage frameworks with two modifications: 1) use dedicated hardware processing rather than software processing; 2) the service is dynamically added by circuit reconfiguration, while in traditional systems a new service is added by loading a piece of code (filter applet) onto devices. Other aspects such as application interface and device level service binding mechanism will remain unchanged. When user requests arrive, the storage controller must first interpret the received I/O commands associated with certain services, then access data on the disk drives and perform computation. In this paper we mainly focus on data processing with reconfigurable hardware, and other software issues for active storage system designs such as device model, user application interface and service binding will not be discussed in detail.

We chose Xilinx Virtex-5 FPGA which supports Partial Run-Time Reconfiguration (PRTR) as the hardware platform and built our prototype design. This paper addressed two practical design issues as follows: 1) Implement reconfigurable accelerators on standard SoC; 2) Propose optimized configuration prefetching and configuration caching algorithms according to active storage service processing pattern.

In real designs, FPGA devices may be too expensive and consume too much power. To solve these problems, reconfigurable accelerators can be implemented on the programmable gate array area of the SoC chip, and the rest static parts of the storage processor can be implemented by low-power ASIC technology on the same chip. Many reconfigurable computing systems (e.g. GARP, Chimaera and Morphosys) have shared this idea. In this paper we used FPGA device to study the reconfigurable hardware architecture and related configuration management issues.

The paper is organized as follows: In section 2, we stated how to implement reconfigurable accelerators based on standard SoC architecture. In section 3, we presented optimized configuration prefetching and configuration caching algorithms to reduce configuration latency when multiple accelerators were available in the SoC. In section 4 we built our prototype system and conducted experiments using some applications as case studies. Finally in section 5 we concluded this paper.

2 Reconfigurable Active Storage Processor Design

There are two ways to couple a dedicated hardware processing engine into the Xilinx SoC: 1) a coprocessor; 2) an accelerator. The coprocessor can be viewed as the function expansion block of a processor's ALU module. Software controls a coprocessor by using some special instructions. In Xilinx Virtex-5 FPGAs, We can connect a coprocessor with a Microblaze processor via FSL interface, and we can also couple a coprocessor to an on-chip PowerPC core. The efficiency of the coprocessor is directly determined by the performance of the processor. The drawback is that the processor must take the responsibility of data transfer during processing. The coprocessor system is suitable for computational-intensive applications, but it's not the right choice for data-intensive application. Most active storage service deals with data files, and a large amount of data should be transferred between the coprocessor and the memory, and this will cause a heavy CPU load.

When a dedicated processing engine is connected on the I/O bus and acts as a peripheral, we call it a hardware accelerator. Software controls an accelerator by writing its registers which mapped into a certain memory space location. Xilinx SoC is based on IBM CoreConnect bus architecture. The accelerator is attached on the PLB bus, and a central DMA controller can be used for data transfer between the memory and the accelerator. In this paper we choose accelerator scheme for active service processing. A SoC prototype design for active storage processor is illustrated in Fig. 1, and a hardware accelerator is integrated in the system.

We used the application of 128-bit AES encryption to study the performance of the accelerator scheme. Our experiments showed that the Xilinx SoC with a 400MHz PPC440 could only get a throughput of 3.7MB/s by running Optimized ANSI Code for the Rijndael cipher (standalone program) [7].

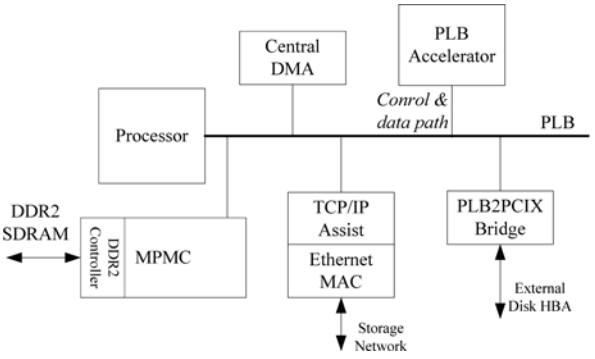


Fig. 1. Storage Processor for Network-Attached Storage Application

The AES encryption is comprised of a variable number of rounds (determined by the key and block lengths) with each round containing four stages: ByteSub, ShiftRow, MixColumn and RoundKeyAddition. The cipher core performs a complete encrypt sequence on a 128-bit word in 22 clock cycles (20 cycles for the 10 rounds, plus 1 cycle for initial key expansion, and 1 cycle for data output). When running at the frequency of 100MHz, this core can reach a maximum throughout of 73MB/s. In the experiments, we have the following hardware settings to test the accelerator performance:

a) Microblaze processor v7.10d (with 8KB I/D Cache), 100MHz
b) PLB, 100MHz
c) Accelerator, 100MHz
d) DDR2 SDRAM, 200MHz

Running a standalone control program (without OS), the AES accelerator reached a sustainable throughput of 16.28MB/s, which is 66 times faster than a 100MHz Microblaze and 4.4 times faster than a 400MHz PowerPC440 processor.

A SoC design with a single reconfigurable accelerator is shown in Fig. 2, and there are some essential components in the system: HWICAP is for chip reconfiguration; SystemACE controller is used to read bit-streams in the compact flash; XPS_LL_SOCKET acts as a virtual peripheral to declare certain memory space mapping for the RPU (Reconfigurable Processing Unit) during static SoPC design phase. Dynamic modules and statics modules must be connected via special bus macros. Before reconfiguration user software uses GPIO to disable the bus macros to disconnect the RPU from the system and enable them when the reconfiguration is over. In real applications we can integrate multiple reconfigurable accelerators into the system to processing application in parallel.

A side-effect of reconfigurable computing technology is the large reconfiguration latency. According to our experiments, an AES-128 encryption module with the bitstream size of 178KB needed 12.8 ms to load the configuration data from DDR2 memory to FPGA device. Reconfiguration latency will enormously downgrade the

system performance when dynamic hardware modules switch at high frequency. With the growing of the Internet, the total data volume becomes larger and larger. Among all the data, most of them are small files (less than 1MB) like emails, text and JPEG images. When storage controllers use reconfigurable computing hardware to process these small data, configuration latency cannot be ignored. Related techniques to reduce reconfiguration latency are configuration compression, configuration data reuse and configuration scheduling [8]. In this paper we mainly focused on configuration caching and configuration prefetching.

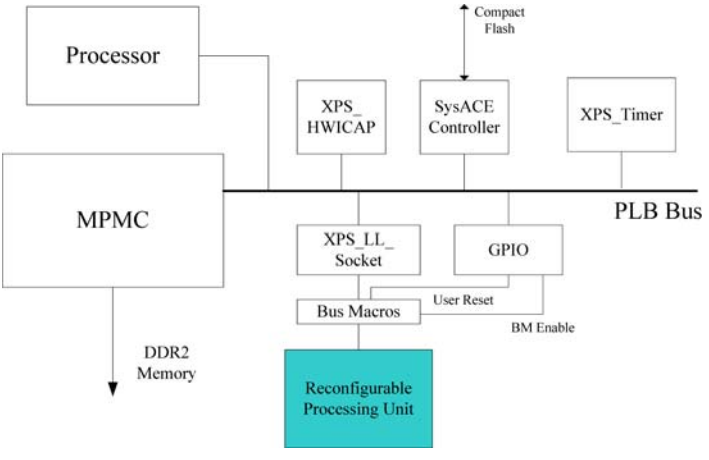


Fig. 2. Reconfigurable Accelerator Design

3 Configuration Scheduling for Active Storage Service

Configuration prefetching and caching are well known techniques for reducing reconfiguration latency when multiple RPUs are available. Before moving deeper into the details, we would first study the active service processing pattern. A service task that sent to the active storage controller always deals with a certain file, and a task often contains several subtasks (also called filters). We define each subtask as a RFUOP (Reconfigurable Function Unit Operation), and in our system a RFUOP can be placed (configured) on any RPU. The active service processing pattern has the following characters:

a) Multiple tasks can be processed in parallel.
b) Subtasks within a task should be processed in sequence.

Configuration scheduling (prefetching and caching) should be based on not only the task processing pattern, but also the reconfigurable hardware system. In this paper we assume that each RPU is big enough to hold any single RFUOP in the system. The task scheduling algorithm is shown in Fig. 3.

```
if  (there are tasks in the queue)
A1:     allocate a RPU for each task according
        to First In First Served principle;

        if (number of available RPUs = 0)
A2:             Tasks in the back of the queue should wait.
                if (other tasks are finished, and there are idle RPUs)
                        goto A1;
        else
A3:             Apply static configuration prefetching to the
                subsequent subtasks of higher priority tasks;

// The queue is empty, but NO idle RPUs are available;
if (a new task arrives and there are suspended static prefetched RFUOPs)
        calculate the remaining waiting time of each suspended RFUOP;

        if (the configuration time of the new task <
           remaining waiting time)
B1:             cancel the static prefetched RFUOPs
                and allocate its RPU for the new task;
        else
                goto A2;

if (the queue is empty and idle RPUs are available)
C1:     apply  Markov dynamic configuration prefetching
        for the last task.
```

Fig. 3. Pseudo Code for Task scheduling algorithm

3.1 Configuration Prefetching

There are two kinds of configuration prefetching techniques: static and dynamic prefetching. Static configuration prefetching improves the performance by overlapping configuration process and computation process [9]. For example, a task with two RFUOPs arrives, and RFUOP-1 must be finished before RFUOP-2 can start. While RFUOP-1 is running on RPU-A, static prefetching configures RPU-B as RFUOP-2 in advance, thus the reconfiguration latency can be hidden before RFUOP-2 starts. We can apply static prefetching to any task with multiple sequential subtasks, and this technique always speeds up a single task's execution. When several tasks are to be processed in the queue, static configuration prefetching may not be feasible. Let's use the above example again, suppose the execution time of RFUOP-1 is much longer than the configuration time of RFUOP-2. When static configuration prefetching happens, RPU-B will be kept suspended for quit a long time before RFUOP-1 finishes. Precious computational resource which can be used for other tasks is wasted. In real applications, it's a common case that execution time is longer than configuration latency, and static prefetching should take place if there are idle RPUs after each task has be allocated a RPU (as in clause A3 of the pseudo code in Fig. 3) , thus we can get better overall performance.

Another technique to reduce the reconfiguration latency is dynamic configuration prefetching. Dynamic configuration prefetching tries to predict the upcoming task and configure its RFUOP before it arrives. Dynamic configuration prefetching happens when the task queue is empty and there is an idle RPU on the chip (as in clause C1 of the pseudo code in Fig. 3). We should always apply dynamic configuration prefetching if possible. This can help to improve the performance if we can "guess" accurately. Markov prefetching is a very unique approach in general-purpose computing systems, and it can also be applied to configuration prefetching. The tasks can be represented as the vertices in the Markov graph and the transitions can be built and updated using the task access sequence. A simplified mechanism is used to update the state transition probability. For each occurrence of state transition (x, y), the probability of state transition (x, z) is updated as:

$$P_{x,z} = P_{x,z} / (1+C), z \neq y$$
$$P_{x,z} = (P_{x,z} + C) / (1+C), z = y \quad (1)$$

In (1) C is the training step size, or learning rate, which determines how fast the predictor adapts to changes in the execution pattern [10]. We can set C to 1 for simplifying the calculation. When dynamic prefetching happens, the first RFUOP in the task that has the biggest probability value should be loaded onto the chip.

Our proposed hybrid algorithm that combines static and dynamic configuration prefetching tries to consider both overall efficiency and the single task's performance.

3.2 Configuration Caching

In clause A1, A3 and C1 of the pseudo code in figure 3, if several idle RPUs are available for a new RFUOP, we need to swap out a victim RFUOP according to the configuration caching algorithm. Caching configurations on an FPGA is similar to caching instructions or data in a general memory. It retains the configurations that will be most-likely used in the future so the amount of configuration data that needs to be transferred to the chip can be reduced. But configuration caching is different from general caching. In reconfigurable systems, the loading latency of configurations may vary due to non-uniform configuration sizes of different RFUOPs. So decision should be based on not only the used frequency of a RFUOP but also its configuration size.

When configuration replacement occurs and there are tasks/subtasks waiting in the queue, we can apply the general offline caching algorithm [11] to choose the victim RFUOP (similar to the Belady algorithm). The configuration manager maintains a window which starts at the current reconfiguration, and contains an occurrence of all currently loaded RFUOP configurations. Fig. 4 illustrates how to setup a reappearance window for Task 1 and Task 2 in the queue. According to the general offline caching algorithm, if a RFUOP need to be swapped out, for each candidate, we multiply the loading latency and its number of appearances in the window, and then replace the RFUOP with the smallest value.

When configuration replacement happens and there are no tasks/subtasks waiting in the queue, we should apply the online caching algorithms. Comparing with the history based the LRU algorithm; the penalty based algorithm [12] is more suitable for the reconfigurable computing systems. The penalty based algorithm used a variable "credit" to determine the victim. Every time an RFUOP is loaded onto the chip,

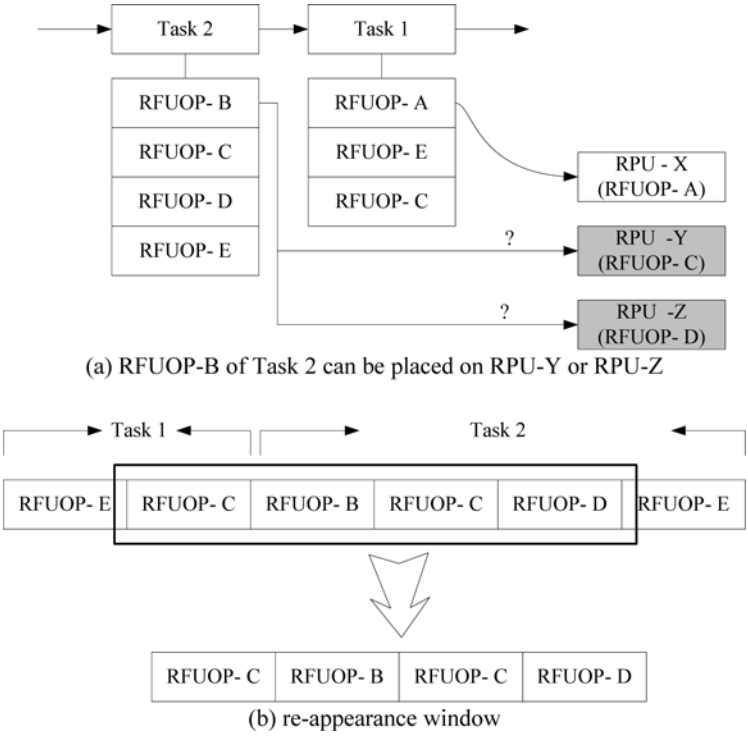


Fig. 4. How to setup the re-appearance window

its credit is set to its size. When a replacement occurs, the RFUOP with the smallest credit is evicted from the chip, and the credit of all other RFUOPs on-chip is decreased by the credit of the victim. The penalty based algorithm isn't suitable for active storage service processing, and this is because the upcoming RFUOP should always be the first RFUOP of the new task. In this paper a modified penalty based algorithm is presented and it is described as follows:

a) Each kind of Task has a credit. Every time the first RFUOP of the task is loaded in the chip, the task's credit is set to the size of its first RFUOP.
b) When a replacement occurs, the victim is chosen according to these two rules:
   b.1 The chosen RFUOP victim should belong to the task that has the smallest credit.
   b.2 If two or more RFUOPs from the same task can be evicted in rule b.1, the RFUOP in the back of RFUOP sequence will be evicted.
c) When the RFUOP is evicted from the chip, the credit of all other tasks on-chip is decreased by the credit of the task contains this victim.

4 Experiments

In the experiments, we built the prototype design on Xilinx XUPV5-LX110T development board. The SoC design contained a Microblaze processor which ran

XilKernel OS, and several reconfigurable accelerators. The system used a 4GB Compact flashcard to emulate a hard disk and a UART interface to communicate with a client machine that sent user service requests. We used applications like AES encryption/decryption, LZRW3 lossless data compression/decompression and image processing (edge detection) to study the performance. The following content lists the service models for active storage systems; the symbolic description uses W for writes and R for reads. The left side of the symbolic name specifies input and the right side specifies output.

a) 1W->2W: Data will be written to the original file, then another new file is created that will receive the data after it has been processed.
b) 1W->1W: Data are processed then written to the original file.
c) 1R->1W: Data previously stored on the storage device are re-processed into a new file.
d) 1R->1R: Data are processed, and then the output is sent to the reading process.

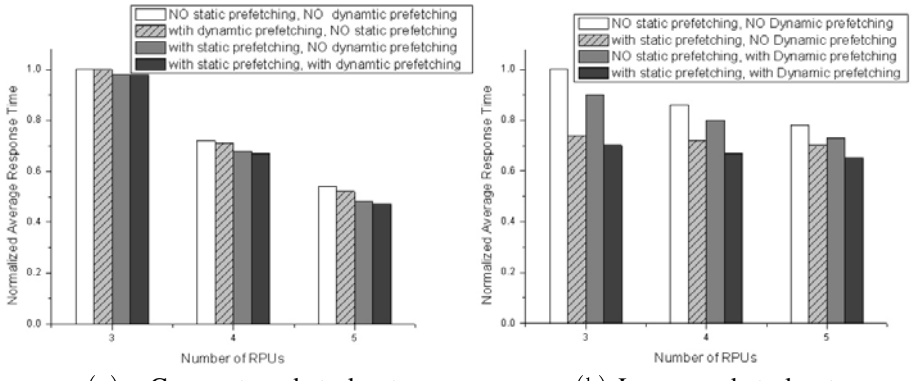
In our experiments, we always used 1R->1W model and user requests were sent to our prototype storage controller via UART interface. The 1R-> 1W pattern combines the following service tasks as shown in Table 1.

Table 1. Task Description

Active Service Tasks	Descriptions
AES128/256 encryption	Provide data security on storage media or data transfer between storage devices and clients
LZRW3 compression	Save disk space or reduce the data transfer between storage devices and clients
AES128/256 decryption	Convert ciphertext on disks to plain text for users
LZRW3 decompression	Decompress data on the disks for users
LZRW3 compression -> AES128/256 encryption	Save disk space or reduce the data transfer between storage devices and clients, and provide data security
AES128/256 decryption -> LZRW3 decompression	Restore the encrypted and compressed data on disks for users
Image graying -> Smoothing -> Sobel edge detector	Image edge detection - Colorful BMP images are grayed and smoothed, and then processed with the Sobel algorithm.

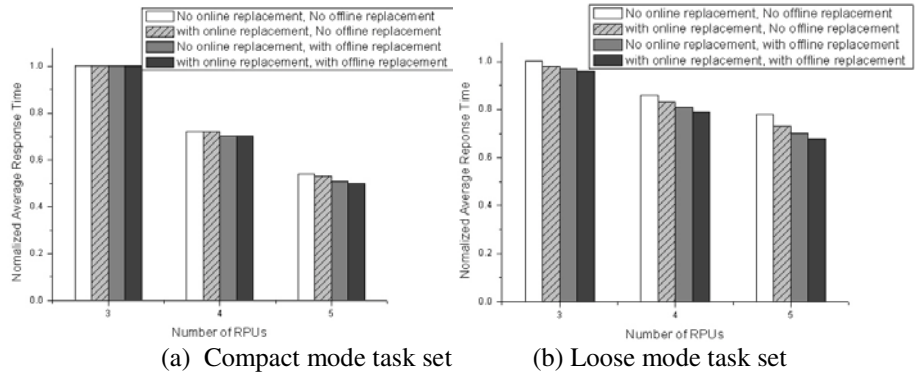
According to Table 1 there are totally 9 RFUOPs and 11 kinds of tasks in our experiments. The client generated the same two sets of mixed workloads (each task was repeated 5 times, totally 55 requests) with two modes respectively: 1) Compact mode – In most cases there is little time interval between two consecutive tasks. 2) Loose mode - On the contrary, in loose mode most tasks are not so tightly scheduled as in compact mode. In the experiments we used average task response time (indicate how fast a service can be served) as the metric to measure the performance.

Here we used small files (less than 0.5MB) as objects of the experiments, and the RFUOP's reconfiguration latency was comparable to its execution time and tasks were made highly dynamic. We first conducted experiments to see the effect of configuration prefetching, and in these tests configuration caching were not applied. Fig. 5 shows the results. For loose mode task set, static prefetching was very effective, particularly when a small number of RPUs were intergraded. For compact mode task set, the number of RPUs always dominated the overall performance, as the number of RPUs increased both static and dynamic prefetching contributed more to system performance. This is because the chance for prefetching also increased.



(a) Compact mode task set (b) Loose mode task set

Fig. 5. Effect of hybrid configuration prefetching

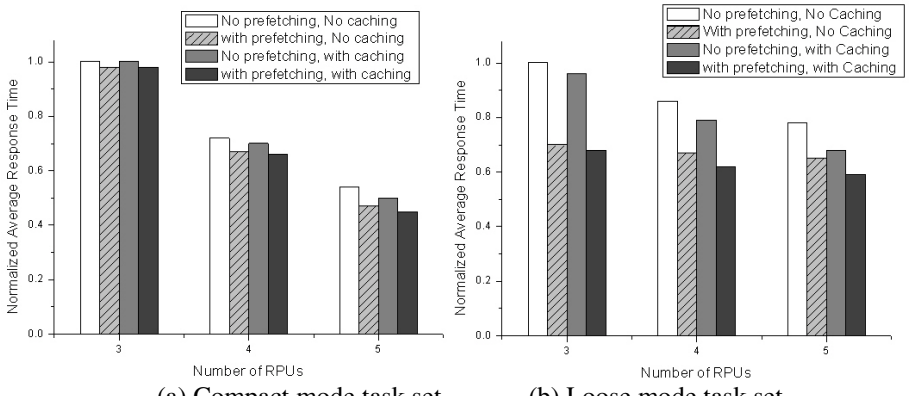


(a) Compact mode task set (b) Loose mode task set

Fig. 6. Effect of hybrid configuration caching

The second experiment was conducted to study the result of configuration caching (configuration prefetching was not applied). Fig. 6 shows the results and we can see that for both compact and loose mode task sets, configuration caching had very limited effect when less than 3 RPUs were available on the chip. Both configuration caching algorithms took effect when RPU number reached 4 or 5.

In the next group of tests, we combined both prefetching and caching algorithms, and the experimental results are shown in Fig. 7. For compact mode task set, configuration prefetching and caching could reduce the average response time by 16.7% when 5 RPUs were available. For loose mode task set, configuration prefetching and caching could reduce the average response time by 32.4% when only 3 RPUs were available.



(a) Compact mode task set (b) Loose mode task set

Fig. 7. Effect of configuration caching and prefetching

Finally, we also had software implementations for all the application algorithms on the storage controller. Using the same task sets, our reconfigurable computing system (with 3-5 RPUs) got 16-37x performance gain over a 400MHz PPC440 hardcore based SoC on virtext-5 FXT device.

5 Conclusion

In this paper we presented an adaptable reconfigurable computing solution for SoC-based active storage controllers to efficiently process active service. The prototype hardware system was implemented on Xilinx virtex-5 LXT FPGA. Reconfigurable accelerators were coupled to the host to speed up computation-intensive applications. The downside of the system is large reconfiguration latency, which will downgrade the performance when applications are highly dynamic. To solve this problem, hybrid configuration caching and prefetching algorithms were proposed according to active service processing pattern.

Acknowledgments. This work is supported by the National Basic Research 973 Program of China under Grant No. 2004CB318201, 863 project 2008AA01A401, and Changjiang innovative group of Education of China No. IRT0725.

References

1. Smullen IV, C.W., Tarapore, S.R., Gurumurthi, S., et al.: Active storage revisited: the case for power and performance benefits for unstructured data processing applications. In: Proceedings of the 5th conference on Computing frontiers, Ischia, Italy, pp. 293–304 (2008)
2. Riedel, E., Gibson, G., Faloutsos, C.: Active Storage for Large-Scale Data Mining and Multimedia. In: Proceedings of the 24th International Conference on Very Large Data Bases, New York, US, pp. 62–73 (1998)
3. Huston, L., Sukthankar, R., Wickremesinghe, R.: Diamond: A Storage Architecture for Early Discard in Interactive Search. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, pp. 73–86 (2004)
4. Xiaonan, M., Reddy, A.L.N.: MVSS: An Active Storage Architecture. IEEE Transactions on Parallel and Distributed Systems 14(10), 993–1005 (2003)
5. Wolf, W.: High-Performance Embedded Computing: Architectures, Applications, and Methodologies. Morgan Kaufmann, San Francisco (2006)
6. Voros, N.S., Masselos, K.: System Level Design of Reconfigurable Systems-on-Chip. Springer, Heidelberg (2005)
7. AES Lounge, `http://www.iaik.tugraz.at/content/research/krypto/AES/`
8. Hauck, S.: Configurable Computing: The Theory and Practice of FPGA-based Computation. Morgan Kaufmann, San Francisco (2008)
9. Li, Z., Hauck, S.: Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In: Proceedings of the 2002 ACM/SIGDA 10th international symposium on Field-programmable gate arrays, Monterey, California, USA, pp. 187–195 (2002)
10. Chen, Y., Chen, S.Y.: Cost-Driven Hybrid Configuration Prefetching for Partial Reconfigurable Coprocessor. In: 14th Reconfigurable Architectures Workshop (RAW 2007), Long Beach, California, USA (March 2007)
11. Li, Z.: Configuration management techniques for reconfigurable computing. Ph.D. thesis (2002)
12. Sudhir, S., Nath, S.: Configuration Caching and Swapping. In: Brebner, G., Woods, R. (eds.) FPL 2001. LNCS, vol. 2147, pp. 192–202. Springer, Heidelberg (2001)

A Reconfigurable Disparity Engine for Stereovision in Advanced Driver Assistance Systems

Mehdi Darouich[1], Stephane Guyetant[1], and Dominique Lavenier[2]

[1] CEA LIST, Embedded Computing Laboratory,
PC94, Gif-sur-Yvette, 91191 France
mehdi.darouich@cea.fr
[2] ENS Cachan Bretagne / IRISA,
Campus de Beaulieu, Rennes, 35042 France

Abstract. Depth extraction in stereovision applications is very time-consuming and requires hardware acceleration in real-time context. A large number of methods have been proposed to handle this task. Each method answers more or less to real-time constraints, depending on the applicative context and user's needs. Thus, flexibility is a strong requirement for a generic hardware acceleration solution, particularly when ASIC implementation is targeted. This paper presents REEFS[1], a reconfigurable architecture for embedded real-time stereovision applications. This architecture is composed of three reconfigurable modules that enable flexibility at each step of depth extraction, from correlation window size to the matching method. It generates VGA depth maps with 64 disparity levels at almost 87 frames per second, answering hard real-time requirements, like in Advanced Driver Assistance Systems.

1 Introduction

Stereovision is widely used in many fields of computer vision, from robotics to intelligent transportation. Its principle is to calculate the depth of scene by triangulation from two images taken from different view points. The depth map is created using different methods and parameters, depending on the target application; in this article we focus on embedded stereovision applications. Some of these applications, such as advanced driver assistance systems (ADAS), have strong accuracy and real-time requirements. Other applications, like in mobile multimedia systems, have lower performance requirements but harder embedded constraints. Besides, a plethora of algorithmic solutions have been investigated to answer these issues.

Strong embedded and real-time requirements of stereovision applications imply the use of hardware accelerators. The algorithmic variability leads us to consider hardware flexibility as an important requirement as well. However, a trade-off must be found between efficiency and the level of flexibility. General

[1] Reconfigurable Embedded Engine for Flexible Stereovision.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 306–317, 2010.

Purpose Processors (GPP) are very flexible, through programming, but do not provide enough processing power to answer current requirements, as it consume too much power (40 to 100W). The use of Graphical Processing Units is more and more investigated to execute real-time vision applications as they provide high processing power and remain user-friendly through programmability. However, their power consumption (up to 100W) remains too high for many applications. FPGAs gives access to a high level of flexibility and high performances, but still with a low efficiency (regarding surface/performance/power ratio). Finally, application specific architectures provide high efficiency and allow real-time execution with a few Watts. However, their flexibility is limited to parameters customization (image size, disparity range, etc).

We propose a hardware architecture, targeting ASIC implementation. This architecture has a level of reconfiguration enabling area-based image processing techniques, and particularly advanced depth maps creation techniques, with a wide range of applicative conditions. Besides, our solution is designed to meet the requirements of applications in ADAS and can be scaled to adapt to any level of performance. The paper is organized as follows: section 2 presents the context of stereovision systems, outlining algorithmic flexibility needs and presenting related works. Section 3 presents the proposed architecture and its three reconfigurable modules. ASIC synthesis results and configuration examples are presented in section 4. Finally, conclusions and outlooks are given in section 5.

2 Stereovision Systems

Depth extraction is done by disparity map generation. Disparity is the gap between a point in the first image, called reference image, and the corresponding point in the second image. The disparity of a point is inversely proportional to its depth. Disparity map creation can be divided into two main steps: matching cost processing and corresponding pixels matching. The first step consists in evaluating the level of correspondence between pixels in the reference image and pixels from the other image: a matching cost is generated for each pixels pair. The second step uses these matching costs to extract the correct disparity. There are two main approaches for disparity map creation: feature-based approaches and area-based approaches. Feature-based approaches match features like edges, corners or points of interest and then require a reduced computational time. Area-based approaches process disparity for all the pixels of the reference image, generally taking into account their neighborhood. In this article, we focus on area-based approaches as they give access to high-density maps and are generally very regular and highly parallelizable. Furthermore, they are executed in deterministic time, which is required in high safety real-time applications.

The matching cost processing step is different from an application to another by the use of various metrics and window sizes and shapes [8]. The common metrics are SAD (Sum of Absolute Differences), SSD (Sum of Squared Differences), NCC (Normalized Cross-Correlation). SAD is often used in embedded solutions because it implies low processing cost; SSD and NCC require complex

operations, like square or multiply, and thus are hardly compatible with embedded constraints. Centered versions of these metrics (ZSAD, ZSSD, ZNCC) are less sensible to illumination variations between the two images, but require more processing time. CENSUS is another interesting metric as it is insensitive to illumination variations and has a low processing cost.

The pixel's neighborhood used in matching cost processing has an influence on the quality of generated disparity maps and the processing time. The window's shape is generally rectangular. Large windows are adapted to low-textured area but are less precise on edges than small ones. However, small windows produce noisy disparity maps. A way of improvement is to use multiple windows: in [4] for instance, the cost is calculated on a centered window and on four peripheral windows. The two peripheral windows with the best cost are added to the central cost. This way, a composite window is processed, that adapts its shape to the data.

The second step, the corresponding pixel matching, can be done by local methods, semi-global methods or global methods. The most common local method, called Winner Take All (WTA), consists in choosing the pixels pair with the best matching cost. This technique implies very low processing time but gives poor results on noisy stereo images and in ambiguous areas (occultation zones or repetitive textures). A possible enhancement is to detect and invalidate wrong matches: the error rate decreases as well as the disparity map density. The level of confidence of a match can be determined by analyzing matching costs distribution [7]. Another way of detecting wrong disparities is to use symmetry constraint [3] and compare the disparity map processed with the left image as reference and the disparity map processed with the right image as reference: disparities that do not correspond are thus invalidated. Semi-global matching methods give access to high density disparity maps with a lower error rate. In these methods, a constraint is applied to a part of the image to enhance disparity continuity and thus to reduce noise. Dynamic programming [6] is a well known semi-global method: disparity on image lines is optimized using order and unicity constraints. However, it requires much more memory than local methods ($\approx$ 100kB for VGA images). Global matching methods are too costly to be used in embedded solutions because their memorization needs are too high ($\approx$ 100MB for VGA images) and the number of operations can be more than 10^8 operations per pixel. Detection quality can also be increased by using preprocessing or post-processing. Rank filter applied on both images before matching cost processing eliminates illumination variation sensibility [8]. In [7], a median filter is applied on the disparity map to eliminate wrong matches.

Image size is an important parameter regarding the quality of the result and the needed processing time. Smaller QVGA stereo pairs (320x240) are often used to shorten the processing time. However, higher image resolutions give access to more detailed disparity maps, especially for far objects, implying more processing time. The maximum disparity range is also an important parameter: it depends on the stereo cameras pair geometry and defines the minimum detection distance. Besides, as close objects detection is generally critical, maximum disparity range

highly depends on detection requirements. For a VGA stereo pair, maximum range usually varies from 64 to 128, or even 256 pixels.

A lot of application specific architectures aim to answer stereovision problem, but without flexibility considerations. For example, the Deepsea processor [9] uses CENSUS metric on 7x7 windows for 52 disparity range and is able to generate 512x480 disparity maps at 200 fps, which corresponds to a performance of 2.6 GPDS[2]. SIMD architectures give access to more flexibility but with lower performance. For example, the IMAP VISION of NEC [6] processes dynamic programming based disparity maps on 128 per 128 regions of interest with 64 dispariy levels at 10 fps (≈11 MPDS). In [2], the proposed architecture parameters (image size, disparity range and correlation window size among others) can be tuned by static FPGA reconfiguration. It reaches a maximum peformance of 8 GPDS. An adaptive architecture targeting both ASIC and FPGA implementations is proposed in [1]. It is flexible on disparity range and windows size, and achieves 2 GPDS. However, the flexibility of the correlation window is based on processing block merging, which limits the possibilities. The VoC [5] is based on a reconfigurable matrix and processes matching costs on windows with various sizes, but only with SAD metric. Its performance is about 1.4 GPDS.

As shown in this section, stereo matching methods taxonomy is wide and the quality of a given method strongly depends on targeted applications, even in the same applicative field. Thus, flexibility is a strong requirement for a generic hardware acceleration solution targeting ASIC implementation.

3 Presentation of the REEFS Architecture

3.1 Global Overview

The proposed architecture is flexible enough to execute a broad range of algorithms. Besides, targeting the ADAS domain, associated requirements are to be fulfiled. First of all, the image frequency for this kind of applications is typically 40 fps. Secondly, as we need high quality detection, we have based our study on VGA images (640x480), which implies a maximum disparity from 64 to 128 pixels. However, other image sizes can be used as well, depending on user's requirements. The maximum window size for matching cost calculation is established to 15x15, which is large enough for most known applications. As shown in figure 1, the REEFS (Reconfigurable embedded Engine for Flexible Stereovision) is a data-flow oriented architecture composed of three reconfigurable modules. Based on the assumption that input images are rectified, the architecture extracts disparity over lines. Thus, for a given line, needed pixels are located on 15 lines in both images which are stored in line buffers. This on-the-fly approach allows memory use reduction.

[2] PDS: Point Disparity per Second, *i.e.* the number of matching costs generated per second. Note that this metric does not take into consideration complexity of the cost generation method and the matching method.

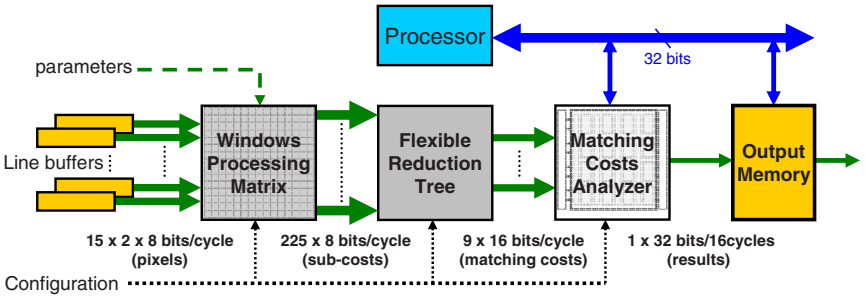


Fig. 1. The REEFS allows the generation and the analysis of 9 matching costs per cycle. It is based on three reconfigurable modules, namely the Windows Processing Matrix (WPM), the Flexible Reduction Tree (FRT) and the Matching Cost Analyzer (MCA).

The first reconfigurable module, called Windows Processing Matrix (WPM), handles the first step of matching costs generation: sub-costs generation. It processes, every cycle, 15x15 windows on both images (225 pixels pairs) and generates as many sub-costs as pixel pairs. The WPM's reconfiguration allows to specify which metric is used for matching costs generation. To reduce the memory bandwith implied by window updates, currently processed pixels are stored inside the WPM. Exploiting data redundancy between two consecutive windows, the neighborhood update is done in one cycle by loading 15 pixels column from buffer lines. The second reconfigurable module, the Flexible Reduction Tree (FRT), sums the 225 sub-costs to obtain matching costs. Regarding current requirements, a minimum of 4 matching costs must be generated every cycle[3]. However, as multiple windows support is targeted, a maximum of 9 matching costs are generated each cycle. The shape and size of the window on which each matching cost is processed are application-dependant and can be modified by reconfiguring the FRT. Finally, the Matching Costs Analyzer (MCA) processes 9 matching costs each cycle to extract related disparity and metadata used by external high-level decision modules. These output data are stored in a buffer memory, named output memory in the figure 1, that is read by external modules. The MCA reconfiguration gives access to various extracting methods. A processor has been added to handle irregular processing; it has access to MCA input and output data and to the output memory. We now describe in detail the three reconfigurable modules: WPM, FRT and MCA.

3.2 Windows Processing Matrix

Most of the matching cost functions can be written as follows:

$$C = \sum_{N_R, N_A} f(P_R, P_A, c_R, c_A) \tag{1}$$

[3] With a given system frequency of 200MHz, generating VGA disparity maps at 40fps implies the processing of $\frac{640 \times 480 \times 40 \times 64}{200e6} \approx 4$ matching costs per cycle.

In this formula, the matching cost C is the sum of all sub-costs $f(P_R, P_A, c_R, c_A)$. Each sub-cost is calculated using a pixels pair P_R and P_A, belonging respectively to reference area N_R and to area N_A, and constants c_R and c_A related respectively to area N_R and N_A. The function f corresponds to the metric used. As we target a system able to generate one matching cost per cycle, we need to process as many sub-costs as there are in the concerned areas, using the desired metric. Besides, among common metrics, SAD, ZSAD and CENSUS are chosen for their simplicity and the fact that each provides result with characteristics different from the others.

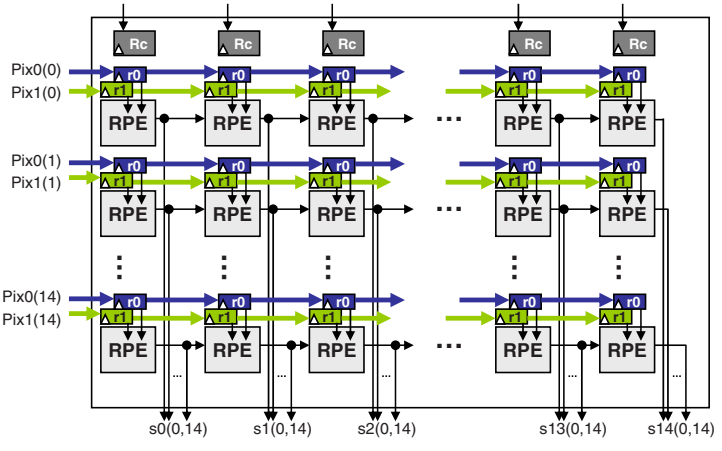


Fig. 2. The Windows Processing Matrix, composed of 15 columns of 15 RPEs each, processes 225 sub-costs necessary for matching costs generation on 15x15 areas each cycle. Reconfiguration of RPEs functionality and interconnection enables flexibility on the implemented metric.

As shown in figure 2, the WPM is composed of 15 columns of 15 Reconfigurable Processing Elements (RPE) each. These 225 RPEs allow generating one matching cost over an area of 15x15 pixels every cycle. Various operations can be executed by the RPE, depending on the chosen configuration: difference (D), absolute difference (AD), comparison (CMP), CENSUS and output weighting. The reconfiguration ability provides flexibility on the sub-cost generation function (f in equation 1) used and thus on the metric implemented.

The inputs connexion of a RPE can also be reconfigured: each input can be connected to one of the two assigned pixel registers (r0 and r1 in figure 2), which store a pixel of the considered neighborhood (P_R and P_A in equation 1). It can also be connected to the column register Rc, which stores data that are common to the whole column. This register is useful when constant parameters (c_R and c_A in equation 1) are needed as inputs, for instance in *Zero-mean* metrics, *CENSUS* or even in *RANK* filter. At last, each input can be connected to the output of the previous RPE. Thus, complex operations can be executed at each cycle. For

example in ZSAD, one RPE handles zero-mean rectification on pixels while the other one performs absolute differences.

3.3 Flexible Reduction Tree

The Flexible Reduction Tree's first purpose is to sum, each cycle, the 225 sub-costs generated by the WPM to obtain up to 9 matching costs. It must also answer flexibility requirements on the shape and the size of windows to ensure a good level of algorithmic variability. As displayed in figure 3, the FRT is divided in 3 levels. The first level is composed of reconfigurable sub-trees each assigned to a WPM column. The second level is also composed of the same type of sub-tree. The configuration of the first level allows to specify vertical patterns while the configuration of the second level impacts horizontal patterns. The last level is a simplified sub-tree (see figure 4(b)) which enables the combination of several matching costs.

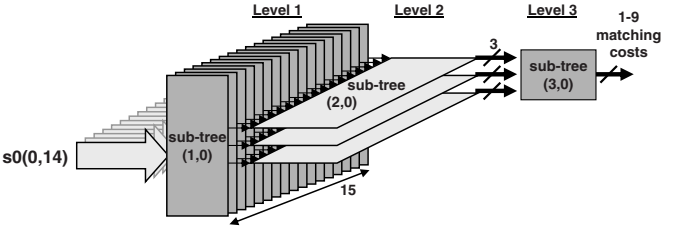
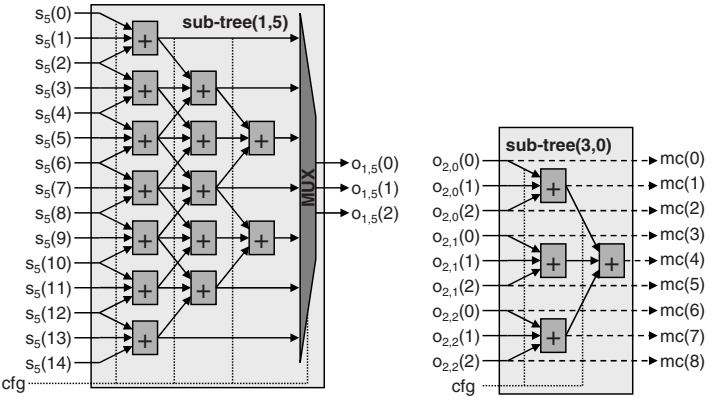


Fig. 3. The Flexible Reduction Tree sums the 225 sub-costs generated each cycle by the WPM. It is divided in 3 levels.



(a) Sub-tree used in level 1 and 2 (b) Sub-tree of level 3

Fig. 4. The first two levels of the FRT are composed of sub-trees shown in 4(a). The third level is a simpler sub-tree shown in 4(b). Active ways are selected by configuration.

The first type of reconfigurable sub-tree is shown in figure 4(a). It consists in a 3-ways adders tree with reconfigurable connectivity: each adder input can be disabled, in order to define the desired pattern, which is specified by a configuration register, *cfg* in figure 4(a). The overlapping of 3-ways adders inputs gives access to higher flexibility in multiple windows generation. Thus, the maximum vertical and horizontal overlapping is 5 pixels between two windows and 3 pixels between three windows, which increases combination possibilities. Nevertheless, the maximum window size depends on the number of matching costs to be processed simultaneously. For example, for a single matching cost, window size can vary from 2x2 to 15x15 pixels. In multiple windows mode, the maximum size is reduced to 11x11.

3.4 Matching Costs Analyzer

The purpose of matching costs analysis is to extract correct disparity and metadata from generated matching costs. This step uses simple operations like comparison or addition, and more elaborated operations like sorting or counting. However, the input data bandwidth is still high (9 matching costs per cycle) which implies the use of parallel processing units.

The Matching Costs Analyzer goal is to apply these operations on generated matching costs. We based the MCA architecture on sorting networks structure as it suits well to data analysis and exchange. Instead of simple compare-exchange operators, our network is composed of reconfigurable elements named ASCE (Add-Sub-Compare-Exchange). These processing elements have two inputs and two outputs and provide a set of operations summarized in table 1. Interconnection between ASCEs from a column to another is fully flexible by reconfiguration. It is also possible to plug inputs and outputs of the MCA together to create longer data paths. Figure 5 shows a MCA composed of 4 columns of 9 ASCEs each. In this figure, *cfg* specifies the configuration for interconnections and ASCEs functions. As the processing power is limited by the number

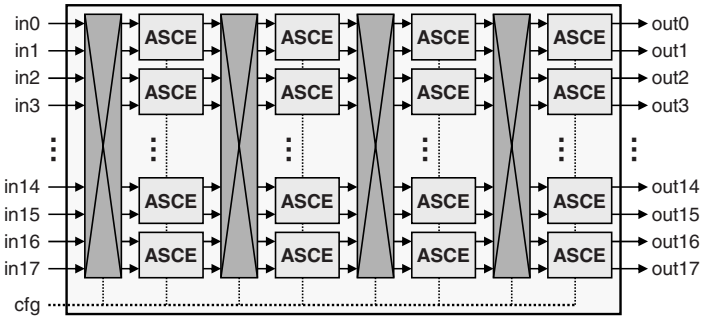


Fig. 5. The Matching Cost Analyzer is composed of reconfigurable elements (ASCE) and handles matching methods. Each ASCE's input can be connected to the previous column ASCE's output through a reconfigurable interconnection (crossed box).

Table 1. Operations supported by an ASCE. *a* and *b* are the inputs. *c* and *d* are the outputs. *Loop* indicates if the operation can be looped, *i.e. b* receives the data present the previous cycle in output *c*. In counter mode, *count* is the internal counter value.

Name	Loop	Outputs	Name	Loop	Outputs
ID	yes	c=a, d=b	**Mm**	yes	c=max(a,b), d=min(a,b)
ADD	yes	c=(a+b), d=0	**mM**	yes	c=min(a,b), d=max(a,b)
CRb	yes	c=count if a=1, d=b	**MC**	yes	c=max(a,b), d=cmp(a,b)
CRbc	yes	c=count if a=1 else c=b, d=0	**mC**	yes	c=min(a,b), d=cmp(a,b)
Necb	no	c=a if b=0 else c=-a, d=a	**Subs**	no	c=(a-b), d=sign(a-b)

of ASCEs, the allowed analysis complexity depends on the number of simultaneous reference pixels processed by the matrix, *i.e.* the number of independent processes executed.

3.5 Reconfiguration

The reconfiguration is done through the configuration port: each configuration register has its own address and thus, each module can be reconfigured independently. The bitstream total size is about 1800 bits (225 bytes). The architecture supports two reconfiguration modes. The first one is the static reconfiguration mode: each module is reconfigured once at the initialization of the system and no reconfiguration is needed while the application is running. Dynamic reconfiguration mode is also supported, as the whole architecture reconfiguration is done in less than 100 cycles. However, the frequency of reconfigurations is limited by performance requirements as bitstream loading implies penalties. The typical reconfiguration granularity is the image line: several processings can be alternately executed on each line, allowing hardware reuse. For instance, at first, the architecture is configured for Rank filtering. All the pixels of a given line are filtered and stored in a line buffer, which takes about 1300 cycles (640 cycles for each image). Then the architecture is reconfigured to perform disparity extraction on filtered pixels. The extraction step takes approximately 41000 cycles (640×64 cycles). The reconfiguration overhead in this case study is less then 0.5% of the whole processing time.

4 Implementation Results and Case Studies

4.1 Real-Time Performances and ASIC Synthesis

An ASIC synthesis of the proposed architecture has been carried out, with ST 65nm Low Power technology, at 200MHz. This synthesis permits us to obtain a good estimation of the surface for each reconfigurable module : 0.7 mm^2 for

the WPM, 0.3 mm^2 for the FRT and 0.5 mm^2 for the MCA. The total surface is around 1.5 mm^2[4], which satisfies the embedded constraints of ADAS. At 200MHz frequency, and considering a VGA stereo pair with a disparity range of 64 pixels, the generation of 1 to 9 matching costs per cycle gives access to a frame rate from 10 fps to 87 fps. It corresponds to a maximum performance of 1.7 GPDS, which is quite high regarding the flexibility provided. Besides, the REEFS architecture is fully compliant concerning real-time requirements for Advanced Driver Assistance Systems which are usualy set to 40 fps.

4.2 Configuration Cases

In the following, two configuration examples are presented to show the flexibility and the real-time abilities of the architecture. The first example enables a high frame rate with basic matching method. In the second example, both matching costs generation and matching method are more complex, which implies a lower frame rate. For each case, results on performance (frames per second) and utilization level are presented. The utilization level represents the number of reconfigurable processing elements used in each module. We consider that the left image is the reference image.

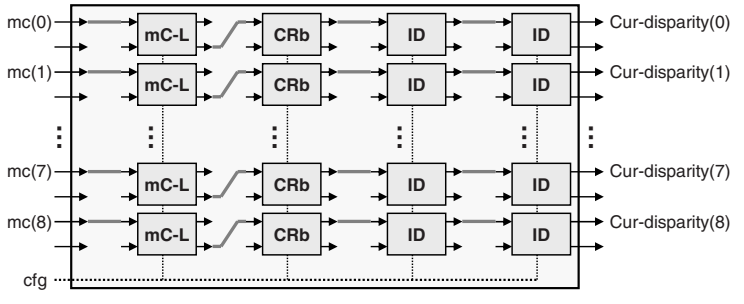


Fig. 6. Example of a MCA module's configuration that allows the processing of WTA method on 9 independent matching costs. The reached frame rate is 87 fps. The suffix *-L* indicates a looped operation.

In the first example, WPM and FRT modules are configured to generate 9 independent CENSUS matching costs on 5x5 windows, without overlapping. The MCA module applies a Winner-Take-All method simultaneously on each cost, as shown in figure 6. Thus, for each MCA line, the first ASCE compares the cost *mc(k)* in input with the current minimum cost (stored inside the element). If a new minimum value is detected, the second ASCE, which is configured as a counter, updates its output with its internal counter value. The result *cur-disparity(k)* represents the gap that gives the best matching cost for reference

[4] The surface occupied by registers is about 28% of the total surface.

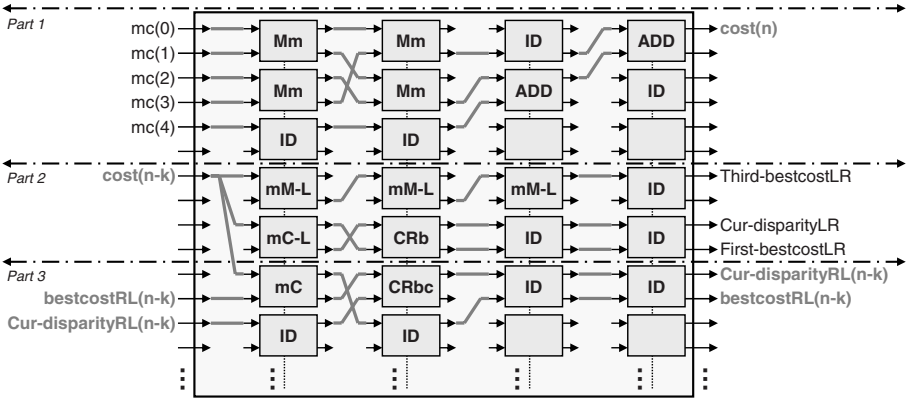


Fig. 7. Example of multiple windows costs generation followed by the extraction of disparity and metadata for symmetry constraint checking and confidence level calculation. Inputs and outputs in bold are plugged together.

pixel k. In this configuration, the theorical frame rate is 87 fps. Concerning the utilization level, both WPM and MCA are used at 100%, and the FRT is used at 72%.

The second configuration example enables the processing of one SAD matching cost on multiple windows, as proposed in [4]. The FRT module generates one central 7x7 window and four 7x7 peripheral windows, with overlapping. Figure 7 shows the MCA's configuration. In a first step, the 2 minimum peripheral costs are extracted and summed with the central cost to generate a combined cost (part 1). Then, this combined cost is used in a second step for the extraction of the left reference disparity (cur-disparityLR) with WTA method and the first and third best related matching costs (part 2). The combined cost is also used for extracting the right reference disparity (cur-disparityRL) with WTA method (part 3). After 64 shifts, all metadata are processed by the processor to filter wrong disparities with symmetry constraint checking and confidence level calculation. As one matching cost is processed each cycle, the frame rate is 10 fps. Only 54% of the WPM's RPEs are used while the FRT is still used at 72%. The MCA module is used at 67%.

5 Conclusion

Disparity maps generation is a key stage in real-time embedded stereovision applications. A high number of algorithmic solutions exists and the choice depends on the targeted application and applicative constraints. Thus, we propose a reconfigurable architecture that enables a high algorithmic flexibility. The REEFS architecture provides flexibility on the metrics, the size and the shape of correlation windows, and on matching methods. Besides it answers real-time requirements as it can produce 640x480 disparity maps at a maximum frame rate of

87 fps. The three main reconfigurable modules have been synthesized and the total surface is about 1.5 mm^2. The characteristics of our architecture satisfy the requirements of Advanced Driver Assistance Systems.

In the future, this architecture will be integrated in a System on Chip for ADAS. This SoC will integrate high-level processing units for disparity map analysis: road-plan extraction or obstacle localisation for instance. In this context, it will be possible to scale and to parallelize the current architecture to increase the maximum frame rate when using large windows or performing complex analysis. The execution of other types of area-based image processing will also be considered in this SoC.

References

1. Ambrosch, K., Kubinger, W., Humenberger, M., Steininger, A.: Flexible hardware-based stereo matching. EURASIP J. Embedded Syst. 2008, 1–12 (2008)
2. Cuadrado, C., Zuloaga, A., Martin, J., Lazaro, J., Jimenez, J.: Real-time stereo vision processing system in a fpga. In: Proc. IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics, pp. 3455–3460 (2006)
3. Fua, P.: A Parallel Stereo Algorithm that Produces Dense Depth Maps and Preserves Image Features. Machine Vision and Applications 6, 35–49 (1993)
4. Hirschmüller, H., Innocent, P.R., Garibaldi, J.: Real-time correlation-based stereo vision with reduced border errors. Int. J. Comput. Vision 47, 229–246 (2002)
5. Jacobi, R., Cardoso, R., Borges, G.: Voc: a reconfigurable matrix for stereo vision processing. In: Proc. 20th Int. Parallel and Distributed Processing Symposium (2006)
6. Kraft, G., Jonker, P.: Real-time stereo with dense output by a simd-computed dynamic programming algorithm. In: PDPTA 2002: Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (2002)
7. Muhlmann, K., Maier, D., Hesser, R., Manner, R.: Calculating dense disparity maps from color stereo images, an efficient implementation. In: Proc. IEEE Workshop on Stereo and Multi-Baseline Vision (2001)
8. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. Int. J. of Computer Vision 47, 7–42 (2002)
9. Woodfill, J., Gordon, G., Buck, R.: Tyzx deepsea high speed stereo vision system. In: Proc. Conference on Computer Vision and Pattern Recognition Workshop CVPRW 2004, p. 41 (2004)

A Modified Merging Approach for Datapath Configuration Time Reduction

Mahmood Fazlali[1,2], Ali Zakerolhosseini[1], and Georgi Gaydadjiev[2]

[1] Department of Computer Engineering, Shahid Beheshti University G.C, Tehran, Iran
[2] Computer Engineering Lab., Delft University of Technology, Delft, The Netherlands
fazlali@cc.sbu.ac.ir, a-zaker@sbu.ac.ir,
g.n.gaydadjiev@tudelft.nl

Abstract. This paper represents a modified datapath merging technique to amortize the configuration latency of mapping datapaths on reconfigurable fabric in Run-Time Reconfigurable Systems (RTR). This method embeds together the different Data Flow Graphs (DFGs), corresponding to the loop kernels to create a single datapath (merged datapath) instead of multiple datapaths. The DFGs are merged in steps where each step corresponds to combining a DFG onto the merged datapath. Afterwards, the method combines the resources inside the merged datapath to minimize the configuration time by employing the maximum weighted clique technique. The proposed merging technique is evaluated using the Media-bench suit workloads. The results indicate that our technique outperforms previous HLS approaches aimed at RTR systems and reduces the datapath configuration time up to 10%.

Keywords: Reconfigurable Computers, Run-Time Reconfiguration, CAD Algorithms for FPGA and Reconfigurable Systems.

1 Introduction

Multimedia applications contain computationally intensive kernels that demand hardware implementation in order to exhibit real time performance. The kernels can be accelerated by employing reconfigurable units which provide flexibility and reusable hardware resources. Often FPGA resources are limited and all the kernels cannot be mapped inside the FPGA. Hence, the kernels are to be configured at run-time in order to create a virtual hardware and accelerate more sections of the applications [1,2]. This is called Run-Time Reconfiguration (RTR).

The RTR in FPGA involves dynamic reconfiguration that with the current technology is a time-consuming process and hence affects the available system performance. For example, the MOLEN reconfigurable processor which is depicted in Fig.1 utilizes custom configured hardware to execute computational intensive functions [3]. The MOLEN architecture consists of two parts; General-Purpose Processor (*GPP*) and Reconfigurable Processor (*RP*), which is usually implemented on an FPGA. The RP is used for hardware acceleration. The execution phase by the RP is divided into two distinct steps: *set* and *execute*. In the *set* phase, RP is configured to perform the

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 318–328, 2010.

required datapath and, in the *execute* phase the actual function is executed. However, the run-time reconfiguration in the *set* phase imposes considerable overhead to the performance of the system. Therefore, the configuration should be done as soon and efficient as possible.

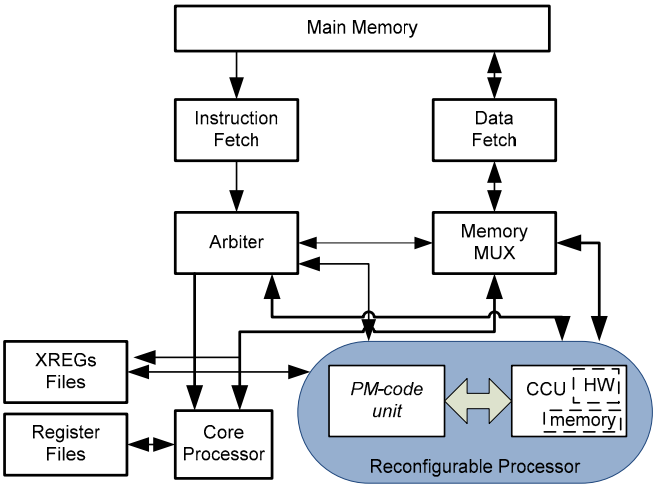


Fig. 1. MOLEN Hardware Organization

There are many proposals for the configuration time reduction in RTR system. Some researchers have attempted to reduce configuration overhead by reusing the same hardware for different applications or rearranging the execution order of tasks in a sequence that requires lower number of reconfigurations [4]. Thus, however, it can not be applied to many realistic applications. Cashing the configuration bit-stream is another solution that was represented in [2] for the configuration overhead reduction however, spatial and temporal locality of references is not yet proven or accepted as a general principle in replacement policy for RTR systems, and needs more investigation. The configuration overhead can be efficiently hidden by employing approaches such as configuration perfecting where a static schedule is present [2]. Such totally predictable application schedules form only a small subset of the total class of the applications suitable for reconfigurable hardware acceleration.

The average configuration time can be computed using the average configuration bit-stream and the speed of the configuration interface [5]. So, it is apparent that the reduction in the bit-stream length is conducting to the reduction in the configuration time. Some researchers focus on reducing the configuration time of a FPGA by compressing the bit-stream. However, such techniques are costly and affect the performance of the system [6]. Therefore, it is better to amortize the configuration time using High Level Synthesis (HLS) [7].

In HLS, Data Flow Graphs (DFG) are created for the computational intensive kernel loops. Afterwards, the resources in the DFG are shared to create a datapath. In this way, the hardware cost is reduced. The synthesis process comprises the major tasks of

scheduling, resource allocation, resource binding, and interconnection binding [8,9,10].

Making a multimode datapath instead of multiple datapaths can effectively reduce the hardware cost [11]. Datapath merging is a technique introduced to create a reconfigurable datapath for two or more DFGs [12,13]. It enables the reuse of hardware resources by identifying similarities among several DFGs. This technique was employed in [14] for the reduction of configuration time. This technique, however, cannot minimize the configuration time in *RTR* systems.

The main contribution of this paper is presenting a modified datapath merging technique for the configuration time reduction in an RTR system. In the proposed technique we combined the resources inside the merged datapath to reconstruct it to a new merged datapath which requires fewer functional units and multiplexers in order to minimize the configuration bit-stream.

The next section explains the configuration time reduction in datapath merging. Section 3 presents the suggested datapath merging technique. The experimental results are included in Section 4, and Section 5 concludes the paper.

2 Configuration Time Reduction in Datapath Merging

The problem can be considered as merging only those DFGs that correspond to computational intensive kernels while making a merged datapath with a reduced configuration time.

Let DFG $G=(V,E)$, where $V=\{v_i \mid i=1...n\}$, is a set of vertices and $E=\{e_j \mid j=1...m\}$, is a set of edges. A vertex $v \in V$ represents an operation executable by a functional unit that has a set of input ports p. An edge $e=(u,v,p) \in E$, indicates a data transfer from the vertex u to the input port p of vertex v.

Vertex-merging is the creation of a vertex v' replacing vertices $v_i\ v_j ...\ v_k$ and edge-merging is the creation of an edge $e'=(u',v',p')$ replacing edges $e_i = (u_i,v_i,p_i)$, $e_j = (u_j,v_j,p_j)$.. $e_k = (u_k,v_k,p_k)$.

In order to merge edges, their vertices are to be merged while their corresponding input ports are matched together. Here, a functional unit is used for each merged vertex v' capable of performing the functions of vertices mapped onto v'.

A merged datapath $MD=(V',E')$, corresponding to DFGs $G_i=(V_i,E_i)$ $i=1...n$, is a directed graph. A vertex $v' \in V'$ represents a merge of vertices $v^j \in V_j$ and an edge $e'=(u',v',p') \in E'$ represents a merge of edges $e^j \in E_j$ where $j \in J \subseteq \{i,...,n\}$.

The configuration time T_C of the merged datapath $MDP=(V',E')$ is:

$$T_C = T_F + T_I$$

Where $T_F = \sum_{\forall v' \in V'} T_f(v')$ is the functional units configuration time and $T_I = \sum_{\forall v' \in V'} T_i(MUX)$ is the multiplexer's configuration time in the merged datapath [14]. $T_f(v')$ is the configuration time of a functional unit (or storage unit) allocated to v', and T_i (MUX) represents the configuration time of multiplexers employed at the input port of a vertex. There are different ways to merge vertices and edges from DFGs and this, in effect, can produce several merged datapaths for the input DFGs.

An optimized merging method is required in order to manage resource allocation, resource binding, interconnection binding and also minimizing the datapath configuration time of *DPM*. It means the realization of a *MDP* for the DFGs is desirable if T_C is minimal.

Fig.2 illustrates an example of datapath merging where DFGs G_1 and G_2 from this figure are merged and the merged datapath *MDP* is created. Considering these DFGs, if operation of a vertex from G_1 and operation of a vertex from G_2 can be computed with the same type of functional unit, they will become potential for merging. For example, $a_1 \in G_1$ and $b_1 \in G_2$ can be executed by the same functional unit. Thus, these vertices are merged together and the vertex (a_1/ b_1) is created for them in *MDP*. If a vertex cannot be merged onto other vertices, it will remain in the merged datapath without any modification. After merging two vertices, multiplexers are employed in the merged datapath to select the currect input operand. This is illustrated in the input ports of vertex (a_5/ b_3) in Fig.2

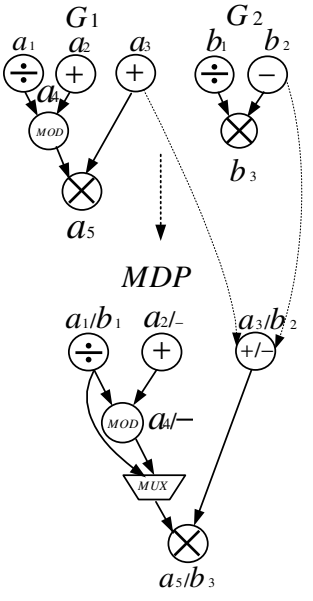


Fig. 2. The merged datapath *MDP* for DFGs G_1 and G_2 [14]

An edge from G_1 cannot be merged onto an edge from G_2 unless the vertices of the edges are merged. As it can be seen in Fig.1, because of merging both of the vertices a_3 and $a_5 \in G_1$ onto the other vertices b_2 and $b_3 \in G_2$, the edges (a_3, a_5) and (b_2, b_3) are merged together and the edge $(a_3/b_2, a_5/b_3)$ is created instead in *MDP*. In this case, it is not necessary to use a multiplexer in the input ports of the vertex (a_5 / b_3) to select the input operands.

3 The Proposed Datapath Merging Technique

Although merging DFGs using Integer Linear Programming (*ILP*) can minimize the merged datapath configuration time, it is an *NP*-complete problem where the number of DFGs increases or number of nodes in DFGs increases [12]. Therefore, in the proposed datapath merging technique, the DFGs are merged together in steps. Afterward, the resources inside the merged datapath are combined to optimize the configuration time. In this way, the desired merged datapath for the input DFGs is created.

Combining the resources inside the merged datapath paves the way for having the merged datapath with smallest configuration time. Thus, we can define the merging problem to create the desired merged datapath as follows:

Given a merged datapath MDP, the desired merged datapath, MDP', is realized such that the merged datapath configuration time T_C is minimal.

Fig.3 shows an example of datapath merging proposed in this paper to merge DFGs. In Fig. 3(a), five simple DFGs, G_1,..., G_5 are illustrated. Each hardware unit and interconnection unit (multiplexer) has its own configuration time. These DFGs are merged in steps employing the method in [14] while the datapath configuration time, T_C, is reduced and *MDP* is created. Fig. 3(b) shows the results of sharing the resources inside *MDP* to create *MDP'* for the DFGs. It combines the resources inside the *MDP,* together to minimize the configuration time. In this figure, the vertices c_1 and e_1 from *MDP* have been merged to create a vertex c_1/e_1 in the *MDP*'. On the other hand, vertices $a_2/b_2/c_2$ and d_2/e_2 are merged together to create vertex $a_2/b_2/c_2/d_2/e_2$ in *MDP*'. Therefore, two edges ($a_2/b_2/c_2$, c_1) and (d_2/e_2 , e_1) in *MDP* can be merged together to create an edge ($a_2/b_2/c_2/d_2/e_2$, c_1/e_1) in *MDP*'.

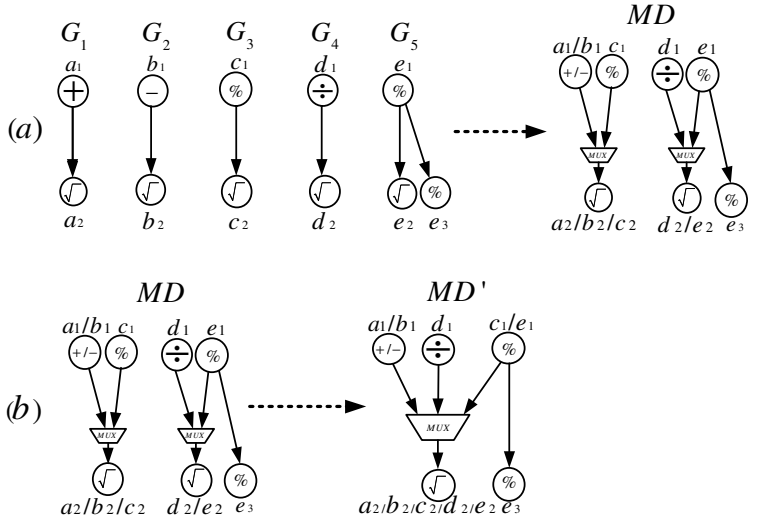


Fig. 3. (a) merging five DFGs G_1,…, G_5 in steps to create merged datapath *MDP*, (b) sharing the resources inside *MDP* to create *MDP'*

We merge the resources inside *MDP* in three stages. In the first stage all merging possibility among the resources inside *MDP* are to be considered as a compatibility graph. To do this, we employed the compatibility graph in [13] but use different approach to reduce the configuration time. Then in the second stage, the maximum weighted clique in this graph " M_c " is found. By searching M_c in the compatibility graph and reconstructing the *MDP* using this clique, *MDP*' is created.

To combine the resources inside the merged datapath, the merging possibilities among the vertices and the merging possibilities among the edges in the merged datapath *MDP* should be taken into consideration. The compatibility graph shows the merging possibility among the vertices inside *MDP* to create the same type of vertex in *MDP*' or, the merging possibility among the edges inside *MDP* to create the same type of edge in *MDP*'. Below, the compatibility graph G_c for the input merged datapath *MDP* is defined formally.

A compatibility graph corresponding to input merged datapath, *MDP* is an undirected weighted graph $G_c=(N_c,A_c)$ where:

- Each Node $n_c \in N_c$ with weight w_c corresponds to:
  - — Vertex-merging that is a possible merging of the vertices $v_i,v_j \ldots v_k \in MDP$ to create a vertex $v_{i..k} \in MDP'$ where it does not merge the same vertex from a DFG G_x onto different vertices from a DFG G_y or vice-versa.
  - — Edge-merging that is a possible merging of edges $e_i=(u_i,v_i,p_i)$, $e_j=(u_j,v_j,p_j) \ldots e_k=(u_k,v_k,p_k) \in MDP$ to create an edge $e_{i..k} \in MDP'$ where it does not merge the same vertex from a DFG G_x onto a different vertices from a DFG G_y or vice-versa.
- Each arc $a_c=(n_c,m_c) \in G_c$ illustrates that its nodes n_c and m_c (merging nodes) are compatible. It means they do not merge the same vertex from a DFG G_x onto different vertices from a DFG G_y or vice-versa.
- Each node's weight, w_c, represents the reduction in configuration time resulting from merging the vertices or merging the edges.

By merging vertices $v_i,v_j..v_k \in MDP$ together, a vertex v' is created in *MDP*'. Besides, for each input port of v' which has more than one incoming edge, *MDP*' will have a multiplexer to select the input operand. That is the reason to add multiplexer to the input ports of v', or add an input to previous multiplexers (if v_i has a multiplexer).

In this case, configuration time reduction of the $n_c \in G_c$ is equal to the difference between the configuration time of the hardware units and multiplexers before merging vertices and, their configuration time after merging, that is:

$$w_i = (T_f(v_i) + T_f(v_j)) - (T_f(v') + m \times T(mux_i)) \quad (1)$$

$T_f(v_i)$ and $T_f(v_j)$ in equation (1) are the hardware units configuration time before the merging and $(T_f(v'))$ is the hardware units configuration time after the merging. Furthermore, $m \times T(mux_i)$ is used to show the increase due to the multiplexer configuration time. If the multiplexer has the same number of inputs as it had before, then $m=0$. Otherwise, m shows the increase in the size of the multiplexer (for example going from 4 ports multiplexer to 8 port multiplexer, m=1).

By merging edges $e_i,e_j..e_k \in MDP$, for each input port of v', one incoming edge is created in *MDP*'. Hence, there will be no multiplexer creation or there will be no change in the number of multiplexer inputs. Configuration time reduction achieved by

this type of merging corresponds to equation (2) that is the weight of removing the multiplexers, or decreasing the size of the multiplexers.

$$w_i = m \times T(mux_i) \tag{2}$$

The nodes in the compatibility graph have signed integer weights. Because, although some vertex-merging have negative weights (i.e., increase in the configuration time), their resultant weight with edge-merging has positive weight (i.e., reduction in configuration time). Note that the edges from *MDP* are not merged unless their vertices are merged. In this way, negative vertices should be included in the compatibility graph.

Up to here, all the merging possibilities inside the *MDP*' have been presented as the compatibility graph G_c. Choosing compatible nodes with more weight in G_c in order to create *MDP*', is equal to finding a completely connected sub graph with more weight from G_c. This graph is called clique in the graph algorithms. A **clique** C_c is a subset of nodes in a compatibility graph, $C_c \subset G_c$, such that for all the distinct nodes $u,v \in C_c$, they are connected together ($u,v \in E_c$). A clique is maximum if there are no larger cliques available in G_c. The **maximum weighted clique** M_c, is a clique in G_c that the total weight of its nodes is more than any other C_c in G_c. By employing the maximum weighted clique, *MDP*' can be created.

Calculation of the maximum weighted clique from the compatibility graph is known to be an *NP*-complete problem [15]. In order to solve the problem, Branch&Bound algorithms can be employed. They provide an appropriate problem search space in order to solve the maximum weighted clique problem. The algorithm presented in [16] chooses an efficient method to select nodes and predicts bounds for quick backtracking. Our proposed technique uses the same optimizations as indicated in [16], but the assumption to have positive weights for the nodes in the input graph have been changed to the signed integer weights for the nodes.

After finding M_c in the compatibility graph, the merging possibilities represented by the nodes in M_c is used to reconstruct *MDP* and create *MDP*'. Each node in M_c gives a merging possibility among the edges (or among the vertices) inside *MDP*. This way, edges and vertices inside the *MDP* are merged together to minimize the configuration time where, these merges do not merge the same vertex from a DFG G_x onto a different vertices from a DFG G_y.

4 Experimental Results and Analysis

To evaluate the effectiveness of the proposed technique, we made comparison with techniques that were proposed in [14] and [7]. In [14] a datapath merging technique, based on inter-DFGs resource sharing, has been proposed. It applies the HLS algorithms such as functional unit allocation, register allocation, and interconnection binding, simultaneously to the DFGs in steps. In [7] conventional HLS technique for RTR system, based on intra-DFG resource sharing, has been proposed. It applies HLS algorithms to each DFG to create its datapath. These techniques and the proposed technique in this paper were applied to five benchmarks from the Media-bench suite [17]. There are computational intensive kernels (inner loop kernels) in each benchmark that their entity makes them suitable for mapping into a reconfigurable unit in the RTR system. Each benchmark was compiled using the GCC compiler and for each loop, a DFG was generated from the loop RTL code.

For each benchmark, up to 3 kernels were considered and their corresponding DFGs were iteratively merged from the larger DFG into the smaller one. The configuration time of bit-stream in a FPGA is estimated as: (size of bit-stream) / (configuration clock frequency). After obtaining the bit-stream of a functional unit and a multiplexer by ISE 10.2, their configuration times were calculated based on their configuration bit-stream. Xilinx FPGAs support partial reconfiguration therefore; they are suitable for RTR systems. In our experiment, FPGA Virtex5-xc5vlx was employed as a targeted platform. In order to present the efficiency of our technique, the maximum configuration clock frequency of the FPGA (100 Mbps) was considered which is the worst case scenario for our method.

Table 1. Kernels configuration time T_C resulted from datapath merging algorithms in Mediabench applications

Benchmark (Number of DFGs)	T_C in [7] (ms)	T_C in [14] (ms)	T_C in Proposed Algorithm (ms)
Epic-decoder(3)	11.04	6.39	5.82
Epic-coder(3)	4.87	2.66	2.49
Mpeg2-decoder(3)	5.83	4.11	3.64
Mpeg2-coder(3)	7.51	5.68	4.87
G721(2)	10.6	6.39	5.82

To apply the proposed technique and previous datapath merging technique to DFGs, initially each DFG was scheduled using As Soon As Possible (ASAP) scheduling algorithm in advance. Later, the proposed datapath merging technique and datapath merging technique in [14] were added to the scheduled DFGs to archive their merged datapaths. To implement the conventional HLS technique in [7], each DFG was scheduled in advance and the resources inside the DFG were shared using the Integer Linear Programming (ILP) algorithm afterward to obtain the datapath. The configuration time of each datapath, and the merged datapaths were calculated based on the configuration clock frequency of the FPGA.

Table 1 illustrates the configuration time to map the kernels on reconfigurable fabric for each benchmark resulted from applying the proposed technique and the techniques in [14] and [7] to DFGs. As illustrated in Table 1, the datapath merging techniques have lower configuration time in general and the proposed technique has the least configuration time. The main reason for this is the shorter length of the generated bit-stream by the proposed technique. For G721, there are two DFGs in the benchmark and the improvement of the proposed technique and previous merging algorithm is the same.

Fig.4 shows the configuration time reduction percentage of the DFGs after applying the above mentioned-techniques to each benchmark where the FPGA Xilinx Virtex5-xc5vlx is the target platform. As illustrated in this figure, there is a substantial difference between the results of the datapath merging techniques and conventional HLS technique in [7] in terms of the configuration time reduction. The proposed datapath merging technique lowered the configuration time up to 50% in comparison to the technique in [7]. On the other hand, it can decrease the configuration time up to 10% more than the datapath merging technique in [14] for these benchmarks.

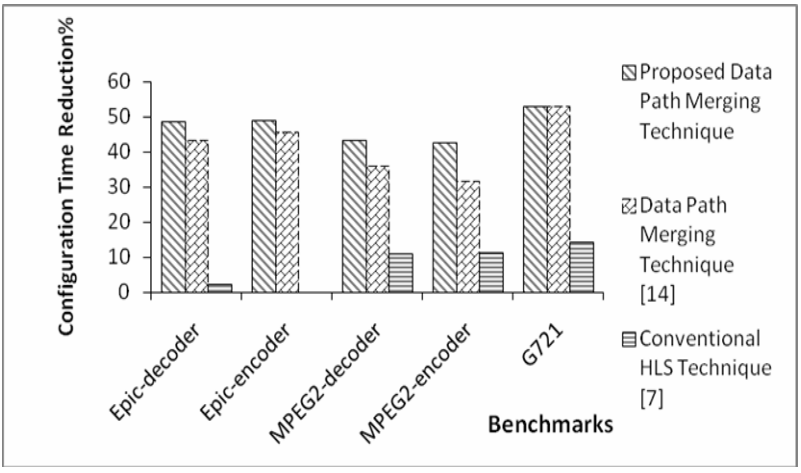


Fig. 4. Percentage configuration time reduction in datapath merging algorithms for Media-bench applications

We conclude that sharing the resources among the DFGs is the main cause of having shorter bit-stream. In the same line, merging more resources in the proposed datapath merging technique results in more reduction in configuration time compared to the previous datapath merging technique. The results lead us to the conclusion that the proposed datapath merging approach is suitable for the synthesizer in run-time reconfigurable systems.

5 Conclusion

In this paper a modified datapath merging technique for the reduction of datapath configuration time was introduced. Our proposed technique combines the resources inside the merged datapath to minimize the configuration time. To this end, we set off to find a compatibility graph that indicates to similarity between the resources inside the merged datapath. Afterwards, we determined a maximum weighted clique in the compatibility graph to reconstruct the merged datapath. This is an *NP*-complete problem thus; a Branch&Bound algorithm was employed to solve the problem. Applying the proposed technique to the workloads from the Media-bench suite in terms of the configuration time reduction, shows an improvement up to 50% in comparison to the conventional HLS algorithm in [7] and an improvement up to 10% in comparison to the previous datapath merging technique in [14]. Overall, the proposed merging technique in this paper can efficiently decrease the configuration time in RTR systems. It should be mentioned that datapath merging approach will increase the kernel execution time; however, the configuration time is measured in milliseconds while the kernel execution time is in nanoseconds. Therefore, the increase in the execution time of kernels is negligible compared to the configuration time reduction. Nonetheless, where kernels have significant number of iterations, this time overhead (increase in the execution time of the kernels) is comparable to the configuration time reduction and we should consider it in datapath merging. Furthermore, we will investigate this factor in datapath merging.

Acknowledgment

This research was supported by the Iran Telecommunication Research Center (ITRC) in the context of the project T/500/3462.

References

1. Wenyin, F., Compton, K.: An Execution Environment for Reconfigurable Computing. In: Proc. of 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), CA, USA, April 2005, pp. 149–158 (2005)
2. Li, Z.: Configuration Management Techniques for Reconfigurable Computing.: Ph.D. Thesis. Northwestern University (June 2002)
3. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K.L.M., Kuzmanov, G.K., Panainte, E.M.: The Molen Polymorphic Processor. IEEE Transactions on Computers (TC) 53(11), 1363–1375 (2004)
4. Ghiasi, S., Nahapetian, A., Sarrafzadeh, M.: An Optimal Algorithm for Minimizing Runtime Reconfiguration Delay. ACM Transactions on Embedded Computing Systems (TECS) 3(2), 237–256 (2004)
5. Rollmann, M., Merker, R.: A Cost Model for Partial Dynamic Reconfiguration. In: Proc. of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), Greece, July 2008, pp. 182–186 (2008)
6. Farshadjam, F., Dehghan, M., Fathy, M., Ahmadi, M.: A new compression based approach for reconfiguration overhead reduction in Virtex-based RTR systems. Elsevier Journal on Computers &Electrical Engineering 32(4), 322–347 (2006)
7. Qu, Y., Tiensyrj, K., Soininen, J.P., Nurmi, J.: Design Flow Instantiation for Run-Time Reconfigurable Systems. EURASIP Journal on Embedded Systems (TECS) 2(11), 1–9 (2008)
8. Yankova, Y.D., Kuzmanov, G.K., Bertels, K.L.M., Gaydadjiev, G.N., Lu, Y., Vassiliadis, S.: DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator. In: Proc. of 17th International Conference on Field Programmable Logic and Applications (FPL), Amsterdam, The Netherlands, August 2007, pp. 697–701 (2007)
9. Meeuws, R.J., Yankova, Y.D., Bertels, K.L.M., Gaydadjiev, G.N., Vassiliadis, S.: A Quantitative Prediction Model for Hardware/Software Partitioning. In: Proc. of 17th International Conference on Field Programmable Logic and Applications (FPL), Amsterdam, The Netherlands, August 2007, pp. 735–739 (2007)
10. Coussy, P., Morawiec, A.: High-Level Synthesis from Algorithm to Digital Circuit. Springer, Heidelberg (2008)
11. Chiou, L., Bhunia, S., Roy, K.: Synthesis of Application-Specific Highly Efficient Multimode Cores for Embedded Systems. ACM Transaction on Embedded System Computing (TECS) 4(1), 168–188 (2005)
12. Moreano, N., Borin, E., Souza, C.D., Araujo, G.: Efficient Datapath Merging for Partially Reconfigurable Architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuit and Systems (TCAD) 24(7), 969–980 (2005)
13. Fazlali, M., Fallah, K.F., Zolghadr, M., Zakerolhosseini, A.: A New Datapath Merging Method for Reconfigurable System. In: Proc. of 5th International Workshop on Applied Reconfigurable Computing (ARC), Karlsruhe Germany, March 2009, pp. 157–168 (2009)
14. Fazlali, M., Zakerolhosseini, A., Sabeghi, M., Bertels, K.L.M., Gaydadjiev, G.: Datapath Configuration Time Reduction for Run-time Reconfigurable Systems. In: Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas Nevada, USA, July 2009, pp. 323–327 (2009)

15. Garey, M., Johnson, D.S.: Computers and Intractability-A Guide to the Theory of NP Completeness. Freeman, San Francisco (1979)
16. Ostergard, P.R.J.: A New Algorithm for the Maximum-Weight Clique Problem. Nordic Journal of Computing (NJC) 8(4), 424–436 (2002)
17. Lee, C., Potkonjak, M., Mangione, W.S.: Mediabench: a Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In: Proc. of 13th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), California, USA, December 1997, pp. 330–335 (1997)

Reconfigurable Computing Education in Computer Science

Abdulhadi Shoufan[1] and Sorin Alexander Huss[2]

[1] Center for Advanced Security Research Darmstadt CASED, Germany
abdul.shoufan@cased.de
[2] Integrated Circuits and Systems Lab, Department of Computer Science
Technische Universität Darmstadt, Germany
huss@iss.tu-darmstadt.de

Abstract. Teaching reconfigurable computing to computer science students demands special attention due to limited student experience in electronics and digital system design. This paper presents a compact course on reconfigurable processors, which was offered at the Technische Universität Darmstadt, and is intended for instructors aiming to introduce a new course in reconfigurable computing. In contrast to courses on digital system design, which use FPGAs as a case platform, our course places this platform at the centre of its focus and highlights its features as a basis for reconfigurable computing.

1 Introduction

With their increasing integration densities and their computational capabilities, FPGAs are invading computer architectures considerably, for example, as application accelerators [1]. Thus, understanding system design with FPGAs is not limited to students focused primarily on hardware design. Rather, students focused on software and algorithms, such as most computer science majors, will benefit from these architectures to build high-performance coprocessors, which serve several applications due to reconfigurability. To qualify those students for this field, special courses on reconfigurable computing should be offered. Unfortunately, most related work on teaching configurable hardware is either specific to students of electrical engineering with corresponding previous knowledge [2,3], or uses FPGA platforms to teach different topics such as digital design foundations and computer architectures [4,5,6,7] with corresponding abstractions from the FPGA platform. This paper describes a course on reconfigurable processors, which we have offered in the computer science department at Technische Universität Darmstadt (TUD) since 2002. In contrast to other courses, which use the FPGA as a case platform, our course places this platform at the centre of investigation as an enabler of reconfigurable computing. This investigation ranges from the effect of a switch transistor on timing behaviour to using embedded processors on FPGAs to set up systems-on-chip. We believe that this thorough learning of today's FPGA platforms is a precondition, not only for its usage in

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 329–336, 2010.

applied reconfigurable computing, but also for understanding the present limits of these platforms as a motivation for further research work in the field of reconfigurable computing. The course has the following didactical features:

1. The course comprises 13 90-minute lectures and 6 labs, which corresponds to a total of 4.5 credit points according to the European Credit Transfer System (ECTS). One credit point in ECTS corresponds to 25 to 30 working hours by students [8]. The presented course assumes a workload of about 135 hours, where 20 hours thereof are lecture time. The remaining hours are spent in labs and self-study.
2. The course is structured based on a new What-Why-How Model (WWH-Model) which presents a guiding theme for all course topics. By this means, instructors are able to extend selected topics or to add new ones without affecting the logical and didactical structure of the course. In the scope of constructive alignment [9], the WWH-Model can be employed to define the intended learning outcomes.
3. The course aims at increasing both the width of declarative knowledge by the variety of lecture topics and the depth of functioning knowledge by lab design assignments.
4. The lectures are supported by interactive slides, demonstrations using commercial tools, and by an excursion in a semiconductor fabrication plant at the Institute for Semiconductor Technology at TUD.
5. Student learning outcomes are evaluated both by formative assessment in the labs and by a summative assessment by a written exam.

Section 2 points out the importance of reconfigurable computing in education. Section 3 describes the structure of the course reconfigurable processors at TUD.

2 Reconfigurable Computing in Education

2.1 Why Teaching Reconfigurable Computing

Reconfigurable computing refers to the computing on a reconfigurable platform, which is currently an FPGA, but can include other adaptable fabric. Many computer scientists consider FPGAs to be more than just hardware chips for fast prototyping. Due to a performance boost of several orders of magnitude and, at the same time, power saving of more than one order of magnitude, some researchers use the concept of reconfigurable computing even to express a paradigm shift in computing, which could or should replace computing paradigm based on von Neumann architecture [10]. Whatever the understanding of reconfigurable computing, its original benefit results from combining two features which are never combined by other architecture solutions such as array processors, supercomputers, and graphical processing units: High performance due to hardware capability and economic efficiency due to reconfigurability, which allows using the same platform for many computing purposes. Thus, reconfigurable computing aims at constructing high-performance application specific processors or

coprocessors which share the same platform. To achieve this aim, researchers and industries have been trying to answer an allocation question, i.e., what the reconfigurable platform should look like? and a mapping question, i.e., how we should map applications onto the reconfigurable platform?

Understandably, the allocation question is more critical as the reconfigurable platform is the technological enabler of reconfigurable computing. Nevertheless, answering this question was always restricted by industrial concerns about economies of scale. Despite several proposals on novel platforms mostly featuring coarser granularity [11], FPGAs remain the most economic choice to support reconfigurable computing. Given the reconfigurable platform, the mapping question largely amounts to a platform-based design process, which has large similarities with other platform-based design approaches. The difference is that the design approach for reconfigurable computing must consider several applications or several application portions sharing the platform. The dream of a platform-independent design approach of reconfigurable systems will remain a dream as long as the more modest dream of Electronic System-Level design (ESL) is not brought to reality.

Knowing its importance and its challenges, which were recently recognized by the Association for Computer Machinery by establishing the Transactions on Reconfigurable Technology and Systems journal [12], it is evident that learning reconfigurable computing at universities is essential for its success.

2.2 Reconfigurable Computing in IEEE/ACM Curriculum

Obviously, reconfigurable computing is a field of computer engineering. The IEEE/ACM curriculum in computer engineering in its last report [13], however, did not address this field explicitly. This can simply be seen by the fact that neither the word *reconfigurable* nor any of its derivatives appear anywhere in this report. Although this situation seems to be advantageous regarding the freedom in defining the learning outcomes for a new course on reconfigurable computing, a reasonable amount of care should be exercised to keep in line with pedagogical precepts of the IEEE/ACM curriculum and to define the scope of such a course in relation to other units in that curriculum, which address FPGAs in some or other way. To be more specific, the IEEE/ACM curriculum in computer engineering addresses FPGAs in 3 areas of knowledge and 4 units, as summarized in Table 1.

From this table it is obvious that FPGAs do not take a central role in any unit, as they do in reconfigurable computing. Therefore, it is highly advantageous to introduce a new unit (course) which tackles reconfigurable computing with FPGAs at its center and addresses other aspects of reconfigurable computing based on a thorough understanding of this platform. Putting the FPGA at the center is essential for two reasons. First, it is almost the only reconfigurable platform, which can be deployed in labs to enhance functioning knowledge. Secondly, to understand why and how reconfigurable computing aims at new platforms, it is essential to understand all the details and limits of current FPGAs.

Table 1. FPGA-related topics and learning outcomes in the IEEE/ACM curriculum in computer engineering

Knowledge Area	Unit (i.e. Course)	Topics	Learning Outcomes
Digital Logic	CE-DIG6: Digital systems design	Totally 7 topics. The last one: Programmable logic devices (PLDs) and field-programmable gate arrays (FPGAs), PLAs, ROMs, PALs, complex PLDs.	Totally 4 learning outcomes. The last one: Utilize programmable devices such as FPGAs and PLDs to implement digital system designs.
Embedded Systems	CE-ESY8: Embedded multiprocessors	Totally 4 topics. The last one: Platform FPGAs as multiprocessors	Totally 3 learning outcomes. Not one relates to FPGAs
VLSI Design and Fabrication	CE-VLS5: Semiconductor memories and logic arrays	Totally 10 topics. The 9th one: FPGA and related devices	Totally 8 learning outcomes. Not one relates to FPGAs
	CE-VLS10: Semicustom design technologies	Totally 7 topics. The 6th one: FPGA and related devices	Totally 3 learning outcomes. Not one addresses FPGAs explicitly.

2.3 Reconfigurable Processors Learning Outcomes

Knowing that reconfigurable computing aims at the development of reconfigurable processors or coprocessors, and to highlight the applied aspects of our course, we entitled it "Reconfigurable Processors". Please note that the word 'processor' in our course does not refer to the central processing unit and that any coprocessor is a processor in the end. In the style of constructive alignment [9] we formulate the learning outcomes of our course as follows:

1. Define a reconfigurable processor and explain its main features
2. Compare reconfigurable processors with other processor types
3. Design a reconfigurable processor based on modern FPGAs
4. Identify fine-grain and coarse-grain reconfigurable resources
5. Explain some applications of reconfigurable processors

While learning outcomes 1, 2, 4, and 5 mostly aim at widening the declarative knowledge of students, learning outcome 3 tends to deepen the functioning knowledge by applying practical approaches and commercial tools to design reconfigurable processors on FPGAs. Learning outcome 3 is therefore supported by lab sessions. The width of declarative knowledge helps to define the relation of this course to other courses at TUD, where this knowledge is deepened

or applied, i.e. becoming functioning knowledge. Our course, for instance, describes SystemC as a high-level design approach for reconfigurable computing. The details of SystemC are treated in depth in the course Embedded Systems I.

3 Course Construction

3.1 What-Why-How Model

A course title is the most essential keyword, which often presents a novel and sometimes mysterious concept for students. An early understanding of the dimensions of this keyword is essential for familiarizing students with the new topic. In engineering fields, a course title has often three dimensions which are highlighted by asking questions of the form: what?, why?, and how? (See Fig. 1).

The question "What?" is usually answered based on students' previous knowledge and experience already acquired elsewhere. Instructors may use this question to verify the suitability of the course in some curriculum or to define the prerequisites for the course. The question "Why?" points out the relevance of the new topic. This can normally be answered by presenting some requirements which can not be fulfilled by objects or methods learnt in other courses. The question "How?" addresses the engineering aspect and describes how to put the learnt material into practice. The number of questions and the way they are asked depend strongly on the course topic and on the word choice in the title.

Specific to the course "Reconfigurable Processors", the authors follow the WWH-Model presented in Fig. 1, which is introduced in the first 90-minute lecture to outline the course topics and to derive its structure. The numbers in Fig. 1 refer to the sequence in which the questions are answered in the first

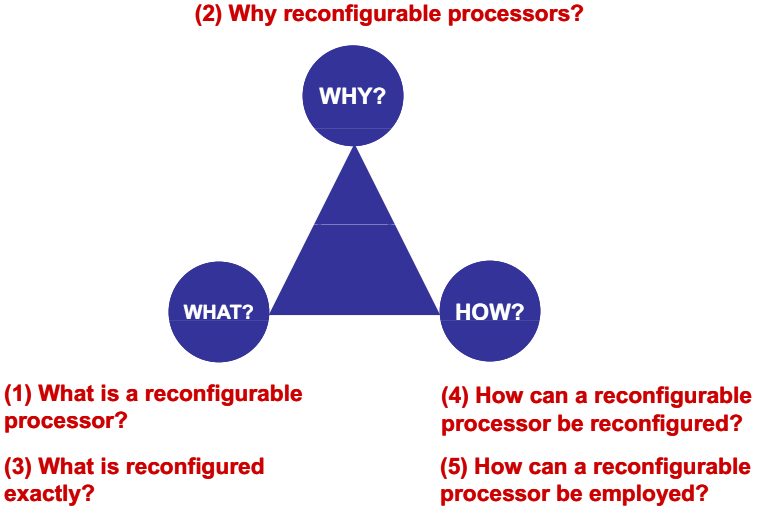


Fig. 1. What-Why-How Model for course construction

lecture for didactical purposes. The first question is answered based on the familiarity of computer science students with the concept of processor: A reconfigurable processor (RP) is a processor which changes its functionality by changing its architecture, as opposed to a general purpose processor (GPP) which never changes its functionality because of its hard-wired architecture. However, GPPs can execute largely different tasks due to the fine granularity of operations they can perform. Any complex task must be divided into a sequence of these fine-grain operations, which are then executed on the GPP sequentially. In contrast, reconfigurable processors perform coarse-grain operations, which make them less flexible than GPPs. This statement introduces the next question: Why a reconfigurable processor? Originally, the advantage of using RPs results from their performance as hardware platforms. Thus, question 2 amounts to: Why is hardware faster than software? A 4-fold answer can be given: Hardware is faster than software because of its ability to support wider data paths, to execute course-grain operations, to use several computational units which operate in parallel or in pipeline mode, and to use internal memories, which accelerates data access. Question 3 addresses the reconfiguration resources. For a processor to be reconfigurable it must contain architecture elements which are changeable. Generally, architectural elements of a processor perform one of three tasks: They either perform computation, interface the processor with outside, or transfer data. Therefore, each reconfigurable processor must contain configurable logic cells, configurable input/output blocks, and configurable routing resources. Reconfiguring these architectural elements demands the writing of new configuration bits. This statement leads to question 4 in Fig. 1, which can now be put as follows: How to generate the configuration bits, how to write them into the reconfigurable processor, and how they are kept on this processor? Generating the bit stream is based on a complex design process, which includes system specification, design entry, simulation, synthesis, placing, routing, and timing analysis. For a general understanding of this design process in the first lecture, we apply it to a simple digital circuit, an adder, based on a VHDL model. The Integrated Software Environment (ISE) from Xilinx is then used to synthesize the code, place and route the netlist, and generate the bit stream which is then written to a Spartan-3 FPGA on a Starter Board. The adder uses some buttons and the DIP-switches on the board to enter the summands. 7-segment displays show the entered summands and the result. Afterwards, the VHDL code is modified slightly to make a multiplier. The design process is started again and the multiplier is demonstrated for persuasion. The last question in Fig. 1 relates to employing a reconfigurable processor and can be answered by giving examples for different applications such as signal processing and cryptography.

3.2 WWH Model and Learning Outcomes

As mentioned earlier, the WWH model can assist in writing learning outcomes. The constructor may ask as many questions as the learning outcomes or vice versa. For our course, Table 2 maps to the questions from the WWH model to the learning outcomes given in Section 2.3.

Table 2. From WWH-Model to learning outcomes

WWH Model Question	Learning Outcome
What is a reconfigurable processor?	Define a reconfigurable processor and explain its main features.
Why reconfigurable processors?	Compare reconfigurable processors with other processor types
How can a reconfigurable processor be reconfigured?	Design a reconfigurable processor based on modern FPGAs
What is reconfigured exactly?	Identify fine-grain and coarse-grain reconfigurable resources
How can a reconfigurable processor be employed?	Explain some applications of reconfigurable processors

Table 3. Reconfigurable processors: lecture structure

Question from WWH-Model	Topic	Lecture No.
All questions in overview	Introduction	1
How can a reconfigurable processor be reconfigured?	Introduction into synthesizable VHDL modeling (1)	2
	Introduction into synthesizable VHDL modeling (2)	3
	Configuration technologies	10
	Partial and dynamic configuration	11
	High-level design of reconfigurable processors	12
What is reconfigured exactly?	Configuration of logic cells	4
	Configuration of I/O blocks	5
	Configuration of routing resources	6
	Configuration of coarse-grain elements	7
Why reconfigurable processors?	Integrated circuits: history and classification	8
	Processors: history and classification	9
How can a reconfigurable processor be employed?	Applications	13

3.3 Lecture Structure

Depending on the WWH-Model the lecture is structured as depicted in column 2 of Table 3. The order of handling these topics given in column 3 presents an alternative which seemed to be reasonable for students in our department. Depending on the situation in other departments some topics may be shifted.

4 Conclusion

We presented a course on reconfigurable processors suitable for students of computer science. In contrast to courses on digital system design, which use FPGAs as a case platform, our course placed this platform in the center of focus and highlights its features as a basis for reconfigurable computing.

References

1. Intel-Corporation: Consistent platform-level services for tightly coupled accelerators, http://download.intel.com/technology/platforms/quickassist/quickassist_aal_whitepaper.pdf
2. Loya-Hernandez, J., Gonzalez-Vazquez, J., Garduno-Mota, M.: Teaching Reconfigurable Hardware Using an Interdisciplinary Problem Based Model to Strengthen Digital Systems Design Skills in Electronic Engineering Undergraduates. In: Proceedings of IEEE International Conference on Microelectronic Systems Education, San Diego, CA (2007)
3. Sklyarov, V., Skliarova, I.: Reconfigurable Systems: Methods, Tools, Tutorials, and Projects. IEEE Transactions on Education 48(2), 290–300 (2005)
4. Teuscher, C., Haenni, J.-O., Gomez, F.J., Restrepo, H.F., Sanchez, E.: A Tool for Teaching and Research on Computer Architecture and Reconfigurable Systems. In: Proceedings of 25th Euromicro Conference, Milant, Italy (1999)
5. Hall, T., Hamblen, J.: Using FPGAs to Simulate and Implement Digital Design Systems in the Classroom. In: Proceedings of 2006 American Society for Engineering Education Southeastern Section Annual Meeting, Tuscaloosa, AL (2006)
6. Stanley, T., Xuan, T., Fife, L., Olton, D.: Simple Eight Bit, Emulated Computers for Illustrating Computer Architecture Concepts and Providing a Starting Point for Student Designs. In: Proceedings of 9th Australasian Computing Education Conference, Ballarat, Victoria, Australia (2007)
7. Torreson, J., Norendal, J., Glette, K.: Establishing a New Course in Reconfigurable Logic System Design. In: Proceedings of 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, Krakow, Poland (2007)
8. ECTS: European credit transfer and accumulation system, users' guide (2009), http://ec.europa.eu/education/lifelong-learning-policy/doc/ects/guide_en.pdf
9. Biggs, J., Tang, C.: Teaching for Qualtiy Learning at University. McGrawHill Education, New York (2007)
10. Zomaya, A.: Handbook of Innovative Computational Paradigms. Springer, Heidelberg (2006)
11. Hauck, A., Dehon, A.: Reconfigurable Computing. Morgan Kaufmann, Burlington (2007)
12. ACM: Transactions on reconfigurable technology and systems, http://trets.cse.sc.edu/index.html
13. IEEE-ACM: Curriculum guidelines for undergraduate degree programs in computer engineering (2004), http://www.eng.auburn.edu/ece/CCCE/CCCE-FinalReport-2004Dec12.pdf

Hardware Implementation of the Orbital Function for Quantum Chemistry Calculations

Maciej Wielgosz, Ernest Jamro, Pawel Russek, and Kazimierz Wiatr

AGH University of Science and Technology,
al. Mickiewicza 30, 30-059 Krakow
ACK Cyfronet AGH
ul. Nawojki 11, 30-950 Krakow
{wielgosz,jamro,russek,wiatr}@agh.edu.pl

Abstract. This paper presents FPGA acceleration and implementation results of a hardware module for generating orbital function. The authors have implemented some of the computationally demanding part of the GPP quantum chemistry source code in FPGA. The orbital function core is composed of the authors' customized floating-point hardware modules. These modules are scalable from single to double precision, capable of working at frequency ranging from 100 to 200 MHz. Besides hardware implementation, the design process also involved reformulation of the algorithm in order to adapt them to the platform profile. The computational procedure presented in this paper is part of an algorithm for generating exchange-correlation potential, and is also recognized as one of the most computationally intensive routines. This feature justifies the effort devoted to develop its hardware implementation.

Keywords: High Performance Reconfigurable Computing, FPGA, quantum chemistry, Custom Computing, HPC.

1 Introduction

The precision of floating-point operations becomes a primary concern when dealing with low-level quantum chemistry procedures, thus the authors have taken various measures to optimize them, both in terms of resource consumption and processing speed. These goals seem to be contradictory, but FPGA technology allows manipulation at the bit level which can be regarded as a huge advantage over GPP and GPU (Graphics Processing Unit) solutions which operate on data of fixed bit-length. The flexibility of the precison adjustment also justifies the choice of VHDL as the design language instead of one of the HLLs (High Level Language).

The proposed implementation employs the RASC platform, a detailed description of which, along with Altix system transfer modes, is covered in [1,7].

2 Algorithm Consideretion

One of the most common and the simplest approximation theories for solving the schroedinger equation is the Hartree-Fock algorithm. The general procedure

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 337–342, 2010.

for solving the Hartree-Fock equation is to make the orbitals self-consistent with the potential field they generate. This is achieved through the self-consistent field (SCF) method [2].

The SCF procedure for solving the Hartree-Fock equation leads to the following equation in matrix formulation:

$$FC - SCE = 0 \tag{1}$$

where F is the Fock-operator, C is the matrix of the unknown coefficients, S is the overlap matrix and E contains energy eigenvalues. All matrices are of the same size.

Solving the Hartree-Fock equation is an iterative process. The coefficients (corresponding to an electric field) are used to build the Fock-operator F, with which the system of linear equations is solved again to get a new solution (a new electric field). The procedure is repeated until a solution reaches a previously established level of accuracy.

The Fock operator depends on orbitals, which in turn are its eigenfunctions. Therefore orbitals have to be calculated for each iteration of the whole Hartree-Fock algorithm. That is why the orbital calculation presented in this paper contributes significantly to the overall performance of the application.

Orbital function is expressed by equation 2 [2]:

$$\chi_{klm}(r) = r_x^k r_y^l r_z^m C_n \sum_i C_i N_i e^{-\alpha_i r^2} \tag{2}$$

where r_x, r_y, r_z and $r^2 = r_x^2 + r_y^2 + r_z^2$ denote atom centered spatial coordinates of each point in the grid. An atom base is represented by C_i and α_i coefficients. The k, l, m indices depend on the type of atom shell (s, p, d or f). C_n and N_i are normalization coefficients.

3 Architecture of the Orbital Module

The RASC accelerator is controlled by the host processor which handles the hardware algorithm execution on the FPGA. Several routines are to be performed by the host processor in order to send data, launch the accelerator and fetch results afterwards. As the FPGA internal memory resources are limited, the maximum single computational data block is parameterized and its size can be adjusted within the range between 128 and 512 of the grid points. The maximum number of atoms composing the molecule is 32. Furthermore it is assumed that the number of atom base coefficients (C_i and α_i) does not exceed 64.

The implementation of the orbital generation function requires both designing hardware modules and software routines. To enable adoption of a modular design approach eq. 2 was decomposed into two sections (similar to [8]) - the exponential part (denoted as EP):

$$\chi_e(r) = \sum_i C_i N_i e^{-\alpha_i r^2} \tag{3}$$

and the polynomial part (PP):

$$\chi_p(r) = r_x^k r_y^l r_z^m C_n \tag{4}$$

where $C_n = 1/3$ if max(k,l,m)=2, 1/15 if max(k,l,m) = 3, 1 otherwise. And shell types are defined by C_n = s if k+l+m=0, p if k+l+m=1, d if k+l+m=2, f if k+l+m = 3.

Both 3 and 4 equations are designed as separate units which make up the orbital function module. The polynomial part (PP) module generates coefficients for the forthcoming atom and at the same time evaluates orbital values for a currently processed atom. Such a pipelined approach requires an EP module to provide the exponential part computed in advance so the PP unit can sustain data processing. Both modules work independently to some degree - controlled by the Fine State Machine (FSM). It should be noted that the same set of polynomial coefficients can be used several times for different EP results. This holds for all the orbitals within the same atom. On the other hand, calculating an EP result takes several clock cycles (one clock cycles per a sum iteration). Therefore for large atoms (composed of many shells), the calculation bottleneck is the EP module. Conversely, for a small one, the calculation bottleneck is in the PP module. As the size of an atom changes, in terms the number of shells, the load balance of the EP and PP modules shifts. A set of FIFO memories have been employed to avoid data transfer stalls between the units and to evenly distribute the load balance for different values.

A uniform input data stream is well suited for FPGA implementation, but unfortunately this is not a case of the algorithm described in this paper. Quantum chemistry complexity is reflected by the diverse structure of input data which imposes some difficulties related to their efficient relocation. Thus a dedicated method of data formatting was introduced and implemented on the host processor to consolidate the data, which are subsequently sent over to the RASC accelerator.

4 Implementation and Acceleration Results

Once the hardware module was implemented on the FPGA, several computational tests were conducted to compare the RASC performance with the Itanium 2 processor. Some of the tests were done for the water molecule calculated in the block composed of 512 point of the three dimensional grid. It took Itanium 2 processor roughly 2885 μs to perform such an operation, which is close to the 3174 μs consumed by FPGA. It is worth noting that the predominate shell of the water molecule is s. Due to the architecture of the hardware module, shell types have an impact on the overall performance. Consequently, an increase of the atom shell size allows full advantage to be taken of LUT and pipeline mechanisms implemented on the FPGA and the accelerated system starts to outperform GPP processor implementation, as depicted in the figures 2,3.

Figure 2 presents various speed-ups achieved for different atom shells dominating in the computations. The atom shell impacts the volume of computations which must be performed to obtain a single orbital result. The number of C_i and

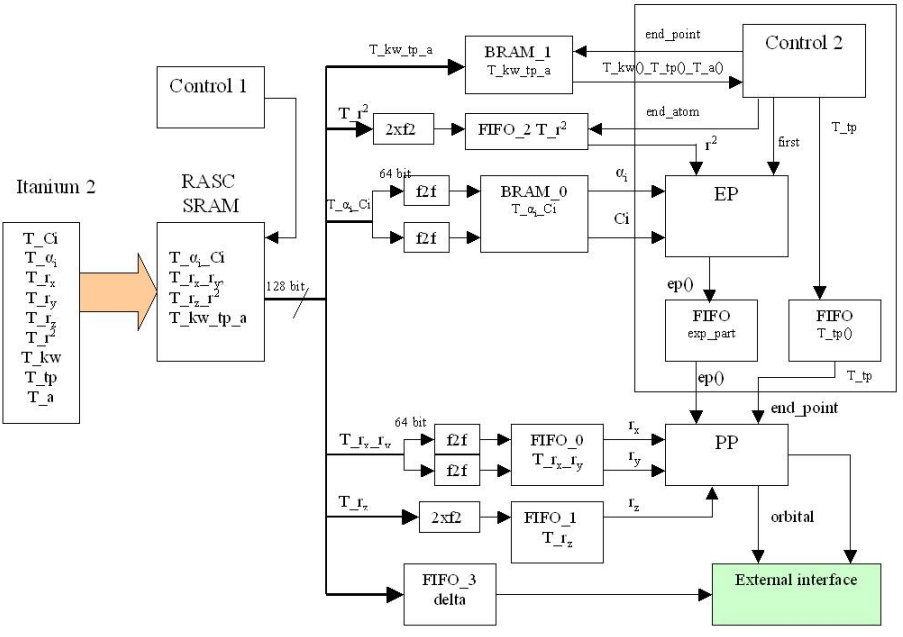


Fig. 1. Block diagram of the orbital generation module

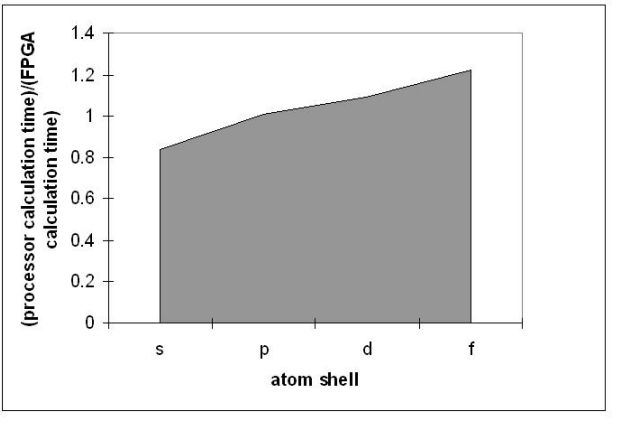


Fig. 2. Impact of the predominating shell on the speed-up (for the constant number of C_i and α_i coefficients = 1)

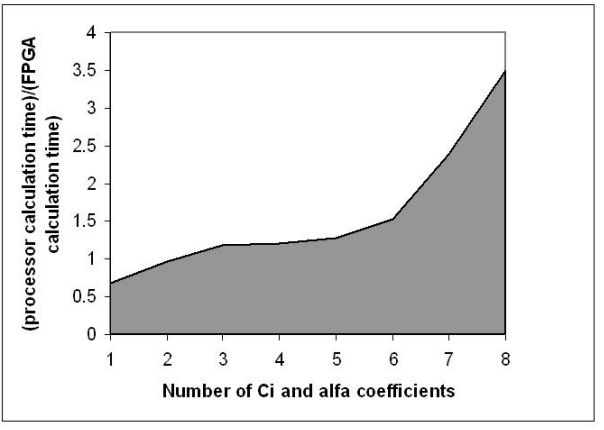


Fig. 3. Impact of the number of atom base coefficients on the speed-up(for the s shell)

Table 1. Implementation results of the Orbital module - single precison

Implementation results	# 4-input LUT	# flip-flops	# 18-Kb BRAMs
Orbital module	8034 (4.5%)	7025 (3.4%)	14 (4.1%)
Orbital module + core services	17365 (9%)	20972 (11%)	37(11%)

Table 2. Primary parameters of the module

Parameter	Value
Frequency [Mhz]	100
Max. Error [ulp]	1
RMSE (Root Mean Square Error)	0,67
Pipeline delay [clk]	64

α_i coefficients also affects the calculation effectiveness(Fig. 3). Discrepancy in the number of C_i and α_i coefficients and the type of atom shell is the main source of dynamic variation of load balance between the EP and PP modules. The ideal case would be an equal number of polynomial C_i and α_i, but unfortunately, this rarely occurs in the computations. Therefore various buffering methods have been employed (Fig.1).

FPGA delivers sustained performance while the processor efficiency drops with the growth of the molecule size. Presented results of a $3,5\times$ speed-up do not seem to be impressive but some enhacements to be implemented are expected to improve the performance of the system. The most important one is a top atom shell prediction mechanism which will eliminate the necessity of generating a complete set of polynomial coefficients for every atom. Only the orbitals which are used for the current calculation will be generated.

FPGA resources consumed by double precision implementation of the orbital module are roughly three times higher than presented in (Tab. 1).

5 Summary

In this paper, a novel approach to generating orbital function in quantum chemistry has been presented. The authors aim to implement the exchange-correlation potential generation and the orbital function is considered to be a milestone on the way to achieving this. On the other hand, the presented implementation was also considered to be a benchmark meant to deliver reliable test results which allow estimation of quantum chemistry acceleration effectiveness on FPGAs. The obtained speed-up is promising and is expected to be higher along with improvements introduced. Furthermore, resource consumption is relatively low due to the 32-40 bit data precision adopted across building units of the orbital module. An additional role of the presented module is also a data serialization for subsequent modules of the system for exchange-correlation potential generating which are expected to contribute significantly to the overall speed-up.

References

1. Silicon Graphics, Inc. Reconfigurable Application-Specific Computing User's Guide, Ver. 005. SGI (January 2007)
2. Mazur, G., Makowski, M.: Development and Optimization of Computational Chemistry Algorithms. In: KDM 2008, Zakopane, Poland (March 2008)
3. Omondi, A.R.: Computer Arithmetic Systems. Prentice Hall, Cambridge (1994)
4. Underwood, K.D., Hemmert, K.S., Ulmer, C.: Architectures and APIs: assessing requirements for delivering FPGA performance to applications. In: ACM/IEEE Conference on Supercomputing, Tampa, Florida, November 11-17 (2006)
5. Gothandaraman, A., Peterson, G., Warren, G., Hinde, R., Harrison, R.: FPGA acceleration of a quantum Monte Carlo application. Parallel Computing 34(4-5), 278–291 (2008)
6. Gothandaraman, A., Warren, G.L., Peterson, G.D., Harrison, R.J.: Reconfigurable accelerator for quantum Monte Carlo simulations in N-body systems. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006, Tampa, Florida, p. 177. ACM, New York (2006)
7. Wielgosz, M., Jamro, E., Wiatr, K.: Accelerating calculations on the RASC platform. A case study of the exponential function. In: Becker, J., et al. (eds.) ARC 2009. LNCS, vol. 5453, pp. 306–311. Springer, Heidelberg (2009)
8. Ramdas, T., Egan, G., Abramson, D., Baldridge, K.: Towards a special-purpose computer for Hartree-Fock computations What's on the table, and how do we take it? Theoretica Chimica Acta 120(1-3), 138 (2008)

Reconfigurable Polyphase Filter Bank Architecture for Spectrum Sensing

Suhaib A. Fahmy[1] and Linda Doyle[2]

[1] School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798
sfahmy@ntu.edu.sg
[2] Trinity College Dublin, College Green, Dublin 2, Ireland
linda.doyle@tcd.ie

Abstract. This paper presents a brief tutorial and background on implementing filter banks for spectrum sensing. It discusses the advantages of this approach over standard FFT-based spectral estimation. A general architecture for implementation of filter banks on FPGAs is then presented, exploiting heterogeneous resources. It takes advantage of the fact that subband filters run at a reduced sample rate, and hence can share the same computational resources. We show how to facilitate this sharing of resources by saving state for each subband. The architecture is fully reconfigurable, allowing sensing parameters to be changed at runtime. Resource usage figures are given, showing the efficiency of the architecture. We finally discuss how this architecture can be adapted to signal reception.

1 Introduction

The Fourier transform (FT) is perhaps the most widely used, and generally applicable component of the signal processing toolbox. It allows us to obtain the frequency domain representation of a time domain signal. Unfortunately, using the FT for signal analysis in DSP systems, we face a problem due to the discrete nature of digital systems. The abrupt edges in the time domain translate into sinc functions in the frequency domain. This results in what is called spectral leakage; where some of the energy in a bin "leaks" into adjacent bins. This gives an inaccurate spectral footprint, especially when adjacent bins have vastly differing amplitudes, as illustrated in Fig. 1. This inaccurate representation can underestimate available spectrum for a cognitive radio. This can be overcome by windowing in the time-domain. Polyphase filter banks are an efficient way of accomplishing this.

Filter banks are an established structure used in multirate signal processing and have applications in compression and image processing. [1] When trying to decompose signals into subbands, we typically modulate a prototype low-pass filter to the appropriate centre frequencies. This requires as many different low-pass filters as there are subbands, and each of these must be run at the input sample rate. Fig. 2 shows a frequency domain representation of the filters.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 343–350, 2010.

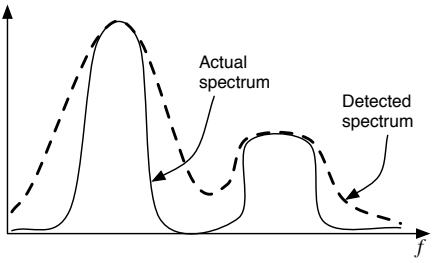


Fig. 1. Spectral leakage using PSD estimation

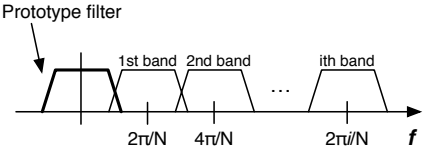


Fig. 2. Frequency representation of a filter bank

Polyphase filter banks enable us to do this more efficiently, by decomposing a single *prototype filter*, into subfilters and effectively modulating these using an FFT. These filters can then be run at a fraction of the original sampling rate. While filter banks are not new, a number of exciting candidate applications, specifically in the area of cognitive radio, are reawakening research in the area, while demanding much larger dimensions than have been previously required. In this paper, we present a brief tutorial on the theory behind polyphase filter banks, we analyse the computational requirements, then present a reconfigurable FPGA-based architecture for polyphase filter banks for spectrum sensing.

Using filter banks for spectrum sensing offers a number of advantages. Firstly, we can scan wide portions of the spectrum (high input sampling rate), efficiently (using fewer resources at a lower sampling rate). Secondly, the results of spectrum sensing using filter banks are far more accurate than basic energy detection using an averaged FFT. Finally, the use of filter banks for transmission and reception of signals is an area that has gained renewed interest at present. It is envisaged that such techniques could supersede current multicarrier transmission methods. [2] It would then be possible to use the same computational structure for sensing and receiving; an attractive proposition for resource limited cognitive radios. The main contribution of this paper is the scheme by which we are able to share the filtering circuitry between subbands while maintaining state for each of them.

This architecture will be used within an FPGA-based cognitive radio framework [3] to explore aspects of spectrum sensing and wideband transmission; hence the importance of a reconfigurable design. We present a brief background in Section 2, followed by an overview of related work in Section 3. We then present the proposed architecture and implementation results in Sections 4 and 5, respectively.

2 Background

Frequency division multiplexing (FDM), allows us to combine multiple transmission streams by arranging them adjacently in the frequency domain. Typically, we split the overall bandwidth of the FDM signal into N subbands, modulating data onto each subcarrier independently. Protocols like Orthogonal Frequency Division Mutiplexing (OFDM) allow us to modulate several independent streams of data across a wide band, overcoming the weaknesses of narrow band techniques applied to wide bands, in terms of robustness.

One of the primary tasks of a cognitive radio (CR) is spectral estimation, that is, determining portions of the spectrum that can be used for transmission. This allows the CR node to establish communication with other nodes without interfering with primary users. This is most simply done by using an FFT to provide a spectral estimate. However, an FFT only provides an instantaneous representation of the spectrum, so this must be averaged over a number of windows to give a spectral estimate that is of any use. When taking an FFT over a fixed sized window, spectral leakage can occur. This is due to the sharp edges of a square window causing what should be delta functions in the frequency domain to become sinc functions, the sidelobes of which can interfere with adjacent frequency bins.

The standard solution for obtaining more accurate estimates is to use windowing to weight the samples in the input window. Windowing over multiple overlapping windows provides more accurate results, and can be implemented using a polyphase filter bank. [4]. An N-band polyphase filter bank consists of a commutator that switches input samples into one of N subband filters successively. The coefficients of the rth subband filter, $h_r(n)$ are derived from the prototype filter, $h_{\mathrm{P}}(n)$, using the following identity:

$$h_r(n) = h_P(r + nN) \tag{1}$$

Hence, for a prototype filter with $N \times M$ coefficients, decomposed into N bands, each filter has M coefficients. This can be seen as a lexicographic reordering; rearranging the 1-dimensional $h_{\mathrm{P}}(n)$ coefficients row-wise into N rows, then reading back the coefficients column-wise. The outputs of the filters are then connected to an N-point Discrete Fourier transform, which creates the modulation effect on the filter responses. When N is a power of 2, we can use an FFT, for which highly efficient architectures exist. A general polyphase filter architecture is shown in Fig. 3.

Each subband filter only operates on $1/N$th of the input samples. So filter $h_0(n)$ operates on inputs x_0, x_N, x_{2N}, $\cdots$, filter $h_9(n)$ operates on inputs x_9, x_{N+9}, x_{2N+9}, $\cdots$, and so on. These subband filters are thus run at $1/N$th of the input sample rate. [5] This is what makes this method so attractive from the computational perspective: the performance requirement in the subband filters is $1/N$th of that for running them at full sample rate. Or alternatively, we can share the same computational circuitry between the subband filters.

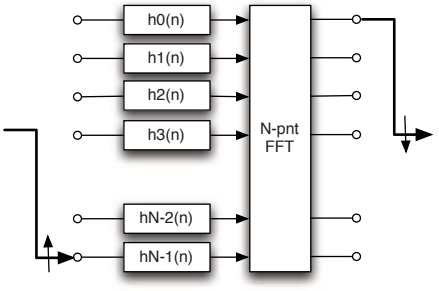


Fig. 3. The polyphase decomposition

3 Previous Work

FPGAs are ideally suited to accelerating filters, due primarily to the provision of embedded multipliers and multiply accumulate (MAC) units on modern devices. Typically, finite impulse-response (FIR) filters are accelerated in hardware by parallelising the MACs; a filter with k coefficients requires k MACs to run at maximum speed. A more compact implementation can use fewer MACs, but at a cost of more clock-cycles per result. Implementing FFTs on FPGAs is straightforward nowadays, with efficient, feature-rich cores provided by device vendors.

Most hardware implementations of polyphase filter banks so far have focussed on much smaller problems. For spectrum sensing or filter bank-based communications, we typically consider 128–2048 subbands and prototype filters of increased dimensions. In [6], a 16-band polyphase filter bank architecture is presented, along with basic optimisations of wordlengths and multiplications. In [7], a wideband channeliser is implemented on a Xilinx Virtex-4 FPGA, to extract frequencies of interest and recombine them in a narrower bandwidth. The number of bands in this implementation is 32, and synthesis results are given. In [8], we are presented with a polyphase filter bank implementation used in MP3 decoding, again with a 32-band decomposition. This implementation is focussed on the MP3 standard, and little discussion of architectural optimisations is presented.

The Xilinx FIR Compiler core [9] does offer a Channeliser mode, however this only allows for a fixed implementation, and the specific architecture used for the implementation is not discussed. We have been unable to identify any work that focusses on polyphase filter bank implementations for spectrum sensing or for reception. The dimensions of the problem differ significantly, and this is what we tackle in this paper. This architecture facilitates much larger prototype filters, a significantly increased number of subbands, and allows for runtime reconfiguration of the filter bank parameters. We also focus on how state can be preserved between application of the computational core to the different subbands in turn.

4 Proposed Architecture

4.1 Overview

Our aim is to create a general filter bank architecture that can be used for both spectrum sensing and for reception as part of a cognitive radio system. The basic structure of polyphase filter banks was discussed in Section 2. An input signal is commutated to a bank of FIR filters, the coefficients of which are derived from the prototype filter using the method explained in Section 2. If we have N subbands, then each subfilter is only activated once for every N inputs. Hence, we can design a compact architecture by sharing the same filter hardware among the subband filters and running it at the input sample rate. The outputs of the filters are fed through to an FFT for combination.

4.2 Efficient Implementation

The challenge in sharing a single filter engine between the subband filters arises because we need to be able to access a new set of coefficients in each clock cycle, and store the intermediate state of the filter once the next sample has been received into the filter. This would allow an N-band filter bank, with M coefficients per subband filter, to consume the same area as an M point FIR filter plus whatever resources are necessary for storing N different states. Fig. 4 shows an overview of the proposed architecture.

Prior to processing, we must load the coefficient memories. To preclude the need for a large memory to store the prototype filter prior to distribution, we distribute the coefficients directly into the combined memories. The *load_coeffs* input is pulsed. In the following cycle, the filter coefficients are fed in, one per clock cycle on the *coeff_in* input. The values of *num_bands* and *num_sbcoeffs*

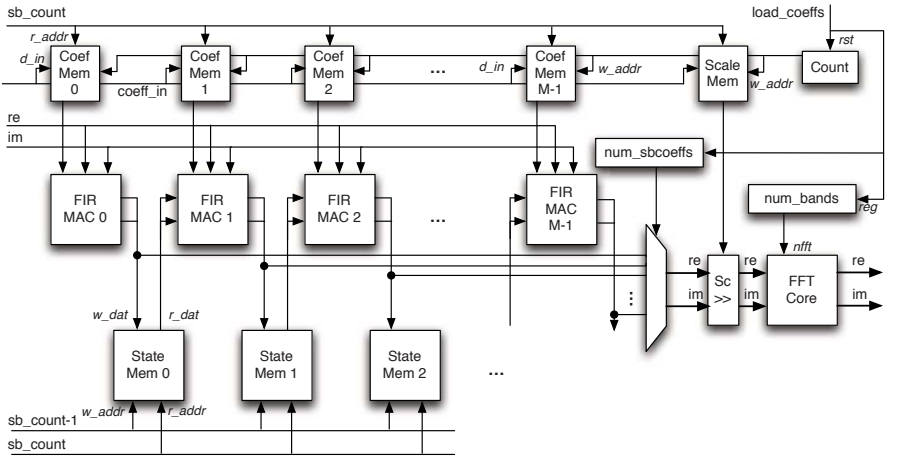


Fig. 4. Proposed architecture

determine how the memories are written to. A shift register with a single 1 is enabled when loading coefficients. This is connected to the write enable signal of each of the memories, and shifts in each cycle, writing into subsequent memories. *num_sbcoeffs* determines when this shift register resets to the first index, hence writing into the second address of the first memory. A counter that counts up after each complete row of coefficients serves as the write address. Once the coefficients are loaded, the circuit switches to processing mode, which is depicted in Fig. 4.

The filters in our system have real coefficients but operate on complex inputs; effectively, we have a duplicate filter for the imaginary part of the input, as for the real part. We implement the filter core in transposed form. Each MAC has a single cycle latency. To manage the sharing of the coefficients and storage of intermediate data, we keep track of the current subband index in the register *sb_count*. This is a counter which counts up in every clock cycle. This counter is used to address the coefficient memory, loading the correct coefficients for the subband being processed.

Storing the intermediate state of the filter is accomplished in a similar fashion to the coefficient control. Each MAC has an associated state memory, which stores the output of the MAC into a location one cycle-delayed from the coefficient index. While we store the result for one subband, we are loading the state for the next. Since we have dual-port memories on the FPGA, this arrangement does not present any problems. By ensuring the index count loops at the correct maximum count (*num_bands*−1), the data is correctly dealt with from one complete iteration through the bands to the next.

The output of the filters is then rescaled (discussed in Section 4.3), before being passed to the FFT core, the result of which, represents the system output. The serial output of the FFT represents the combination of subband outputs as per the polyphase definition.

4.3 Wordlength Considerations

When dealing with very large filters, we expect to see some very small coefficients. In the absence of floating point arithmetic units, we can either use wide fixed point representations or manage scaling manually. Rather than make the resource sacrifice necessary for wide filters, we choose to take advantage of a property of the subband filters. Since each filter is independent of the next, it is feasible to scale the coefficients of each filter independently of the others for the filtering operation, as long as the scales are restored when the outputs of the subband filters are combined. This gives us a significant area saving, while maintaining filter accuracy.

In this implementation, the input samples have 16-bit signed real and imaginary parts, the coefficients are also in 16-bit signed representation. This allows us to make efficient use of the embedded DSP Blocks on the FPGA. The architecture allows for coefficients to be input as 32-bit numbers along with a scale value. The scale value for each MAC is stored in a memory, and is used at the output of the filter to correctly rescale the samples before they are passed to the

FFT. The scale value must be computed in advance by the user, but is simply a case of finding the maximum dynamic range for each subband filter and scaling by a power of 2.

4.4 Reconfiguration

Changing the number of subbands or the size of each subband filter is facilitated at runtime. The *load_coeffs* signal is simply asserted with new values for *num_bands* and *num_sbcoeffs*. New coefficients are then fed into the system. Such a reconfiguration is applied by changing the count limit for the commutator, adjusting which MAC output feeds into the FFT, and changing the FFT point size (which is a supported feature of the Xilinx FFT Core). In this way, we can dynamically switch the properties of the architecture to suit different sensing and reception requirements.

5 Initial Results

The architecture has been synthesised for the Xilinx Virtex-5 XCVFX70T, as found on the ML507 Development Board which is used for our cognitive radio implementations. The architecture itself can achieve a maximum clock speed of 150MHz (or 125MHz as we exceed 80% device usage), which exceeds the 100MHz bus requirement for our implementation framework. We initially provide results for a reconfigurable implementation with a maximum subband count of 1024.

Resource usage can be estimated in advance, based on the maximum allowable values for *num_bands* and *num_sbcoeffs*. While these values can be changed at runtime, we must set limits, which determine the resource usage. The maximum number of subbands determines the depth of the memories used for storing coefficients and state information. The maximum number of subband coefficients determines the number of MAC units and hence the number of memories required. For 16-bit coefficients, we can store up to 1024 entries in a single 18Kb Block RAM. Note that the Block RAM count returned by the tools for a Virtex-5 device counts 36Kb Block RAMs (which can be used as two 18Kb Block RAMs). Hence, reducing the number of bands does not reduce the memory usage for storing the coefficients.

The state memories store 64-bit data (concatenated 32-bit real and imaginary parts), hence for 1024 bands, we would require four 18Kb (or two 36Kb) Block RAMs per state memory. For 512 bands, we would require two 18Kb (one 32Kb) Block RAMs, and for 256 or fewer bands, we would require one 18Kb Block RAM per state memory. We expect logic usage of the filter portion not to change with a reduced number of bands, as the only difference will be the width of the address counters.

The design's area is dominated by BlockRAM and DSP48E usage, though the FFT core adds considerable logic usage. For a 1024 band implementation, we would be restricted to a maximum filter size of 57 taps, though we expect the clock speed to be reduced when nearing 100% usage.

Table 1. FPGA resource utilisation

Filter Coeffs	LUT/FF Pairs	BlockRAMs	DSP48Es
8	1,120	20	16
16	1,760	40	32
32	3,170	80	64
48	4,480	120	96
1024pt FFT	7,100	4	13
Available	44,800	148	128

We believe performance can be increased by further pipelining of the memories and MAC units, though this is not required for our purposes.

6 Conclusion

By exploiting the embedded memories and multipliers on modern FPGAs, we have been able to design an efficient generalised architecture for polyphase filter banks. It allows a single filter structure to be used by alternate subbands in consecutive clock cycles. The architecture scales up to the large dimensions required for spectrum sensing, and due to the memory-based implementation facilitates reconfiguration at runtime. By incorporating this design into our Cognitive Radio framework [3], we can time-multiplex sensing with other functions of a cognitive radio system on a smaller device. The reconfigurability of this architecture makes it ideally suited to explorative research in the area.

References

1. Vaidyanathan, P.: Multirate Systems And Filter Banks. Prentice Hall PTR, Englewood Cliffs (1992)
2. Farhang-Boroujeny, B., Kempter, R.: Multicarrier communication techniques for spectrum sensing and communication in cognitive radios. IEEE Communications Magazine 46(4), 80–85 (2008)
3. Fahmy, S., Lotze, J., Noguera, J., Doyle, L., Esser, R.: Generic software framework for adaptive applications on FPGAs. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA (April 2009)
4. Harris, F.: On detecting white space spectra for spectral scavenging in cognitive radios. Wireless Personal Communications 45(3), 325–342 (2008)
5. Harris, F.: Multirate Signal Processing for Communication Systems. Prentice Hall PTR, Englewood Cliffs (2004)
6. Fiore, P.: Low-complexity implementation of a polyphase filter bank. Digital Signal Processing 8(2), 126–135 (1998)
7. Egg, B., Harris, F., Dick, C.: Ultra-wideband 1.6GHz channelizer: Versatile FPGA implementation. In: Proceedings of the SDR 2005 Technical Conference and Product Exposition (2005)
8. Valdés, M., Moure, M., Diéguez, J., Antelo, S.: Hardware implementation of a polyphase filter bank for MP3 decoding. In: Proceedings of Southern Conference on Programmable Logic (SPL), pp. 19–24 (2008)
9. Xilinx Inc.: FIR Compiler v4.0 Product Specification (DS534) (June 2008)

Systolic Algorithm Mapping for Coarse Grained Reconfigurable Array Architectures

Kunjan Patel and C.J. Bleakley

UCD Complex and Adaptive Systems Laboratory
UCD School of Computer Science and Informatics
University College Dublin, Dublin 4, Ireland
chris.bleakley@ucd.ie, http://www.kunjanpatel.co.nr/contact

Abstract. Coarse Grained Reconfigurable Array (CGRA) architectures give high throughput and data reuse for regular algorithms while providing flexibility to execute multiple algorithms on the same architecture. This paper investigates systolic mapping techniques for mapping biosignal processing algorithms to CGRA architectures. A novel methodology using synchronous data flow (SDF) graphs and control and data flow (CDF) graphs for mapping is presented. Mapping signal processing algorithms in this manner is shown to give up to a 88% reduction in memory accesses and significant savings in fetch and decode operations while providing high throughput.

1 Introduction and Related Work

Biosignal processing is widely used in the field of biomedical engineering. Biosignals are generally one dimensional and multichannel. To perform monitoring without interrupting the patient's daily life, development of portable low power biosignal processing devices is essential especially for implantable devices. A Coarse Grained Reconfigurable Array (CGRA) architecture consists of a grid of interconnected reconfigurable processing units which can perform logical or arithmetic operations. CGRA architectures promise low power consumption and high performance while maintaining high flexibility [1]. Mapping applications to array architectures has been a topic of interest to researchers since efficient algorithm mapping is crucial for achieving high performance. Mapping of some DSP algorithms onto the MONTIUM coarse grained reconfigurable architecture was presented in [2]. Applications were mapped specifically for the MONTIUM architecture and the mapping was performance centric. A Synchronous Data Flow (SDF) graph was presented in [3] for mapping and scheduling applications to parallel DSP processors. It showed the usability of the SDF graph for concurrent and automatic scheduling for parallel processors. A Cyclo-Static Data Flow (CSDF) graph was presented in [4]. It allowed static scheduling of high frequency DSP algorithms in multi-processor environments. However, it is not always possible to find repeatable finite schedules [5].

This paper presents a novel two layer data flow graph approach to map biosignal algorithms in a systolic manner to CGRA architectures. There are two main

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 351–357, 2010.

differences between the proposed approach and previously proposed approaches. First, the mapping of algorithms is done for CGRA architecture and hence the mapping is constrained by the architecture. Second, the proposed mappings are focused on low power consumption rather than high performance. Reading data from memory is one of the most power consuming processes in the processor execution cycle [6]. Hence, power consumption is reduced by reducing the number of memory accesses and elimination of the fetch-decode stages of the execution cycle. The high degree of computational parallelism in CGRAs allows for aggressive voltage scaling. Case studies of mapping for various biosignal processing algorithms are provided. To the authors' knowledge, this is the first time that systolic style mapping for CGRA architectures has been described and evaluated.

2 Proposed Algorithm

To map an algorithm in a systolic manner onto a CGRA architecture requires spatio-temporal mapping of the algorithm. To address this problem, the algorithm which is going to be mapped is first presented as a Synchronous Data Flow (SDF) graph [3] which is a very abstract view of the application and then each node of the SDF graph is presented as a Control and Data Flow (CDF) graph [7]. The algorithm is as follows:

Step 1: *Prepare the SDF graph for the application*

Step 2: *Rearrange SDF graph for systolic mapping*

To map the algorithm in a systolic manner, all of the computing elements should run simultaneously and so the blocking factor (j), which determines the number of nodes run in parallel, should be equal to the number of nodes (n) presented in the SDF graph. A series of rearrangements of the SDF graph needs to be performed until the targeted blocking factor is achieved.

Step 3: *Schedule the SDF graph*

In systolic arrays, each CFU executes an operation in a single cycle and it executes the same operation during every cycle until the CFU is reconfigured or disabled. If i is the index of the node in the SDF graph then scheduling (ψ) will be:

$$\psi_i = \{1\}; \forall i \tag{1}$$

Step 4: *Prepare CDF graph for each node in SDF graph*

A CDF graph for each node in the SDF graph is prepared. As mentioned before, each CFU operation must be finished in a single cycle. So, this phase is dependant on the architecture of the CFU. If a CFU is not able to execute the operation in a single cycle then the operation will be divided into smaller operations and the mapping process is repeated from Step 1.

Step 5: *Get the topology matrix and delay matrix*

The topology matrix for the SDF graph is prepared. The operations of nodes are allocated to CFUs according to the connectivity in the topology matrix (Γ). This operation allocation task is dependant on the interconnection topology of the array and topology matrix act as a guide for this purpose. Because a systolic mapping requires synchronization in data injection, the delay matrix is prepared for applications where the data is not injected at the same time in all CFUs.

There are no specific rules in the SDF graph paradigm to constrain the number of I/O ports in the node. So, to keep the mapping constrained to the number of I/O ports in the CFU and since $j = n$, the following conditions should be satisfied for the topology matrix.

Condition 1 (for binding number of output ports):

$$\sum_{i=1}^{R} p_{in} \not> \text{total number of outputs in a CFU}$$

where R = number of arches in the SDF graph and p = number of produced tokens at the node.

Condition 2 (for binding number of input ports):

$$\sum_{i=1}^{R} c_{in} \not> \text{total number of inputs in a CFU}$$

where c = number of consumed tokens at the node.

3 Application Mapping

The algorithms listed in Table 1 were manually mapped to CGRA as shown in Figure 1. The CFU model is shown in Figure 1(a). Each CFU in the CGRA can perform one of the five operations, multiply accumulate (MAC), multiply subtract (MSUB), addition (ADD), subtraction (SUB) or no operation (NOP). The array architecture has interconnections as shown in Figure 1(b).

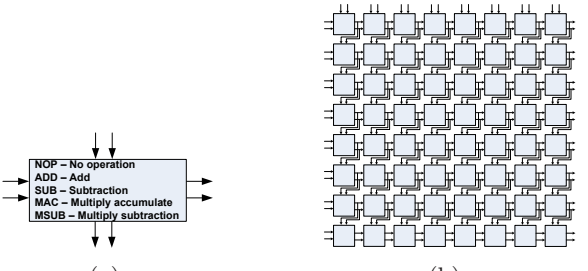


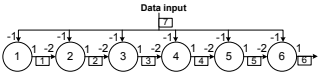
Fig. 1. a) A model of the considered CFU; b) A CGRA architecture example

4 Results

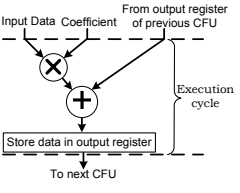
All the algorithms described above were modelled and simulated using a Configurable Array Modeller and Simulator (CAMS) [8] for CGRA architectures. CAMS is a cycle accurate functional simulator for CGRA architectures written in the Java programming language. For filters, coefficients were determined using Matlab and the results were verified against Matlab. The performance figures for the CGRA were compared with those for the TI C5510. For TI C5510 DSP processor, the results were derived from manual and mathematical analysis of equations from [9]-[10]. Table 1 shows a comparison in terms of the number of operations and the number of CFUs required to map the algorithms discussed in the previous section. The number of operations required for the CGRA architecture and DSP is almost same in all the cases. However, using the CGRA architecture, higher throughput can be achieved for continuous data processing because of parallelism and systolic mapping.

$$\Gamma = \begin{bmatrix} 1 & -2 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

(a)



(b)



(c)

Fig. 2. a) 5 taps FIR filter SDF graph; b) FIR filter CDF graph for a single CFU

Table 1. Peformance of some common biosignal applications

Algorithm	Iterations	Total Operations (for single iteration)		Number of CFUs
		CGRA	DSP	
FIR filter	256	6	6	6
Matrix Multiplication	25	67	64	16
Matrix Determinant	25	15	17	5
FFT Butterfly	25	9	8	8
Wavelet Filterbank	256	8	8	10
DFT	8	61	61	61

Table 2 shows the number of register accesses (RGA) and the number of RAM accesses (RMA) required for the CGRA architecture and DSP. An improvement in RMA of up to 8.5 times can be seen because of the systolic mapping.

Table 2. Register and RAM accesses comparison for some biosignal algorithms

Algorithm	CGRA		DSP		Memory access reduction (%)
	RGA	RMA	RGA	RMA	
FIR filter	12	2	12	7	71
Matrix Multiplication	208	42	256	192	78
Matrix Determinant	35	16	26	10	73
FFT Butterfly	25	8	68	5	-60
Wavelet Filterbank	16	2	16	9	77
DFT	188	15	183	130	88

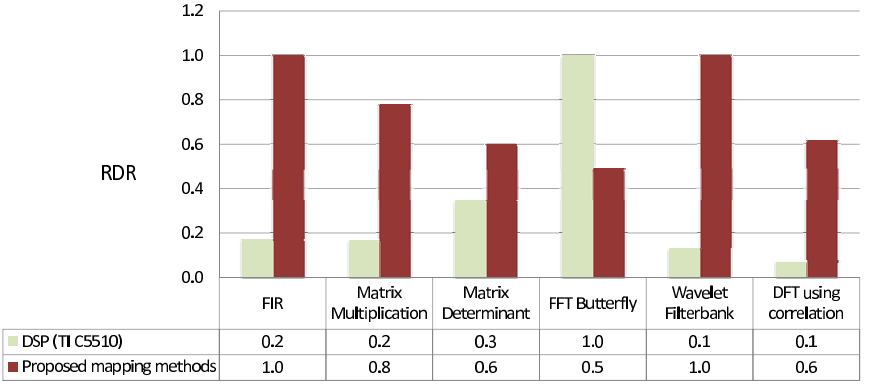


Fig. 3. RDR comparison of some biosignal algorithms

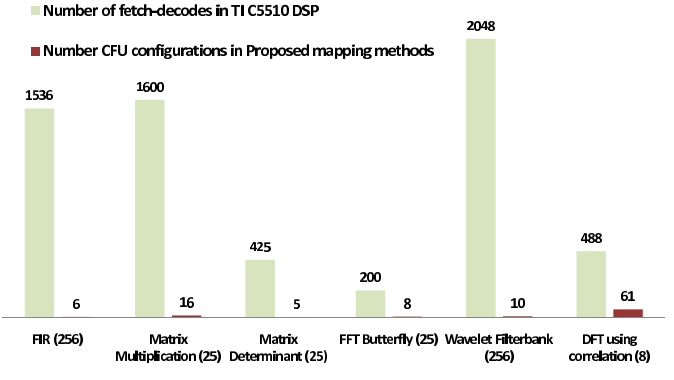


Fig. 4. A comparison of the required number of fetches, decodes and configurations

Figure 3 shows a comparison of RAM Data Reuse (RDR) for all three CGRA architectures. RDR is given by:

$$RDR = \frac{Number\ of\ unique\ RAM\ addresses\ accessed}{Number\ of\ RAM\ accesses} \tag{2}$$

It is clear from the results that data reuse for CGRA architectures is considerably higher than that of DSP processor except for the FFT butterfly.

Figure 4 shows a comparison of the number of fetch-decodes required on a TIC5510 DSP and the number of reconfigurations required for the CGRA architecture to execute the algorithms described before. The number of iterations are shown in brackets. It can be seen that activity can be reduced by avoiding the fetch and decode steps using systolic CGRA architectures for regular biosignal processing algorithms.

5 Conclusion

This paper proposed mapping biosignal applications in a systolic manner onto a CGRA architecture. To map biosignal processing algorithms on the CGRA architecture, two types of graphs, SDF graph and CDF graph, were integrated in the mapping procedure to garner the structure of the signal processing algorithms. This type of signal processing technique shows up to a 88% reduction in memory accesses for regular algorithms compared to that of a conventional DSP. The paper illustrates the efficiency of the proposed approach for low power biosignal applications.

Acknowledgments

This research was funded as a part of the Efficient Embedded Digital Signal Processing for Mobile Digital Health (EEDSP) cluster, grant no. 07/SRC/I1169, by Science Foundation Ireland (SFI).

References

1. Hartenstein, R.: Coarse grain reconfigurable architecture (embedded tutorial). In: Proceedings of the 2001 conference on Asia South Pacific design automation, pp. 564–570. ACM, New York (2001)
2. Heysters, P., Smit, G.: Mapping of DSP algorithms on the MONTIUM architecture. In: Proceedings of International Parallel and Distributed Processing Symposium, 6 p. (April 2003)
3. Lee, E., Messerschmitt, D.: Synchronous Data Flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
4. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cyclo-Static Data Flow. In: International Conference on Acoustics, Speech, and Signal Processing, ICASSP 1995, vol. 5, pp. 3255–3258 (May 1995)

5. Parks, T., Pino, J., Lee, E.: A Comparison of Synchronous and Cyclo-Static Dataflow. In: 1995 Conference Record of the Twenty-Ninth Asilomar Conference on Signals, Systems and Computers, October 1- November 1995, vol. 1, pp. 204–210 (1995)
6. Casas-Sanchez, M., Rizo-Morente, J., Bleakley, C.: Power Consumption Characterisation of the Texas Instruments TMS320VC5510DSP. In: Paliouras, V., Vounckx, J., Verkest, D. (eds.) PATMOS 2005. LNCS, vol. 3728, pp. 561–570. Springer, Heidelberg (2005)
7. Namballa, R., Ranganathan, N., Ejnioui, A.: Control and Data Flow Graph Extraction for High-Level Synthesis. In: IEEE Computer Society Annual Symposium on VLSI, p. 187 (2004)
8. Patel, K., Bleakley, C.J.: Rapid Functional Modelling and Architecture Exploration of Coarse Grained Reconfigurable Array Architectures (2009) (Submitted, under review)
9. Smith, S.: Digital signal processing: a practical guide for engineers and scientists. Newnes (2003)
10. Meher, P.: Efficient Systolic Implementation of DFT Using a Low-Complexity Convolution-Like Formulation. IEEE Transactions on Circuits and Systems II: Express Briefs 53(8), 702–706 (2006)

A GMM-Based Speaker Identification System on FPGA

Phak Len Eh Kan, Tim Allen, and Steven F. Quigley

School of Electronic, Electrical and Computer Engineering,
University of Birmingham, Edgbaston,
Birmingham B15 2TT,
United Kingdom
{exp692,s.f.quigley}@bham.ac.uk

Abstract. Speaker identification is the process of identifying persons from their voice. Speaker-specific characteristics exist in speech signals due to different speakers having different resonances of the vocal tract and these can be exploited by extracting feature vectors such as Mel frequency cepstral coefficients (MFCCs) from the speech signal. The Gaussian Mixture Model (GMM) as a well-known statistical model then models the distribution of each speaker's MFCCs in a multidimensional acoustic space. The GMM-based speaker identification system has features that make it promising for hardware acceleration. This paper describes the classification hardware implementation of a text-independent GMM-based speaker identification system. A speed factor of 90 was achieved compared to software-based implementation on a standard PC.

Keywords: Speaker Identification, MFCC, GMM, Field Programmable Gate Array (FPGA).

1 Introduction

Speaker recognition is the process of automatically recognizing who is speaking by using speaker-specific information included in the speech waveform [1]. It is receiving increasing attention due to its practical value, and has applications ranging from police work to automation of call centres. Speaker recognition can be classified into speaker identification (discovering identity) and speaker verification (authenticating a claim of identity).

Traditionally, most speaker identification systems have been based on software running on a single microprocessor. The problem with software is that its sequential operation means that it can be slow for high throughput real time signal processing applications. FPGAs have been used in many areas to accelerate algorithms by exploiting pipelining and parallelism in a much more thorough way than can be done using general-purpose microprocessors. To date, most attempts to apply FPGA processing to speech problems have focused on the problem of speech recognition [2-6]. Relatively few researchers have investigated the problem of hardware implementation of speaker identification [7] and these investigations have not aimed to achieve large speed-ups.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 358–363, 2010.

This paper presents results for the implementation of a text-independent, closed-set speaker identification classification system on a platform consisting of an Alpha Data RC2000 PCI card equipped with a single Xilinx Virtex-II XC2V6000 FPGA. The goal was to achieve a system that can process a large number of voice streams simultaneously in real time.

2 Speaker Identification System

A block diagram shown in figure 1 is the top-level system designed to implement speaker identification. The input speech is sampled and converted into digital format. Feature vectors are extracted from the input speech in the form of MFCCs. Training is performed offline and results in a set of stored speaker models that are stored in RAM. The hardware implementation processes input speech streams, and performs feature extraction and classification.

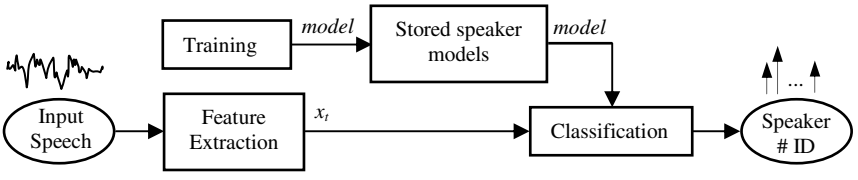


Fig. 1. Top-level Structure of Speaker Identification System

2.1 Feature Extraction

The purpose of feature extraction is to convert the speech waveform to a set of features for further analysis. The speech signal is a slowly time varying signal and when it examined over a sufficient short period of time, its characteristics are fairly stationary, whilst over long periods of time the signal characteristics change to reflect the different speech sounds being spoken. In many cases, short time spectral analysis is the most common way to characterize the speech signal. Several possibilities exist for parametrically representing the speech signal for the speaker identification task, such as MFCC, Linear Prediction Coding (LPC), and others. In this work MFCCs are chosen. Figure 2 shows a block diagram of the MFCC feature extraction. The digital speech signal is blocked into frames of N samples, with adjacent frames being separated by M samples. The first frame consists of the first N samples. The second frame begins M samples after the first frame, and overlaps it by N-M samples and so on. Each individual frame is windowed so as to minimize the signal discontinuity at the beginning and end of each frame.The Fast Fourier Transform (FFT) converts each

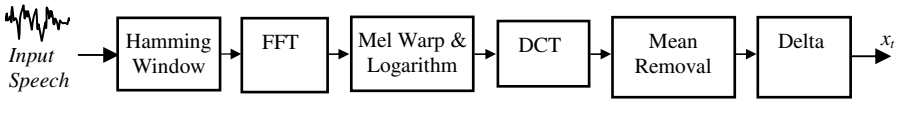


Fig. 2. MFCC Feature Extraction Block Diagram

frame of samples from the time domain into the frequency domain. The frequency scale is then converted from the hertz to the mel scale, using filter banks, with frequency spaced linearly at low frequencies and logarithmically at high frequencies, and the logarithm is then taken. This is done in order to capture the phonetically important characteristics of speech in a manner that reflects the human perceptual system. The Discrete Cosine Transform (DCT) is then applied to the output to produce a cepstrum. The first 17 cepstral coefficients of the series are retained, their means are removed and their first order derivatives are computed. This results in a feature vector of 34 elements, 17 MFCCs and 17 deltas. These vectors (x_t) are then passed on to the classification stage. The logic resources used for the hardware implementation of feature extraction are shown in table 1. Due to the low frequency of speech data and fully pipelined nature of the MFCC datapath, a very large number of speech streams can be multiplexed through one MFCC unit.

Table 1. Logic resources for MFCC Module

Logic Resources	Numbers
Slices	8696
FFs	9187
BRAMs	2
LUTs	16317
IOBs	98
Emb. Mults	1

2.2 Gaussian Mixture Models (GMMs)

The GMM forms the basis for both the training and classification processes. This is a statistical method that classifies the speaker based on the probability that the test data could have originated from each speaker in the set [1,8].

A statistical model for each speaker in the set is developed and denoted by λ. For instance, speaker s in the set of size speaker S can be written as follows

$$\lambda_s = \{w_i, \mu_i, \sigma_i\} \qquad i = 1,\ldots..,M \; ; \; s= 1,\ldots\ldots,S \tag{1}$$

where, w is weight, μ is mean, and σ is diagonal covariance. A diagonal covariance, σ is used rather than a full covariance matrix, Σ, for the speaker model in order to simplify the hardware design. However, this means that a greater number of mixture components M will need to be used to provide adequate classification performance.

The training phase (which is performed offline) makes use of the expectation maximization (EM) algorithm [1,8] to compute the means, covariances and weights of each component in the GMM iteratively.

In the classification stage a series of input vectors are compared and a decision is made as to which of the speakers in the set is the most likely to have spoken the test data. The input to the classification system is denoted as

$$X = \{x_1, x_2, x_3,\ldots\ldots\ldots..,x_T\} \tag{2}$$

The rule to determine if X has come from speaker s can be stated as

$$p(\lambda_s \mid X) > p(\lambda_r \mid X) \quad r = 1,2,\ldots\ldots S\ (r \neq s) \tag{3}$$

Therefore, for each speaker s in the speaker set, the classification system needs to compute and find the value of s that maximizes $p(\ _s|X)$ according to

$$p(\lambda_s \mid X) = \frac{p(X \mid \lambda_s)\, p(\lambda_s)}{p(X)} \tag{4}$$

The classification is based on a comparison between the probabilities for each speaker. If it can be assumed that the prior probability of each speaker is equal, then the term of $p(\lambda_s)$ can be ignored. The term $p(X)$ can also be ignored as this value is the same for each speaker [1], so

$$p(\lambda_s \mid X) = p(X \mid \lambda_s) \quad \text{where, } p(X \mid \lambda_s) = \prod_{t=1}^{T} p(x_t \mid \lambda_s) \tag{5}$$

Practically, the individual probabilities, $p(x_t \mid \lambda_s)$, are typically in the range 10^{-3} to 10^{-8}. With a test input of 10 seconds there are 1000 test vectors. When 10^{-8} is multiplied to itself 1000 times a standard computer and certainly any system implemented on an FPGA will underflow and the probability for all speakers will be calculated as zero. Thus $p(X \mid \lambda_s)$ is computed in the log domain in order to avoid this problem. The likelihood of any speaker having spoken the test data is then referred to as the log-likelihood and is represented by the symbol *L*. The formula for the log-likelihood function is [1]

$$L(\lambda_s) = \sum_{t=1}^{T} \ln(\ p(x_t \mid \lambda_s)) \tag{6}$$

The speaker of the test data is statistically as the speaker s that maximizes the log-likelihood function L.

3 Hardware Implementation of Speaker Classification

The formulas implemented in the datapath of the classification unit are:

$$p(x_t \mid \lambda_s) = \sum_{i=1}^{M} w_i b_i(x_t) \tag{7}$$

$$b_i(x) = \frac{1}{\sqrt{(2\pi)^D \left|\sum_i\right|}} \exp\left(-\frac{1}{2}(x - \mu_i)' \sum_i^{-1} (x - \mu_i)\right) \tag{8}$$

The hardware uses Gaussian mixture models with 32 components. Equation 8 is used to compute log of the probability of each vector having come from a particular component of a given speaker model. This is computed in the log domain by a parallel array of datapaths using dynamically scaled 16-bit fixed point data. In general it is not possible to simultaneously instantiate 32 copies of the datapath, so the data must be multiplexed through a smaller number of computation units. For the

XC2V6000 chip, four copies of the datapath could be instantiated simultaneously. The limiting factor was the number of block multipliers available.

The resulting values for $ln(w_i b_i(x))$ are then fed through log-add blocks to compute equation 7. The log-add logarithm is a method that uses look-up-tables of a modest size to provide an efficient way to compute $ln(A+B)$ in hardware when $ln(A)$ and $ln(B)$ are known [9]. The log-likelihood values $L(\ _s)$ for each speaker in the set are computed through equation 6. Finally a classification is made based on the speaker with the largest log-likelihood value.

4 Testing

The accuracy of speaker recognition was tested using an utterance length of five seconds. Table 2 summarizes these results for both hardware and software. The accuracy is fairly similar with the software system showing some improvement over the hardware system. This is to be expected as the software implementation uses full double precision accuracy. However, the difference is tolerable given the significant speed up achieved in hardware.

Table 2. Hardware and software results for testing with 5s of test utterance

Utterance length	Software – 5 seconds	Hardware – 5 seconds
Test 1	80.77%	78.30%
Test 2	56.40%	55.20%
Test 3	68.54%	64.90%
Test 4	72.75%	69.40%
Mean	69.62%	66.95%
S.D	10.17	9.61
95% confidence	**9.96**	**9.42**

Table 3. Hardware and software timing parameters

Parameter	System	
	Hardware – XC2V6000	Software
Data Transfer	16.8ms for 500 vector transfer	Not applicable
Classification	0.8ms per vector for speaker set of size 20	69ms per vector for speaker set of size 20

The speed of the software system was measured using a speaker set of size 20. Test data from one of the speakers was used and the experiment was repeated 100 times consecutively with an average being computed over the hundred. Table 3 presents the results from the software testing along with the results from the hardware testing on the XC2V6000 board.

The classification part of the speaker identification system implemented on the XC2V6000 platform is 90 time faster than software. When considering real time implementation of speaker identification with feature vectors from one speech input

being provided every 10 ms the software system would only be able to perform calculations on five speaker models.

5 Conclusion

The analysis of hardware versus software has demonstrated that speaker identification classification is about 90 times faster on hardware. This means that the hardware system is capable of processing 90 times more audio streams in real time than could be done in a PC. The limiting factor for implementation on the XC2V6000 device is the number of embedded multipliers and its maximum clock speed.

References

1. Reynolds, D., Rose, R.: Robust Text-independent Speaker Identification using Gaussian Mixture Speaker Models. IEEE Trans. on Speech and Audio Processing 3(1), 72–83 (1995)
2. Melnikoff, S., Quigley, S., Russell, M.: Speech Recognition on an FPGA Using Discrete and Continuous Hidden Markov Models. In: International Workshop on Field-Programmable Logic, pp. 202–211 (2002)
3. Melnikoff, S., Quigley, S., Russell, M.: Implementing a Simple Continuous Speech Recognition System on an FPGA. In: IEEE Symposium on Field Programmable Custom Computing Machines, Los Alamitos, pp. 275–276 (2002)
4. Miura, K., Noguchi, K., Kawaguchi, H., Yoshimoto, M.: A Low Memory Bandwidth Gaussian Mixture Model (GMM) Processor for 20,000-Word Real-Time Speech Recognition FPGA System. In: International Conference on Field Programmable Technology (2008)
5. Yoshizawa, S., Wada, N., Hayasaka, N., Miyanaga, Y.: Scalable Architecture for Word HMM-Based Speech Recognition and VLSI Implementation in Complete System. IEEE Trans. on Circuits and Systems, 70–77 (2006)
6. Lin, E., Yu, K., Rutenbar, R., Chen, T.: A 1000- Word Vocabulary, Speaker-Independent, Continuous Live- Mode Speech Recognizer Implemented in a Single FPGA. In: International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 60–68 (2007)
7. Ramos-Lara, R., López-García, M., Cantó-Navarro, E., Puente-Rodriguez, L.: SVM Speaker Verification System based on a Low-Cost FPGA. In: Field-Programmable Logic and its Applications, pp. 202–211 (2009)
8. Holmes, J.N., Holmes, W.J.: Speech Synthesis and Recognition, 2nd edn. Taylor & Francis, London (2001)
9. Melnikoff, S., Quigley, S.F.: Implementing the Log-add Algorithm in Hardware. Electronic Letters (2003)

An FPGA-Based Real-Time Event Sampler

Niels Penneman[1,2], Luc Perneel[3],
Martin Timmerman[1,3], and Bjorn De Sutter[1,2]

[1] Electronics and Informatics Department, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
[2] Computer Systems Lab, Ghent University
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{niels.penneman,bjorn.desutter}@elis.ugent.be
[3] Dedicated Systems Experts
Bergensesteenweg 421 B12, 1600 St-Pieters-Leeuw, Belgium
{l.perneel,m.timmerman}@dedicated-systems.info

Abstract. This paper presents the design and FPGA-implementation of a sampler that is suited for sampling real-time events in embedded systems. Such sampling is useful, for example, to test whether real-time events are handled in time on such systems. By designing and implementing the sampler as a logic analyzer on an FPGA, several design parameters can be explored and easily modified to match the behavior of different kinds of embedded systems. Moreover, the trade-off between price and performance becomes easy, as it mainly exists of choosing the appropriate type and speed grade of an FPGA family.

Keywords: real-time testing, event sampling, logic analyzer, FPGA.

1 Introduction

Real-time (RT) computing systems have constrained reaction times, i.e., deadlines have to be met in response to events. Ensuring that the constraints are met on a device for a given operating system and set of applications becomes difficult as soon as either of them shows non-trivial behavior. For hard RT systems guarantees have to be provided, which is often done by means of worst-case execution time analysis and by relying on predictable algorithms and hardware [1,2].

Given the criticality of the RT behavior for safety, quality of service, and other user-level requirements, extensive testing of the RT behavior is often performed on a full system. Hence precise methods are needed for observing that RT behavior. Some requirements of these methods are that (1) they should be fast and accurate to allow the correct observation of events happening at high rates, (2) they should be non-intrusive to make sure that the system under test (SUT) behaves as similar to the final system as possible, (3) the methods should be configurable for different measuring contexts, given the wide range of RT deadlines and of application behaviors, and (4) given that relatively few developers will test RT behavior, the required infrastructure needs to be cheap.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 364–371, 2010.

On many embedded systems, the rate at which events occur is so high that software-only solutions cannot meet the first two requirements. This paper presents an FPGA-based event sampler which can be used in a hardware-software cooperative approach. The SUT emits signals through hardware when events occur, which are then captured by the sampler. This system will be (1) fast enough because it runs on an FPGA, (2) reconfigurable by altering design parameters or by choosing different FPGAs, (3) cheap because it does not require high-end FPGAs, and (4) non-intrusive because the required changes to the SUT are minimal. Our proposal is in line with today's use of FPGAs to speed up EDA tasks such as software-hardware co-design and system simulation.

The remainder of this paper is structured as follows. Section 2 provides more background information on the problem of RT event sampling. The design of an FPGA-based sampler is presented in Section 3, after which Section 4 evaluates the performance obtained with this design, and Section 5 draws conclusions.

2 Real-Time Sampling System

The goal of our system is to sample signals emitted by the SUT, timestamp them, and send them to the tester's workstation for further interpretation.

Hardware signals corresponding to events on the SUT have to be emitted with the least possible intrusion on its behavior. Furthermore, the emitted signals have to reach the logic analyzer with predictable, fixed latency; otherwise, precise tracking of RT behavior is not possible. This rules out several existing communication interfaces such as PCI, PCI Express and IEEE 1149.1 JTAG.

By contrast, General Purpose Input Output (GPIO) interfaces can be controlled with fixed latency, using memory-mapped IO through GPIO registers. The adaptation of a RT OS to emit signals via a GPIO interface upon events is then limited to manual instrumentation of the code, adding at most a few instructions per event. Similar uses of GPIO can be found in, e.g., [3,4].

Six commercially available logic analyzers have been evaluated [5,6,7,8,9,10]. None of these devices can sample for a prolonged time with high accuracy. Firstly, most of them sample continuously, instead of only storing samples as events occur. Hence, large amounts of data are generated. Secondly, the available memories tend to be too small to capture a reasonable amount of events.

It is clear that both a large memory and event-based sampling are key to solving our problem. For that reason, our design captures data from eight input channels: four channels for actual test data, two channels for interrupt generation testing, one channel for the SUT to signal an error, and one channel for timestamp counter overflow detection (optionally configurable as external input [11]).

Figure 1 shows an overview of our design. Samples are 32 bits in size, of which 8 map to the input signals, and 24 are dedicated to the timestamp. The timestamp is provided by a counter, operating at the sampling frequency. In order to reconstruct the exact timing, counter overflows are also stored as events. A control block on the data path enables the sampler to start and stop registering events. Users can interact with this block through the management interface. The

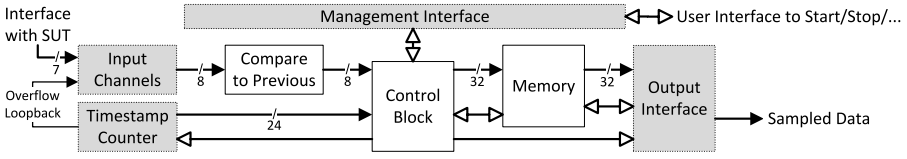


Fig. 1. Conceptual design of the logic analyzer device. Black arrows indicate data edges, while white arrows indicate control signals.

control block may also stop the sampling process whenever errors are detected, and provides reset functionality as a recovery mechanism. The samples stored in internal memory can be retrieved through the output interface.

3 Logic Analyzer Design

This section discusses the different components of our design as a so-called System on a Programmable Chip (SOPC) on Altera's Cyclone III FPGA devices. Although our design will also work on more advanced FPGAs, our design choices will be based on the EP3C25F324C8 FPGA. Some properties of the Cyclone III device family are presented in Table 1.

3.1 Communication with the Workstation

As indicated on the right in Figure 1, the sampler needs to transmit sampled data to the workstation of the tester. Moreover, the tester must be able to control the sampler. Ethernet is fast enough for our purpose and future-proof with respect to commonly used workstation configurations. Although hardware TCP/IP stack implementations exist [12,13], they are either limited in functionality or overly complex in terms of hardware and resource usage. Alternatively, Altera provides a SOPC development environment [14] with ready-to-use components [15,16,17], including the Nios II soft-core CPU and an Ethernet MAC suitable to run a software TCP/IP stack. Opting for this solution requires us to design a custom SOPC. The CPU can then be used to run both the TCP/IP stack and the software control over the whole sampler.

In the structure of the whole RT sampler design, the five components on the top right of Figure 2 implement the Ethernet interface. The Ethernet MAC core has two streaming interfaces, one for transmitting (TX) and one for reading (RX) data. Each of these interfaces needs to be connected to the data memory using

Table 1. Overview of some relevant features of targeted Altera Cyclone III devices

Device	Logic Elements	9kb Memory Blocks	Total RAM Bits	PLLs	Global Clock Networks
EP3C25	24,624	66	608,256	4	20
EP3C40	39,600	126	1,161,216	4	20

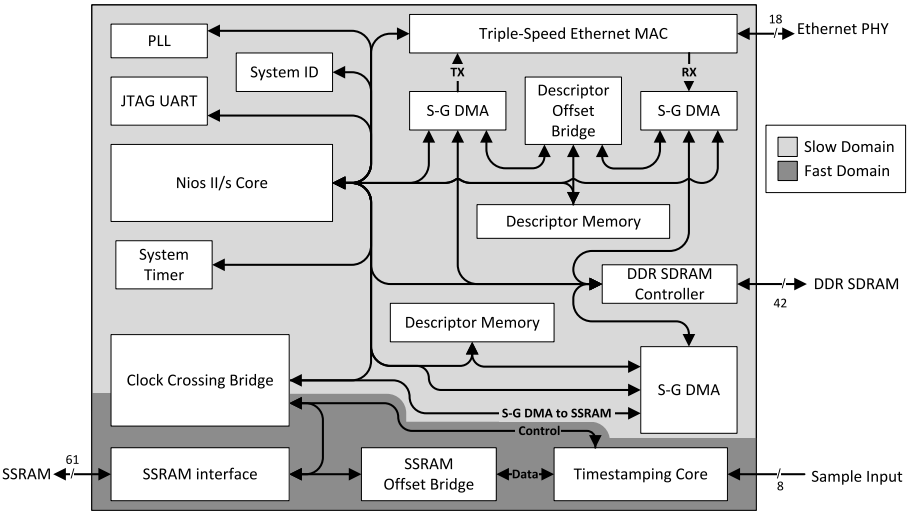


Fig. 2. Structure of the whole logic analyzer design

scatter-gather DMA controller cores. The operation of these cores is defined through DMA descriptors. In order to improve performance, a separate on-chip memory region is allocated to hold these descriptors. The Nios II CPU connects to all of these components using memory-mapped interfaces. The Ethernet MAC and DMA cores implement slave ports and interrupt senders through which they can be controlled and monitored. The CPU also needs access to the DMA descriptor memory to allocate and initialize descriptors.

The software TCP/IP implementation running on the Nios II core controls all this hardware. Output data to be transmitted to the tester's workstation is obtained from external DDR SDRAM via the DDR SDRAM controller. The rationale for this type of memory is explained in Section 3.3.

3.2 Control over the Logic Analyzer

The Nios II CPU, on which the control software will run, comes in three different configurations [16]: economy, standard and fast. We chose the standard configuration: the fast configuration is too large, offering features our software cannot exploit, while the economy version does not offer enough performance.

The software [11] is built on the MicroC/OS-II RT OS [18]. Alternatives are available, such as uClinux [19] and eCos [20]. Their Nios II ports [21] were not considered because they were either outdated or lacked integration support with recent Altera software versions. The TCP/IP stack is provided by InterNiche.

The initial memory footprint of the software varies between 1256 and 1350 kB, depending on how much debug code is included. As detailed in Section 3.3, samples need to be moved from the SSRAM into the SDRAM before they can be

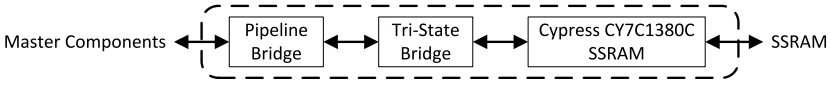


Fig. 3. SOPC components constituting the SSRAM interface

sent over the network. Therefore, a buffer is allocated statically in the SDRAM with the size of the SSRAM, which accounts for 1024 kB of the footprint.

3.3 Memory Architecture

Our logic analyzer requires memories to store the timestamped signals before they are being transferred to the tester's workstation. We opted for an approach with three types of memory: on-chip RAM, SSRAM, and DDR SDRAM.

SSRAM provides burstable and hence predictable storage. The timestamping core acts as a master to this memory to store samples through the interface shown in Figure 3. The DMA controller at the bottom of Figure 2 also acts as a master to transfer samples from the SSRAM to the SDRAM through the clock crossing bridge. Clock crossing bridges add their address as an offset to the address of their slave components. Since Altera requires SOPC designs to use a flat memory model, offset bridges must be introduced to compensate for this behavior.

In order to prevent other components from interfering with the single-port SSRAM, the slower SDRAM is used as general-purpose data memory. This means that the Ethernet interface also gets its data from the SDRAM. Samples are transferred to this SDRAM from the SSRAM in a fast and predictable manner by the DMA controller, controlled by the software. By filling the buffer internal to the timestamping core instead of writing samples directly to the SSRAM, the SSRAM port can be freed to allow transfers to the SDRAM. Because the software controls both processes (unlike the Ethernet interface, which is also dependent on external factors), it is capable of scheduling the reads and writes to the SSRAM without blocking the timestamping core unnecessarily.

3.4 Timestamping Core

The gateware for the actual timestamping needs to fit in Altera's SOPC model to allow interaction with other components such as the CPU and memory devices. For that reason the timestamping core is designed as a custom component following the Altera SOPC standards. This core samples external signals, adds timing information and manages an integrated on-chip buffer.

The pipelined data flow within the timestamping core is depicted in Figure 4. The timestamp counter is continuously incremented at the sampling frequency. First, raw input from the FPGA pins is combined with the value of the timestamp counter into a sample. Next, the input bits are compared with the relevant bits of the previous sample. When sampling is enabled and the inputs are different, the new sample is enqueued in the on-chip FIFO buffer.

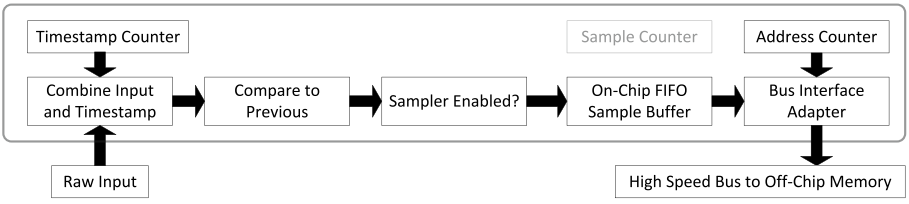


Fig. 4. Pipelined data flow within the timestamping core

The on-chip FIFO buffer provides independent read and write ports. Adapter logic enables the read side to operate as a memory-mapped bus master. The address to write to in the SSRAM is provided by the address counter, which is incremented on each write operation. A separate sample counter keeps track of the number of samples written, and is incremented at the same time.

Due to the design of the memory-mapped master interface, the address counter starts just before the base address of the external memory. While the number of stored samples could be derived from the rightmost 19 bits of the destination address, doing so would require a 19-bit addition. Although reporting on this number is generally not a critical operation, detecting that the off-chip memory is full is however critical. In the separate sample counter, bit 19 indicates this condition. Therefore, using separate counters outperforms the solution with a 19-bit addition.

3.5 Clock Domains and Other Logic

The whole design was split in two clock domains, because the control over the whole system is less time-critical than the components involved in sampling, allowing optimal placement on the FPGA of the latter. Both domains interface with each other through a clock crossing bridge as depicted in Figure 2.

3.6 Tuning the Design

The sampling accuracy of our design can obviously be improved by using faster FPGAs or by using bigger ones on which the additional resources increase the freedom of the fitter, which will hence be able to reach higher clock speeds.

The accuracy can also be increased by introducing a third clock domain for the timestamping core, which runs at a higher clock rate than the SSRAM interface. In this configuration, the size of the FIFO buffer internal to the timestamping core will determine the maximum burst size. Larger buffers allow longer bursts, but also require larger FPGAs. Since our design uses 63 out of 66 memory blocks on our target device, there is barely any room left to experiment with these features. Using more memory blocks severely limits the freedom of the fitter, and hence results in a significant drop of the maximum sampling frequency.

Table 2. Resource utilization of the complete design on an EP3C25F324 FPGA

Logic Elements	RAM bits	9kb Memory Blocks	PLLs	Pins
16,670 (67.7%)	238,382 (39.2%)	63 (95.5%)	2 (50.0%)	130 (60.2%)

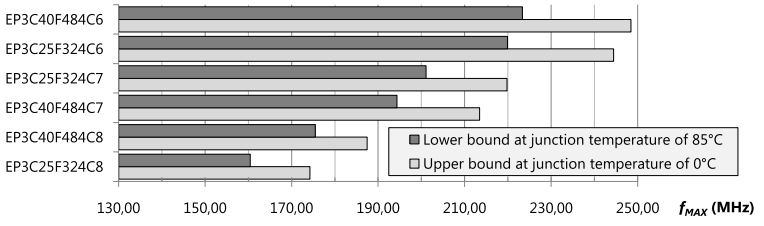


Fig. 5. Maximum sampling frequency obtained on several Cyclone III FPGAs

4 Performance Evaluation

All synthesis was performed with Altera Quartus II v9.0 [14], targeting the Altera Cyclone III FPGA starter kit. Table 2 shows the resource utilization of the complete design on the target FPGA. Figure 5 shows the maximum sampling frequency on different Cyclone III devices. For each device, the SDRAM interface was set to operate at the maximum frequency according to its speed grade.

To interpret these results, one should know that the EP3C40F484 devices are next in line to EP3C25F324 devices when it comes to available LEs, M9K blocks, and configurable IO pins; detailed specifications are shown in Table 1. The larger devices result in a speed gain for the fastest (C6) and slowest (C8) speed grades. However, the difference for the C6 grade is not significant, as results may slightly vary with pin assignments or different synthesis parameters.

We can draw the following conclusions: Except for the C8 speed grade, the resource utilization of the device on the FPGA does not harm its performance. For sampling frequencies up to 150 MHz, the smallest, cheapest and slowest FPGA (EP3C25F324C8) is sufficient. For sampling frequencies up to 200 MHz, the C6 speed grade of the same size (EP3C25F324C6) is a safe bet.

5 Conclusions

This paper presented the design of an FPGA-based RT event sampler that can be used to test the RT behavior of embedded systems. It supports fast, non-intrusive sampling, and is cheap because low-cost FPGAs suffice. Its performance scales very well with different FGPA classes and speed grades. Furthermore, by changing some design parameters, such as buffer sizes, a wide range of RT behaviors can be targeted easily, e.g., with or without long bursts of events.

References

1. Sha, L., Abdelzaher, T., Årzén, K.E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real time scheduling theory: A historical perspective. Real-Time Syst. 28(2-3), 101–155 (2004)
2. Buttazzo, G.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2nd edn. Real-Time Systems Series. Springer, Heidelberg (2005)
3. Choudhuri, S., Givargis, T.: FlashBox: a system for logging non-deterministic events in deployed embedded systems. In: SAC 2009: Proceedings of the 2009 ACM symposium on Applied Computing, pp. 1676–1682 (2009)
4. Chen, X., Zhang, D., Yang, H.: Design and implementation of a single-chip ARM-based USB interface JTAG emulator. In: IEEE International Symposium on Embedded Computing, pp. 272–275 (2008)
5. Active Technologies: AT-LA500 USB logic analyzer (2008), `http://www.activetechnologies.it/02products/Atla/000verview/text.htm`
6. CWAV Corporation: USBee DX Test Pod Users Manual. 3.1 edn. (2008), `http://www.usbee.com/dxmanual.pdf`
7. Intronix Test Instruments Corporation: Intronix LA1034 LogicPort PC-based logic analyzer with USB interface (2008), `http://www.pctestinstruments.com`
8. Janatek Electronic Designs: Annie-USB PC-Based Logic Analyzer: User's Manual, 2nd edn. (2008), s `http://www.janatek.co.za/annie-usb_main.htm`
9. Janatek Electronic Designs: LA-Gold-36 PC-based logic analyzer (2008), `http://www.janatek.co.za/la-gold-36_main.htm`
10. Link Instruments Corporation: IO-3200 USB logic analyzer and pattern generator for windows (2008), `http://www.linkinstruments.com/logana32.htm`
11. Penneman, N.: A renewed sampler system for evaluating and benchmarking (RT)OS. Master's thesis, Vrije Universiteit Brussel (2009)
12. Sutton, P., Brennan, J., Partis, A., Peddersen, J.: VHDL IP stack (2001), `http://www.itee.uq.edu.au/~peters/xsvboard/stack/stack.htm`
13. ADESCOM: Wire-Speed Internet: IP Core for VoIP and IPTV Internet (2009), `http://www.adescom.com/ipac1.htm`
14. Altera Corporation: Quartus II Handbook. 9.0.0 edn. (2009), `http://www.altera.com/literature/hb/qts/quartusii_handbook.pdf`
15. Altera Corporation: Triple Speed Ethernet MegaCore Function User Guide. 9.0 edn. (2009), `http://www.altera.com/literature/ug/ug_ethernet.pdf`
16. Altera Corporation: Nios II Processor Reference Handbook. 9.0.0 edn. (2009), `http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf`
17. Altera Corporation: Using high-performance DDR, DDR2, and DDR3 SDRAM with SOPC Builder. Application Note 517, Altera Corporation (2008), `http://www.altera.com/literature/an/an517.pdf`
18. Labrosse, J.J.: MicroC/OS-II: The Real Time Kernel, 2nd edn. CMP Media, Inc., USA (2002)
19. Arcturus Networks Incorporated: uClinux™– embedded Linux microcontroller project, `http://www.uclinux.org`
20. eCos: embedded configurable operating system, `http://ecos.sourceware.org`
21. Nios Community Forum: `http://www.niosforum.com`

A Performance Evaluation of CUBE: One-Dimensional 512 FPGA Cluster

Masato Yoshimi[1], Yuri Nishikawa[2], Mitsunori Miki[1], Tomoyuki Hiroyasu[1], Hideharu Amano[2], and Oskar Mencer[3]

[1] Faculty of Science and Engineering, Doshisha University, Japan
[2] Graduate School of Science and Technology, Keio University, Japan
[3] Department of Computing, Imperial College London, UK
myoshimi@mail.doshisha.ac.jp

Abstract. This paper reports an evaluation of CUBE, which is a multi-FPGA system which can connect 512 FPGAs in a form of a simple one dimensional array. As the system well suits as stream-oriented application platforms, we evaluated its performance by implementing edit distance computation algorithm, which is a typical streaming algorithm. Performances are compared with Cell/B.E., NVIDIA's GeForce GTX280 and a general multi-core microprocessor. The report also analyzes pipeline utilization, and discusses performance efficiency, logic consumption and power efficiency with comparison to other multi-core devices.

1 Introduction

There is a growing demand of high-performance computing systems to cope with increasing data size and amount of arithmetic operations in various scientific computation. On top of this, several multi-FPGA systems have been proposed as reasonable methods to efficiently solve large-scale applications[1] which adopt extremely iterative algorithms, or use large-scale data that cannot be loaded onto a single FPGA[2][3][4]. This is the leading motive for launching a new project in 2008 to design a hardware platform called CUBE[5], a system that can integrate a maximum of 512 FPGAs. CUBE is an extremely simple system; 64 identical FPGAs on one board are connected in a form of one-dimensional array topology, and its diameter can be extended by connecting multiple boards. These FPGAs do not have shared memories nor external local memories, and all data are simply passed on to neighboring FPGAs with their 64-bit data path.

Despite its simplicity, CUBE is expected as high-performance platform for various applications with intense computation by exploiting its systolic architecture. One suitable example is stream-oriented applications. In this work, we evaluate performance for computing Levenshtein distance, which is a typical algorithm for stream-oriented processing. Levenshtein distance is also called an edit distance of two character strings, which is a number of operations for modifying one character string into the other. This is a computational-bound algorithm with various exploitable parallelism, and is often applied for performance evaluation of computation systems. The performance of CUBE is studied in detail by comparing it with multi-threaded execution of software on Cell/B.E., NVIDIA's GeForce GTX280 and Core 2 Quad.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 372–381, 2010.

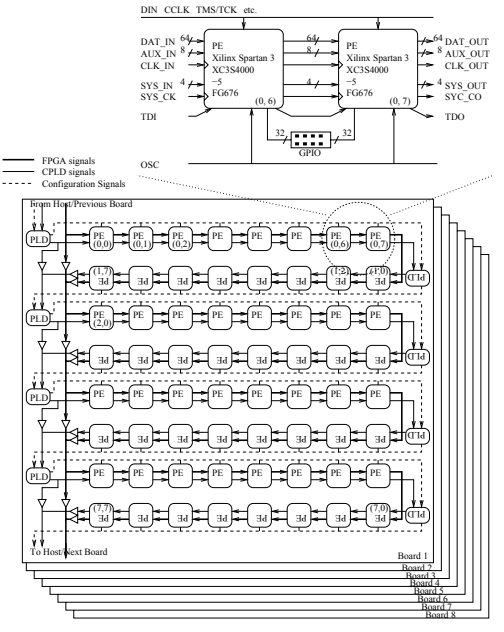


Fig. 1. Architecture of CUBE

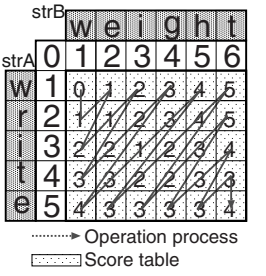


Fig. 2. An example of operation sequence

Table 1. Environment of C-LD program code

CPU	Intel Core 2 Quad Q6600 @ 2.4GHz
RAM	4.0 GByte
OS	GNU/Linux 2.6.23 X86_64
Compiler	gcc-4.1.2(-O3 -lpthread)

2 CUBE: A 512-FPGA Cluster

2.1 Architecture Overview

CUBE is a massively-parallel processing platform that consists of a large number of FPGA chips. Fig.1 is a block diagram showing the basic idea of CUBE. The system is made from multiple boards each containing a large number of FPGAs (64 FPGAs in the figure). In the current design, each board contains 64 Xilinx FPGAs (XC3S4000-5-FG676), and eight boards can be connected at maximum to integrate a total of 512 FPGA devices.

Each FPGA on the board is regarded as "PEs", and they are connected in a form of one dimensional array. Each of them accepts data from the previous, and passes to the next PE. Like this, data are processed in a systolic manner. Global buses are not adopted to the architecture in order to suppress wiring delay and to achieve higher scalability.

Due to systolic arrangement of FPGAs, application with intensive stream computation is preferable. Five types of applications have already been implemented on CUBE[5]: 64 FPGA devices on a single board of CUBE could search 2^{40} key space in 1460 seconds with a power dissipation of 104W. Similar computing performance can be obtained with a cluster of 359 Intel Quad-Core Xeon 2.5GHz processor, but with approximately 690 times larger power consumption.

3 Levenshtein Distance

Levenshtein distance (LD), also called "edit distance", is a score representing a degree of similarity between two character sequences.

The score of edit distance is obtained by counting a number of operations required to convert an original string (*strA*) to another string (*strB*). To do so, three types of basic operations are defined:(1) substitution, (2) insertion, and (3) deletion. The calculation proceeds obtaining values in element of the score table c. The size of the score table is $(lenA + 1) \times (lenB + 1)$, where *lenA* and *lenB* are numbers of characters in *strA* and *strB*, respectively.

All $c(i, 0)$ and $c(0, j)$ (The first row and first column) store initial values of i and j, respectively. Each $c(i, j)$ is calculated as following two steps; (1) Temporary variable a is stored "1" when $(i - 1)$th character $strA_{i-1}$ in $strA$ equals to $(j - 1)$th character $strB_{j-1}$ in $strB$. If $strA_{i-1}$ is not equivalent with $strB_{j-1}$, a becomes "0". (2) $c(i, j)$ is set as the minimum value among $c(i-1, j)+1$, $c(i, j-1)+1$ and $c(i-1, j-1)+a$. Finally, a score between *strA* and *strB* is stored into a element $c(lenA, lenB)$. Fig.2 shows an example of operating sequence for calculating the score between "write" and "weight".

4 Parallel Execution of LD Algorithm

This section describes the implementation of LD algorithm in C language. The program was implemented to make its parallelism clear before FPGA implementation. It is also used for comparing performance with multi-core processors and other accelerators whose results are shown later. First, we discuss the inherent parallelism of LD algorithm and then introduce problem reduction technique for multi-thread execution.

4.1 Parallelism of LD Algorithm

Calculation of the score table can be divided into multiple blocked processes as shown in Fig.3. It shows the procedure to divide two sets of 24 strings into 36 sub-blocked problems with six sets of 4 strings.

It is possible to compute Block B(0,1) and B(1,0) in parallel after completion of B(0,0). In a similar way, blocks on dashed-line T_n can be calculated simultaneously after computation of blocks on T_{n-1}. Inputs of the calculation $B(i, j)$ are: sub-sequences of *strA* and *strB*, intermediate data $p(i, j - 1)$ and $q(i - 1, j)$ of a row and a column, and top-left most data $a(i - 1, j - 1)$ from neighboring blocks. $B(i, j)$ transfers $p(i, j)$, $q(i, j)$ and $a(i, j)$ to its neighboring blocks. The number of simultaneously-computable blocks increases when the length of an input sequence is large.

4.2 Implementation of Multi-thread Execution

The multi-thread program calculates the score among sequences using blocked algorithm, which is supported by multi-thread execution using pthread library.

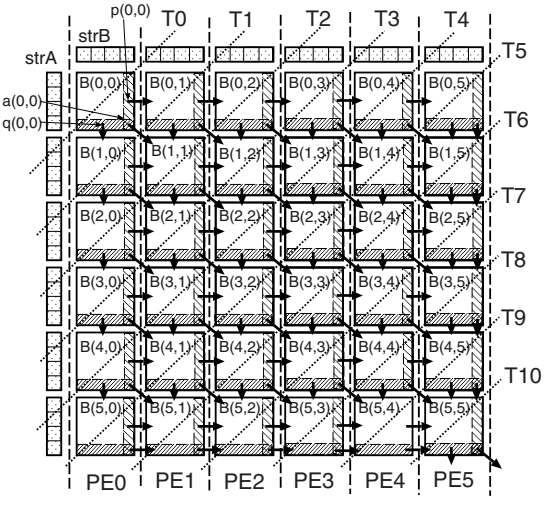


Fig. 3. Multi-thread algorithm with data-flow

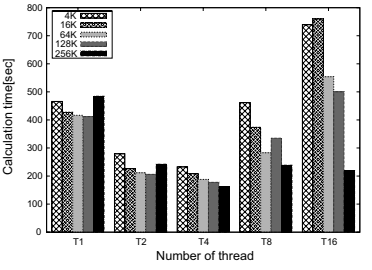


Fig. 4. Calculation time versus number of thread with various block size

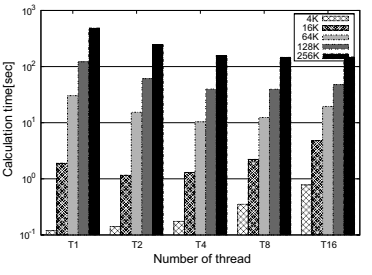


Fig. 5. Calculation time versus sequence length with various number of thread

The program introduces two parametrized features to evaluate the effectivity of multi-thread operations. First is the block size for dividing the score table. For maximizing parallel efficiency, a block must consist of a character, while overhead of thread synchronization increases according to granularity of a block. Second is the number of threads which run simultaneously. It can be specified independently from the number of core in a microprocessor. Throughput increases when multiple threads is executed on a single core. Both parameters can be specified as command-line options.

4.3 Performance of a Recent Multi-processor

To investigate the performance of recent multiprocessor, detailed analysis is necessary using various parameter configurations. Number of calculating Levenshtein distance were evaluated with a recent multi-processor as shown in Table.1. The implemented program code was run with configurations set from two viewpoints, block size and a number of thread.

First, we examined the number of concurrently-executable threads by fixing the length of character sequences and run the program with various block sizes. Fig.4 shows time required for computing character sequences which consist of 256×1024 characters with seven types of block sizes. It was improved according to the increase of block

size and the number of thread. This performance degradation in small block size with large number of thread may be caused by overhead of synchronizations among many blocks. Performances using four threads were well balanced when the number of thread was equal to the number of physical cores in Core 2 Quad processor.

Second, we studied the relationship between length of character sequence and computation time with fixed block size and variable number of threads. The result is shown in Fig.5. Note that y-axis is a logarithmic axis. Computation time increases according to the increase of character length following the computational order of Levenshtein distance $O(N^2)$. Fig.5 also shows that the performance is optimal when the number of thread is equal to the number of core, except for a case when the length of a character sequence is 4096. In this special case, the best performance is obtained using only one thread, because the length of character sequence is equal to the block size.

5 Implementation on CUBE

For hardware implementation on CUBE, calculation- and data-flow among FPGA chips must be discussed first. As CUBE is a large one-dimensional FPGA array, data-flow must have one-way traffic throughout the first FPGA chip to the last one. In addition, for achieving pipeline efficiency, it is essential to consider both on-chip and off-chip data flow. As a set of data is transferred between adjacent two FPGAs in one-way traffic, CUBE can be regarded as a large systolic array. Therefore, calculation of LD using CUBE is divided into two stages: data-transfer stage and computation stage.

5.1 Data-Flow between FPGAs

The blocked algorithm is also considered as a reasonable and proper way to divide operation for parallel execution using FPGAs. Through operation process of LD in Fig.2, an appropriate approach is for each FPGA to take a vertical block array, as shown in Fig.3. Fig.3 shows an example of off-chip data-flow, when six FPGA chips consist the system.

An FPGA "PE0" in Fig.3 holds the first sub-sequence of $strB_0$ and compares with the sub-sequences from $strA_0$ to $strA_5$ per calculation stage. PE1 and following PEs calculate the i-th column of the score table in a similar way.

Fig.6 shows a computational scheme of LD with four blocks on four FPGAs connected in series. Before starting computation, sub-sequences $strB_j$ are distributed to each PEs. The size of sub-sequence is selected appropriately in order to be calculated in a PE. If the number of sub-sequences is larger than the number of PEs, the calculation is repeated. In case when the number of sub-sequence is less than the number of PEs, computation would be done using FPGAs with small ID numbers. In this case, the remaining FPGAs are not involved in the computation, and its results would be bypassed throughout the systolic chain.

At a transfer stage, sub-sequences $strA_i$ are thrown into the input signals of CUBE. At a calculation stage, all PEs compute a block algorithm using input $strA_i$ and p_i from the previous PE, then transfer $strA_i$ and $p_{(i,j)}$ to the next PE. All PEs transfer calculated results $a_{(i,j)}$, p_i and sub-sequence $strA_i$. Calculated result $q_{(i,j)}$ and sub-sequence $strB_j$ are reused in the PE.

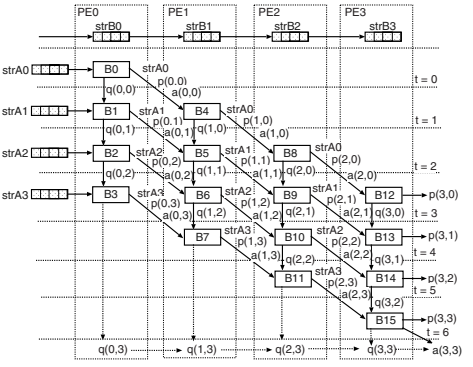


Fig. 6. Expecting data-flow on CUBE

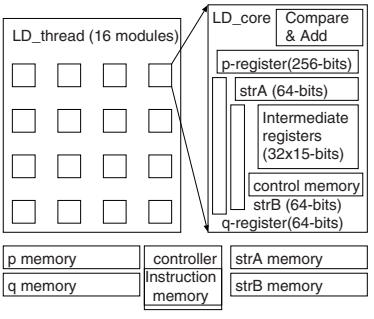


Fig. 7. LD architecture in a FPGA

At the end of computation, the tail PE outputs the score $a_{(3,3)}$ with other intermediate data $p_{(3,j)}$ and $a_{(3,3)}$ which are used in the next round for computation of a large sequence. In this case, intermediate data $q_{(i,3)}$ in each PE have to be gathered to PE3 for reuse after computation.

5.2 Calculation Inside FPGA

Each FPGA performs a block calculation. The core computation is consisting of an addition and comparison for three data (See Section 3). The block calculation is composed of circulation of the core computation for two sub-sequences and intermediate score data.

An implemented architecture for LD is shown in Fig.7. The architecture consists of two layers: LD_thread module and LD_core module. The microcode-based architecture is adopted for extending or updating FPGA. Both modules have a controller with memory to select calculation data. These memories are implemented by BlockRAM of Xilinx's FPGA.

An LD_core calculates a score of 8×8 strings with a computational core which has three sets of a comparator and an adder. The input data is selected by control memory every clock cycle. As all modules are recommended to drive over 100MHz from hardware constraint. LD_core module forms four-stage pipeline consisting of reading computation data from registers, adding scores, comparing them, and writing back the score to intermediate registers and p or q registers for output transfer.

The LD_core completes calculation for 8×8 strings for 80 clock cycles. The computation ratio, which is the rate of productive operation by the whole clock cycles, is $64/80 = 0.8$.

The LD_thread has 16 LD_cores for effective use of FPGA. The controller in LD_thread issues instruction of 8×8 strings calculation to each LD_core in series. The controller issues instructions every 83 clock cycles each of which includes reading and writing to p and q memories in the LD_thread. As the number of LD_core increases and decreases according to advance of computation, the net of computation ratio in an LD_core becomes 0.398.

Table 2. Resource utilization and operating frequency

	LD_thread	LD_unit	XC3S4000
Slices	22483	1340	27648
FFs	17985	1063	55296
LUTs	42369	2496	55296
BRAMs	10	1	96
Freq.	125.594	156.966	-

Table 3. Power requirement for each systems

Vendor	Device	Power (W)
Intel	Core2Quad Q6600	105
Sony	ZEGO(BCU-100) Cell/B.E.	330
NVIDIA	GeForce GTX280	236
Imperial	CUBE (8 boards)	832

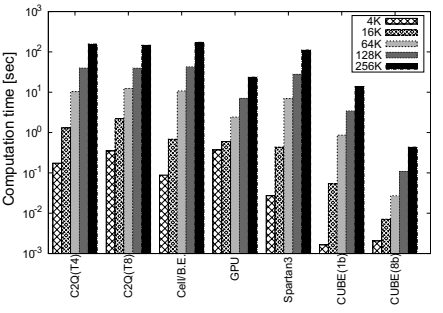


Fig. 8. Calculation time for C2Q, Cell/B.E. and GeForce GTX280 with CUBE

6 Evaluation

6.1 Hardware Resources

All modules were written in Verilog-HDL, and synthesis, placement and routing were done by Xilinx's ISE10.1i. Target device of the design is Xilinx's Spartan-3 (XC3S4000-5-FG676) on CUBE. Any instruction memories in LD modules and p- and q- memories in LD_thread were utilized from Xilinx's LogiCORE BlockRAM.

Table 2 shows estimations of logic resources and operating frequencies of each module. Constraints of the design are to integrate all LD_thread modules and data-transfer FIFOs into a single FPGA, and also to maintain the internal operating frequency higher than 100MHz. The architecture of this implementation can be easily extended for other FPGAs with larger logic resources.

6.2 Computation Speed

The performance of each FPGA in CUBE is estimated assuming 100MHz of operating frequency, and data transfer rate of 64 bits per clock cycle. Each FPGA takes 2573 clock cycles to calculate score of 128 strings, and 82 clock cycles to transfer 644 Bytes. The transferring data has a header flit at the head of a packet. Above values are introduced by following equations:

$$[\text{calculation time}] = 83 \times (1 + 2 + \cdots + 16 + 15 + \cdots 1) = 2573[\text{clock cycles}] \quad (1)$$

$$\begin{aligned}[\text{transfer time}] &= (strA_i + p_i + a_{ij})/64 = (1 \times 128 + 4 \times 128 + 4)[\text{Bytes}]/64 \\ &= 80.5[\text{clock cycles}] \end{aligned} \quad (2)$$

Since sending and receiving operations are overlapped, the total time by calculation and transfer stages is $2573 + 82 = 2655$ clock cycles. When the number of block n in both sequences are less than 512, number of block calculation is $2n - 1$ times. When n is larger than 512, the computation time is simply the multiple of computation time

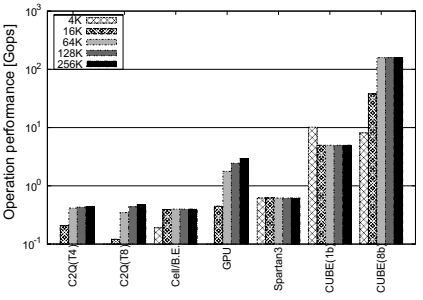


Fig. 9. Operation performances for LD

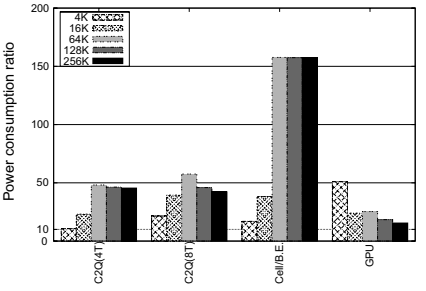


Fig. 10. Power consumption ratio compared to CUBE

in case of $n = 512$. If sequence length is 128K characters ($n = 1024$), calculation of $n = 512$ is repeated for four times.

6.3 Competitions of Computing Environment

To analyze the computing performance for LD, four computing environments were used to measure their performance: (1) three variations of CUBE: a single Spartan-3, one board of CUBE, and full system of CUBE with eight boards, (2) Core 2 Quad (discussed in Section 4), (3) Cell/B.E. which is equipped on ZEGO(BCU-100)[6], and (4) NVIDIA's GeForce GTX280 as a GPU.

Performances of these platforms are evaluated by three standpoints, computation time, operating performance and power efficiency. The results of them are shown in Fig.8, Fig.9 and Fig.10, respectively. Fig.8 shows the calculation time to obtain the score of various length of LD.

As the LD algorithm described in Section 3 is accumulation of score calculation to fill a score table, performances are also obtained by the number of operation to get a score in each operation. Fig.9 shows the operation throughput in one second. Power consumptions are shown in Fig.10, where the results are normalized to the consuming power of CUBE.

7 Discussion

7.1 Performance Improvement for CUBE

As the result of these evaluations, CUBE shows the best performance compared to other devices even though naive blocked algorithm was adopted. For achieving better performance, the most important factor is to consider the density of calculation. According to Section 5.2, the computation ratio in LD_thread is 0.398. The simplest method to achieve higher computation ratio is to modify the "granularity" of computation.

In the current implementation, the base unit of computation is a "string" with 8 characters to be calculated by a single LD_core module, and 16 strings comprise a single

"block". Thus, 128 characters are calculated within a single block. In this case, the computation ratio r in LD_thread is:

$$r = 0.398 \times (1 + 2 + \cdots + 16 + 15 + \cdots + 1)/16/(16 + 16 - 1)) = 0.205 \quad (3)$$

According to this computational ratio, logic resources in CUBE is not effectively used. Due to the nature of blocking algorithm, there is only one stage out of 31 stages that all 16 LD_core modules are being used in case of a single block with 128 characters.

To improve the computational ratio r, the number of characters in a single block should be increased. Assuming that 256 characters are in a single string (or LD_thread), the computational ratio becomes:

$$\begin{aligned} r = {} & 0.398 \times (1 + 2 + \cdots + 16 \times 16 + 15 + \cdots + 1)/16 \times 2 \\ & /((15 + 16 + 15) \times 2) = 0.268 \end{aligned} \quad (4)$$

In this case, all 16 LD_thread modules become active for multiple stages, which enhance the operational ratio of computational units.

7.2 Performance Comparison

From the results of GPU and CUBE in Fig.9, operation throughput increases quadratically while the system has resource capacity. However, when the size of sequences reaches the specific length, operation throughput are suppressed the definite speed.

Power consumptions were estimated as shown in Fig.10 normalized to the power of CUBE. This was obtained based on Fig.9 and data-sheet values of each devices shown in Table 3. Fig.10 shows that CUBE can provide a good power efficiency. These quantitative results suggest that CUBE is a power efficient system.

8 Conclusion and Future Work

This paper reported and discussed implementation and evaluation of Levenshtein distance (LD) algorithm for CUBE, which is a computation system using maximum of 512 FPGA devices. By exploiting data- and loop- level parallelism in LD, CUBE shows the best performance compared to other implementations of x86 multiprocessor, Cell/B.E. and GeForce GTX280.

As a future work, bit-parallel and pruning techniques will be implemented on CUBE. Also, further performance analysis such as relationships between number of blocks and computational performance will be conducted in order to find the number of threads that extracts the highest potential performance of CUBE.

Acknowledgments

This work is supported by VLSI Design and Education Center(VDEC), The University of Tokyo with the collaboration with Cadence Corporation.

References

1. Arnold, J.M., Buell, D.A., Davis, E.G.: Splash 2. In: SPAA 1992: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, pp. 316–322. ACM, New York (1992)
2. Hamada, T., Fukushige, T., Kawai, A., Makino, J.: PROGRAPE-1: A Programmable, Multi-Purpose Computer for Many-Body Simulations. Publications of the Astronomical Society of Japan 52, 943–954 (2000)
3. Chang, C., Wawrzynek, J., Brodersen, R.: BEE2: A High-End Reconfigurable Computing System. IEEE Design & Test of Computers 22(2), 114–125 (2005)
4. Burke, D., Wawrzynek, J., Asanovic, K., Krasnov, A., Schultz, A., Gibeling, G., Droz, P.: RAMP Blue: Implementation of a Multicore 1008 Processor FPGA System. In: Proceedings of the Fource Annual Reconfigurable Systems Summer Institute (RSSI 2008) (July 2008)
5. Mencer, O., Tsoi, K.H., Craimer, S., Todman, T., Luk, W., Wong, M.Y., Leong, P.H.W.: CUBE: A 512-FPGA CLUSTER. In: Proc. IEEE Southern Programmable Logic Conference (SPL 2009) (April 2009)
6. SONY. BCU-100 Computing Unit with Cell/B.E. and RSX, `http://pro.sony.com/bbsccms/ext/ZEGO/files/BCU-100_Whitepaper.pdf`

An Analysis of Delay Based PUF Implementations on FPGA

Sergey Morozov, Abhranil Maiti, and Patrick Schaumont

Virginia Tech, Blacksburg, VA 24061, USA
{morozovs,abhranil,schaum}@vt.edu

Abstract. Physical Unclonable Functions promise cheap, efficient, and secure identification and authentication of devices. In FPGA devices, PUFs may be instantiated directly from FPGA fabric components in order to exploit the propagation delay differences of signals caused by manufacturing process variations. Multiple delay based PUF architectures have been proposed. However, we have observed inconsistent results among them. Ring Oscillator PUF works fine, while other delay based PUFs show a significantly lower quality. Rather than proposing complex system level solutions, we focus on the fundamental building blocks of the PUF. In our effort to compare the various delay based PUF architectures, we have closely examined how each architecture maps into the FPGA fabric. Our conclusions are that arbiter and butterfly PUF architectures are ill suited for FPGAs, because delay skew due to routing asymmetry is over 10 times higher than the random variation due to manufacturing process.

1 Introduction

A Physical Unclonable Function (PUF) has the unique advantage of generating volatile chip-specific signatures at runtime. It not only excludes the need of an expensive non-volatile memory for key storage, but also offers robust security shield against attacks. It is emerging as a promising solution to issues like intellectual property (IP) protection, device authentication, and user data privacy by making device specific signatures possible.

The majority of the PUF designs are based on delay variation of logic and interconnect. The fundamental principle followed in these delay-based PUF is to compare a pair of structurally identical/symmetric circuit elements (composed of logic and interconnect), and measure any delay mismatch that is introduced by the manufacturing process variation, and not by the design.

We will show that Arbiter PUF and Butterfly PUF are inherently difficult to implement on FPGA due to the delay skew present between a pair of circuit elements that are required to be symmetric in these PUFs. This static skew is an order of magnitude higher than the delay variation due to random process variation. Our main contribution in this paper is to present the complexities in implementing two PUFs on a 90nm commodity FPGA platform.A more detailed technical report discussing the root causes of these complexities and details of our implementations is available in [6].

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 382–387, 2010.

2 Background

A PUF is a function that generates a set of responses while stimulated by a set of challenges. It is a physical function because the challenge-response relation is defined by complex properties of a physical material, such as the manufacturing variability of CMOS devices. Its unclonability is attributed to the fact that these properties cannot be controllably reproduced, making each device effectively unique. Though many PUF architectures have been proposed, we focus on the categories of PUF based on the delay variation of logics and interconnects, specifically the arbiter PUF (APUF) and the Butterfly PUF (BPUF). In the technical report [6], these architectures are also compared with the ring oscillator PUF architecture [5].

Arbiter PUF - An APUF, proposed by Lim et.al [2], is composed of two identically configured delay paths that are stimulated by an activating signal(Fig. 1(a)). The difference in the propagation delay of the signal in the two delay paths is measured by an edge triggered flip-flop known as the arbiter. Several PUF response bits can be generated by configuring the delay paths in multiple ways using the challenge inputs.

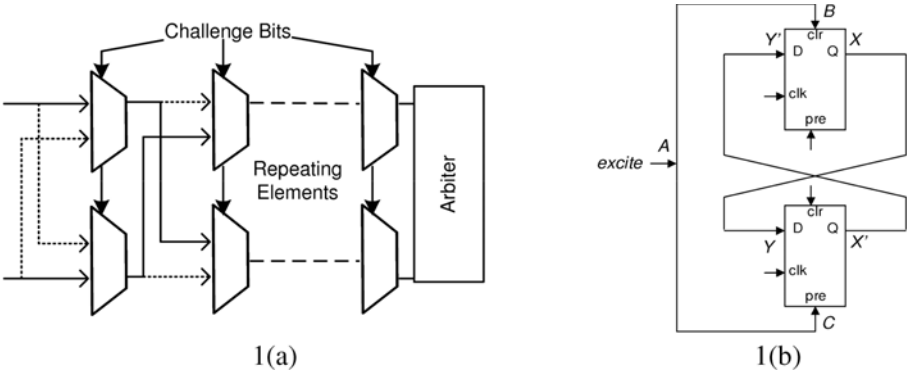


Fig. 1. (a). The Arbiter PUF structure. The symmetric pairs of components are highlighted with matching patterns. (b) A BPUF cell with two cross coupled latches.

Butterfly PUF - The Butterfly PUF, proposed by Kumar et.al [3], is a technique that aims to emulate the behavior of an SRAM PUF [1]. However, the functionality of this PUF is based on the delay variations of interconnects. A BPUF cell employs two cross-coupled latches, and exploits the random assignment of a stable state from an unstable state that is forcefully imposed by holding one latch in preset while the other in clear mode by an excite signal (Figure 1(b)). The final state is determined by the random delay mismatch in the pair of feedback paths and the excite signal paths due to process variation.

The equation for delay d of a net N in a circuit is shown in Equation 1, where d_S is the static delay as determined by the static timing analysis tools, and d_R is the random delay component due to process variation.

$$d_N = d_S + d_R \tag{1}$$

The delay difference between two nets, N_1 and N_2 , in a circuit maybe be expressed as a sum of static delay difference Δd_S and random delay difference Δd_R [6] as shown in Equation 2.

$$\Delta d = d_{S1} - d_{S2} + d_{R1} - d_{R2} = \Delta d_S + \Delta d_R \tag{2}$$

A delay-based PUF circuit involves extraction and comparison of the random delay, d_R while minimizing Δd_S. In the ideal case for a delay based PUF, $\Delta d_S \rightarrow 0$ and the delay skew is purely a function of the random delay component. However, typically the output of a given PUF structure will be at least partially dependent on Δd_S, causing the output to be biased. Further, if $\Delta d_S > \Delta d_R$, the effect of random variation becomes insignificant, and the output of the PUF structure becomes static regardless of d_R. The effectiveness of the PUF depends on how much symmetry we can achieve between a particular pair of elements in order to minimize the effect of Δd_S. This symmetry requirement determines the implementation complexity of a PUF on FPGA.

In Figure 1(a), the basic structure of APUF, is shown. The pairs of nets connected to the multiplexers (pairs shown with different patterns) need to be symmetric in order to minimize Δd_S. In figure 1(b), a BPUF cell is presented. For a functional BPUF, the pair of nets AB/AC as well as XY/X'Y' need to be symmetric along with the latches.

3 PUF Architecture in FPGA

The Xilinx Spartan3E FPGA device that we used as a platform in our experiments is made up of Configurable Logic Blocks (CLBs) surrounded by a sea of interconnect. Inside the CLB, there are 4 slices which contain configurable look-up tables and flipflops. In this section, we examine the mapping of PUF elements into this structure.

3.1 Arbiter PUF

The two primary components required for the Arbiter PUF architecture are the switches and the arbiter itself. A useful secondary component is a delay element: trivial logic that inserts additional delay into the paths. We used identical 2-input MUXes in two different slices for the switches. The arbiter was instantiated as a positive clock-edge triggered flipflop in a slice. A look-up table in a slice served as the delay element. We examined several different mapping schemes for these components [6].

Using timing analysis tools, we observed the routing delay caused by each component. This value does not account for the manufacturing variability. Therefore, we hoped to find identical static delays for symmetric routes. Figure 2(a) shows the delay caused by each component for a possible route. There are two values for the switch component: Switch Nominal is the delay of the signals when the

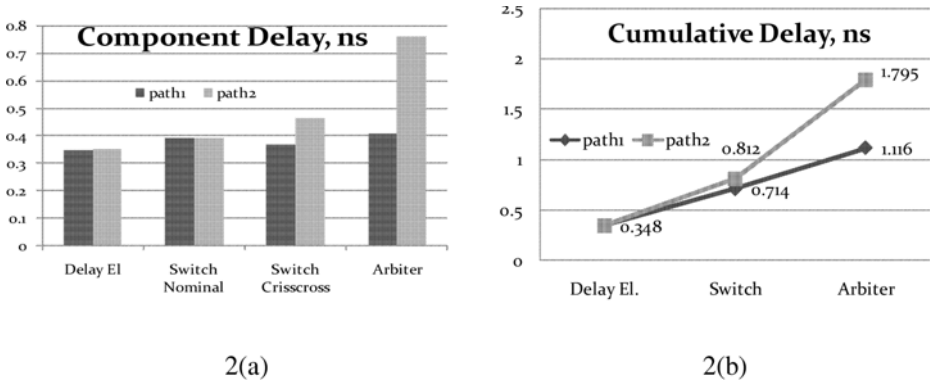


Fig. 2. (a) Individual delays of each component. (b). Cumulative delay of the path through after each component.

paths are straight, Switch Crisscross is the delay of the signal when the paths are crossed. Figure 2(b) shows the cumulative delay of the signal propagated along the route when the switches are crossed.

These results indicate that this PUF structure will not function. The 3σ value of delay variation due to process variability in 90 nm technology has been estimated to be approximately 3.5% [4]. On the other hand, we observe that in this case, variation due to routing is much higher than expected variation due to process variability: d_S/d_R is 25.6 times.

There are two causes of routing variation in this design: the asymmetric routes for the switch crisscrossing paths and the much more dramatic asymmetric routes to the arbiter. The larger difference in arbiter routes is due to the fact that routing to a CLK input of a flipflop requires sending the signal through multiple additional segments to reach the CLK port, whereas the route to the D input of the flipflop is comparatively simple.

All attempted mapping schemes produced similar results. While some directions significantly reduced the routing delay, a difference of at least 100 ps remained. Figure 3 shows the delays we observed for the arbiter component in each direction, depending on the location of the arbiter, and the mapping scheme. Under the best possible conditions of 2 CLB mapping and West to East placement, we observe $\Delta d_S/\Delta d_R$ to be 11.6 times.These results indicate that additional delay due to the complexity of routing a signal to the CLK input of a flipflop cannot be avoided using current routing schemes and architectures. Asymmetry in routing of crisscrossing switch routes is also present, but that delay difference is dwarfed by the arbiter.

3.2 Butterfly PUF

Though the pair of latches can be safely assumed to be identical in a Butterfly PUF, the real design challenge comes when a designer has to ensure that the

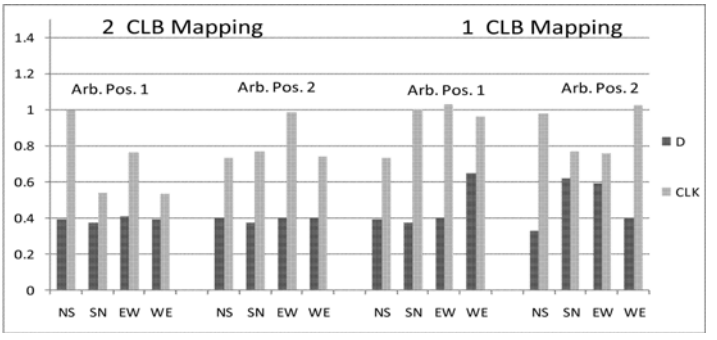
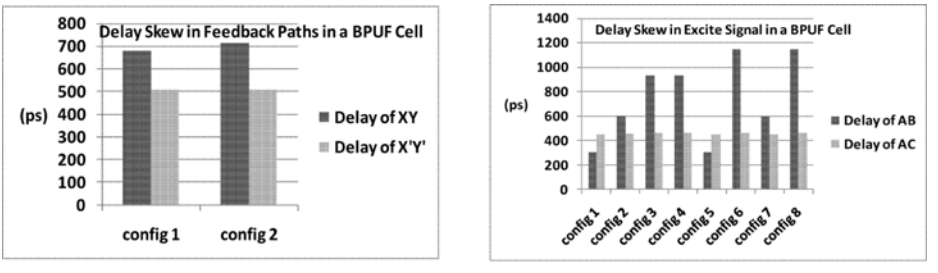


Fig. 3. Delay difference in routing to the D input and the CLK input of a slice under various conditions. NS - North to South layout; SN - South to North; EW - East to Wes; WE - West to East.



(a) (b)

Fig. 4. (a) The delay values of nets XY and X'Y'. Two configurations correspond to two possible placements of the pair of latches inside a CLB. (b) The delay values of nets AB and AC. Eight configurations correspond to four possible placements of the excite signal buffer in a CLB for each of the two possible placements of the latches in the CLB.

interconnection pairs (XX'/YY' and AB/AC) are symmetric. Since no layout level information inside the switch box is available, we depend on the static delay values provided by the design tool. In Figure 4, we present a set of data showing the delay skew in the pair of nets AB/AC and XY/X'Y' (refer to figure 2(b)) as a result of automatic routing with timing constraint.

For the delay skew in XY/X'Y' net pair, the minimum value of the ratio $\Delta d_{\mathrm{S}}/\Delta d_{\mathrm{R}}$ is estimated to be 17 whereas the same quantity for the delay skew in AB/AC is 16. Since Δd_{S} is an order of magnitude higher than Δd_{R}, it is obvious that this PUF implementation will produce highly biased outputs. Even with the manual routing, it has been observed that the delay of a pair of interconnects do not match based on the static delay value. From the results it is evident that BPUF cell suffers from the asymmetric nature of the FPGA routing resources.

Thus, our observations contradict the results presented in [3]. However, we note that our experiments have been done on a Spartan-3E FPGA, and not on a Virtex-5 as used in [3].

4 Conclusion

In this work, we have analyzed how the peculiarities of FPGA routing affect the implementations of delay based PUFs. Our results show that symmetry requirements for Arbiter and Butterfly PUF architectures cannot be satisfied using available FPGA routing schemes, despite the apparent routing flexibility of FPGA devices. Using the best possible routing, the delay difference due to static variation routes is an order of magnitude higher than expected delay variation due to manufacturing variability. Yet an architecture without the mirror symmetry requirement, such a Ring Oscillator based PUF, can produce a working PUF. Ultimately, understanding how a particular PUF architecture maps into FPGA fabric allows us to select a promising architecture for further investigation and characterization of PUF circuits in FPGAs.

Acknowledgement

This work was supported in part by NSF grant no 0855095.

References

1. Guajardo, J., Kumar, S.S., Schrijen, G.-J., Tuyls, P.: FPGA intrinsic PUFs and their use for IP protection. In: Cryptographic Hardware and Embedded Systems (2007)
2. Lim, D., Lee, J.W., Gassend, B., Suh, G.E., Van Dijk, M., Devadas, S.: Extracting secret keys from integrated circuits. IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2005)
3. Kumar, S.S., Guajardo, J., Maes, R., Schrijen, G.J., Tuyls, P.: The Butterfly PUF: Protecting IP on every FPGA. In: IEEE International Workshop on Hardware-Oriented Security and Trust, HOST (2008)
4. Sedcole, P., Cheung, P.Y.K.: Within-die delay variability in 90nm FPGAs and beyond. In: Proceedings of IEEE International Conference on Field Programmable Technology (2006)
5. Suh, G.E., Devadas, S.: Physical Unclonable Functions for Device Authentication and Secret Key Generation. In: Proceedings of Design Automation Conference (2007)
6. Morozov, S., Maiti, A., Schaumont, P.: A Comparative Analysis of Delay Based PUF Implementations on FPGA. IACR ePrint tbd/2009 (submitted December 19, 2009)

Comparison of Bit Serial Computation with Bit Parallel Computation for Reconfigurable Processor

Kazuya Tanigawa, Ken'ichi Umeda, and Tetsuo Hironaka

Hiroshima City University, Graduate School of Information Sciences,
Ozuka-higashi 3–4–1, Asaminami-ku, Hiroshima, 731-3194 Japan
reconf09@csys.ce.hiroshima-cu.ac.jp

Abstract. Most of reconfigurable processors are adopting bit-parallel computation. On executing a program on such a reconfigurable processor, since bit-parallel computation require more hardware than bit-serial one, programs are often divided into several configuration and executed sequentially, which cause large overhead on performance. To solve the problem, we have developed a reconfigurable processor based on bit-serial operation, which can execute more operations in a reconfigurable part enough to prevent such a division of configuration and keep the chip area small. This paper shows that a reconfigurable processor based on bit-serial computation achieves higher performance than the traditional one based on bit-parallel computation under the condition of same chip area, by the evaluation using median filter as benchmark program.

Keywords: Reconfigurable processor, Bit-serial computation, Performance evaluation, DS-HIE.

1 Introduction

For various types of applications many systems have been implemented with a reconfigurable processor as an accelerator [1][2]. In principle, a reconfigurable processor can achieve high performance as same as ASIC when enough operation units required in the target application is included in the reconfigurable processor. In fact, however, it is difficult to prepare enough operation units since the overhead of chip area or delay by the circuit to provide its reconfigurability is large. Therefore, we need to divide the target application into several parts then execute each part with dynamically reconfiguration. Such a division of target application, however, causes more overheads on chip area or delay since data transfers between divided parts are inserted. As the result, the performance gap between ASIC and reconfigurable processor becomes larger. From these point, it is important to decrease such an overhead caused by data transfer.

To prevent the overhead by data transfer, our objective is to prepare many operation units in a reconfigurable part while keeping chip area small. To achieve the objective, we have developed a reconfigurable processor DS-HIE [5][4] which

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 388–393, 2010.

adopts bit-serial computation method. By utilizing bit-serial computing, more operation units in the DS-HIE processor can be implemented than that of the well-known bit-parallel computing processor. This paper confirms that the DS-HIE processor achieves higher performance compared with that of bit-parallel computation by adopting bit-serial computation, under the condition with same chip area.

The organization of this paper is as follows. Section 2 summarizes the DS-HIE processor then Section 3 explains the detailed design of the DS-HIE processor. Section 4 compares the DS-HIE processor with a processor which adopts a bit-parallel computation in terms of gate counts. Section 5 shows the results of performance evaluation of the DS-HIE processor, the benchmark program is median filter. Finally, Section 6 concludes this paper.

2 Overview of DS-HIE Processor

The key features of the DS-HIE processor which achieves high performance in a reasonable chip size are as the following

- By adopting bit-serial computation, the DS-HIE processor can have more operation units that which adopts bit-parallel computation.
- Benes network adopted as routing resource can provide a rich network while keeping chip size small.
- Feedback path in a Operation Stage can provide further flexibility on routing.

In this paper, the page count is limited so please refer to the paper [4] [5] for its details.

Figure 1 shows the block diagram of the DS-HIE architecture, which consists of *a Data Supply Unit*, *an Input Buffer*, *an Output Buffer*, *a Context Buffer* and *a Reconfigurable Part*. A RISC processor plays the role of a control processor of the DS-HIE processor. Upon execution, the RISC processor transfers the data to *the Input Buffer* through *the Data Supply Unit*, and sends an execution signal. When the DS-HIE processor receives the execution signal, it starts processing the data in a streaming manner. While the DS-HIE processor is processing the data, the RISC processor prepares the next piece of data and receives the calculated data from *the Output Buffer*. Furthermore, *the Context Buffer* has two context entries, which represent configuration information for the DS-HIE processor. While one of the two entries is used for execution on the DS-HIE processor, the another can be used for prefetching the next Context.

3 Detailed Design of Bit-Serial FU

This section describes the detailed design of *the Bit-Serial Function Unit*(BS-FU) in the DS-HIE processor developed for this paper's evaluation. At first, we will look at timing adjustment method adopted in *the bit-serial FU* then will move onto the specification of *the bit-serial FU*.

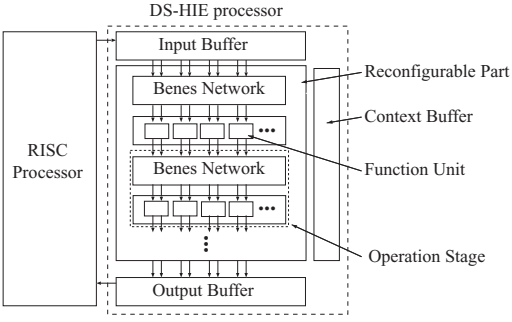


Fig. 1. Block Diagram of DS-HIE processor

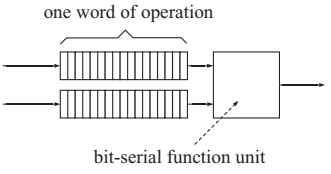


Fig. 2. Method for Timing Adjustment

3.1 Timing Adjustment Unit

Bit-serial computation processes only one bit of data at a time, so that the bit positions of data inputted into a function unit must be aligned when the function requires two or more inputs. To align bit position, we adopted a method which stores two inputs before a BS-FU then feeds the two input data into the BS-FU, as shown in Figure 2. In this method, each buffer stores input data until the another buffer keeps an input data. If both input buffers are filled, then both buffers start to feed the data into its BS-FU at same timing.

The reasons why we adopted this method are as follows:

- The number of *timing adjustment units* required is same as the number of the BS-FU. So it is reasonable to place a *timing adjustment unit* before a BS-FU rather than to place *a timing adjustment unit* independent with a BS-FU.
- For shift operation, we can implement it in a BS-FU within small chip area by utilizing the buffer for timing adjustment placed in front of the BS-FU.
- For multiply operation, we can adopt serial-parallel multiplier [3] by improving *a timing adjustment unit* to feed bit-parallel data as multiplier's input. By the improvement of *a timing adjustment unit*, a BS-FU can be implemented in a small chip area, compared with one adopted serial-serial multiplier.
- A buffer in *a timing adjustment unit* can be used as a storage for storing a constant data.

3.2 Specification

In DS-HIE processor, *bit-width of operation* executed in the BS-FU is 16bits. The reason why 16bit was selected is because the first target application of the DS-HIE processor was image processing. The BS-FU in the DS-HIE processor has two inputs and one output, however the destination of the output can be sent to two different destination.

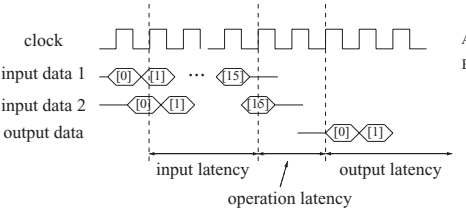


Fig. 3. Timing chart of BS-FU

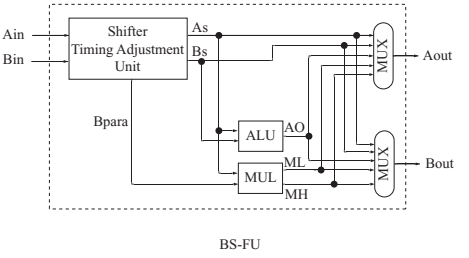


Fig. 4. Block Diagram of BS-FU

Figure 3 shows the timing chart of the BS-FU. It takes 16 cycles to store all bits of an input data since its first bit is stored. These input data are stored in buffers inside of *a timing adjustment unit. The timing adjustment unit* allows input data of the BS-FU to arrive at different timing. After both input data are arrived, *the timing adjustment unit* supplies these data at same timing.

3.3 Organization

Figure 4 illustrates the block diagram of the BS-FU. The BS-FU consists of a *Shifter/Timing Adjustment Unit*, ALU and MUL. *The Shifter/Timing Adjustment Unit* includes a timing adjustment unit described in previous subsection. Further, the MUL in the BS-FU requires bit-parallel data of the input B_{in}, as described later, so that *the Shifter/Timing Adjustment Unit* also provides the bit-parallel data of B_{in}. These functions of *the Shifter/Timing Adjustment Unit* are specified by a context data.

The operations executable in the ALU are ADD, SUB, AND, OR, XOR, NOT, and NOP. The ALU has two bit-serial input port and one bit-serial output port. These operation of the ALU is specified by a context data.

The MUL is a serial-parallel multiplier[3] which one input is a bit-serial data and the another is a bit-parallel data. The serial-parallel multiplier requires a buffer to convert a bit-serial data to a bit-parallel data and save it. The buffer in *the Shifter/Timing Adjustment Unit* is also used as a buffer for the conversion, so we do not need an additional buffer in the BS-FU. The low order word of the product appears at the ML output, and the high order word appears at the MH output. The MUL starts to output high order word after the last bit of the low order word is outputted, so it takes 50 cycles to finish entire multiply operation including the high order word of the product.

The configuration data for the BS-FU requires 16 bits, which consists of the following bits, 6 bits for output select, 4bit for ALU, 4 bits for shift operation and multiplier, and 2 bits for constant data mode.

4 Evaluation of Gate Counts

This section shows the area of two DS-HIE processors: one is the DS-HIE processor with bit-serial computation (we call it DS-HIE processor ver.BS) as we

Table 1. Specifications of DS-HIE processors

	ver. BS	ver. Para
# of Operation Stage	4	4
# of FUs(total)	512	128
Context Buffer	2 entries	2 entries
Input latency of FU	16	1
Operation latency of FU	2	2
Output latency of FU	16	1

Table 2. Synthesis result of each DS-HIE processor

	ver. BS	ver. Para
FU	1.28k	4.68k
Benes Network	18.7k	26.5k
Operation Stage	183k	179k
Reconfigurable Part (total)	731k	716k
DS-HIE processor	1,360k	811k

Table 3. Result of performance evaluation of median filter

	1 data access per cycle		4 data access per cycle	
	ver. BS	ver. Para	ver. BS	ver. Para
execution time	624 ms	3,310 ms	624 ms	1,500 ms
# of context	1	4	1	4
transferred data	220 Mbyte	1,190 Mbyte	220 Mbyte	1,190 Mbyte

previously mentioned, and the another is a DS-HIE processor with bit-parallel computation (we call it DS-HIE processor ver.Para). To fairly compare these DS-HIE processors on performance, this section clarifies the area of these Reconfigurable Part in DS-HIE processors are about same. Table 1 summarizes the specifications of the DS-HIE processors ver.BS and ver.Para.

Table 2 shows the synthesis results. The results were evaluated by NAND gate counts, and the clock frequency of DS-HIE processor ver.BS and ver.Para were 200 MHz and 60 MHz respectively. From the table, we can find that the area of *the Reconfigurable Part* in the DS-HIE processor ver. BS is as about same as that in the DS-HIE processor ver. Para. The total area of the DS-HIE processor ver. BS, however, is larger than that of the DS-HIE processor ver. Para. The reason is that the number of port in *the Input Buffer* and *the Output buffer* are increased and the configuration data is also increased so that required buffers are increased. In this paper, we focus on the area and performance of Reconfigurable Part, so we don't mind the difference.

5 Performance Evaluation

This section shows the results of performance comparison of the DS-HIE processor ver. BS with the DS-HIE processor ver. Para.

In this evaluation, we evaluate the performance on executing median filter with an image with 1280×960 pixels. As the ability of data transfer, we evaluated two cases: one is to access one 16 bits data per cycle, and the another is to access four 16 bits data per cycle.

Table 3 shows the result of performance evaluation which benchmark program is median filter. On the evaluation result of 1 data access per cycle, the execution time of the DS-HIE processor ver. Para is worse than the result of 4 data access per cycle. The reason is that the ability of data transfer is not enough in the case of 1 data access per cycle. On the other hand, the performance of the DS-HIE processor ver. BS is as same as the result of 4 data access per cycle. This result shows the DS-HIE processor ver. BS can achieve higher performance even the condition which the ability of data transfer rate is lower.

6 Conclusion

In this paper, we evaluated a reconfigurable part with bit-serial computation, compared with that with bit-parallel computation. The DS-HIE processor was used as a target processor. From logic-synthesis result, we found that the reconfigurable part with 512 *bit-serial function units* has about same gate counts as that with 128 *bit-parallel function units*. Further, we found that the reconfigurable part with 512 *bit-serial function units* can achieve higher performance than that with 128 *bit-parallel function unit* by keeping a required data transfer rate lower.

Acknowledgments. The part of this work is supported by VLSI Design and Education Center(VDEC), the University of Tokyo in collaboration with Synopsys, Inc. and Rohm Ltd. This research was partially supported by the Ministry of Education, Culture, Sports Science and Technology(MEXT), Grant-in-Aid for Scientific Research, 18300016, 2006.

References

1. Bougard, B., De Sutter, B., Verkest, D., Van der Perre, L., Lauwereins, R.: A coarse-grained array accelerator for software-defined radio baseband processing. IEEE Micro 28(4), 41–50 (2008)
2. Nakamura, T., Sano, T., Hasegawa, Y., Tsutsumi, S., Tunbunheng, V., Amano, H.: Exploring the optimal size for multicasting configuration data of dynamically reconfigurable processors. In: International Conference on ICECE Technology, FPT 2008, pp. 137–144 (December 2008)
3. Richard Hartley, K.K.P.: Digit-Serial Computation. Kluwer Academic Pub., Dordrecht (1995)
4. Tanigawa, K., Hironaka, T.: Evaluation of compact high-throughput reconfigurable architecture based on bit-serial computation. In: International Conference on Field-Programmable Technology (ICFPT 2008), pp. 273–276 (2008)
5. Tanigawa, K., Zuyama, T., Uchida, T., Hironaka, T.: Exploring compact design on high throughput coarse grained reconfigurable architectures. In: Proceedings of the 18th International Conference on Field Programmable Logic and Applications (FPL), pp. 543–546 (2008)

FPGA Implementation of QR Decomposition Using MGS Algorithm

Akkarat Boonpoonga[1], Sompop Janyavilas[1],
Phaophak Sirisuk[1], and Monai Krairiksh[2]

[1] Department of Computer Engineering, Mahanakorn University of Technology, Nongchok, Bangkok 10530, Thailand
akkarat@mut.ac.th

[2] Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok 10520, Thailand

Abstract. FPGA implementation of MGS-QRD is presented in this paper. Mapping conventional QR triangular array of $(2m^2+3m+1)$ cells onto a linear architecture of $m+1$ cells is employed to reduce the number of required QR processors. The architecture for MGS-QRD implementation is discussed, including the structure of a boundary cell (BC) and internal cell (IC). A divider in BC is modified as a Look-Up Table (LUT) and multiplier. The multiplier divided from the divider can be accomplished by sharing it with another multiplier to reduce the resource for BC implementation. Furthermore, the conventional complex multiplication in IC is also modified with three multipliers and four adders. The designed architecture based on discrete mapping of MGS-QRD is implemented to examine FPGA resource utilization. The implementation results show the FPGA performance and resource utilization of MGS-QRD.

1 Introduction

As demand for wireless communication increases rapidly throughout the coming decade, the need for higher data rates will require a technique in order to increase capacity. Recently, multiple-input multiple-output (MIMO) systems have been received considerable interest [1]. It is well known that MIMO systems can significantly improve the throughput of wireless communication systems by transmitting and receiving multiple data streams concurrently [1]. MIMO systems use multiple antennas at both receiver and transmitter. For implementation aspect, a signal processing unit involved in a MIMO receiver should be taken into account. Therefore, an implementation method which provides new architectures with high performance requirements becomes challenging.

Matrix inversion and triangularization such as QR decomposition (QRD) or singular value decomposition (SVD) are important for all MIMO receivers. The QR decomposition can be done by using Givens rotations (GR) [2], Householder reflection (HR) [3], or Modified-Gram-Schmidt (MGS) [3]. The MGS is a slightly

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 394–399, 2010.

modified and numerically superior version of the Classical Gram-Schmidt algorithm. It was proven in [3]-[4] that the MGS is numerically identical to GR offering the same level of numerical stability and accuracy. The number of operations for QRD using MGS and GR of matrix is almost identical when $M \geq N$[5]. The MGS requires fewer operations than GR and HR when $M \geq 2N/3$.

Due to the complexity of the MGS-QRD, the MIMO receivers are traditionally implemented on digital signal processors (DSPs), such as the Bell Labs layered space-time (BLAST) system. Since it does not support parallel computation, the speed of the DSP implementation is often limited, especially as the number of antennas increases. As the main drawback of DSP, a field programmable gate array (FPGA) device is widely used to meet the specific requirement in MIMO systems. An efficient architecture, a triangular systolic array, was proposed for QRD to achieve highly parallel computation. However, to implement it, for example on FPGA, the utilized resource for hardware implementation may be very large. This is not practical when the number of antenna element is extremely increased. To reduce the complexity of hardware implementation, mapping QRD systolic array onto another architecture was proposed [6]-[7]. Although the approach can significantly reduce hardware complexity, recursive least square (RLS) algorithm is not suitable for high data rate applications because it need convergence time to obtain an optimal solution.

This paper presents FPGA implementation of MGS-QRD for least square (LS) MIMO systems. The architecture based on discrete mapping is employed to implement MGS-QRD. The structure of processor cells is designed for complex-value matrix. The implementation results are shown to compare the performance and resource utilization of different structures.

2 Architectural Design for MGS-QRD

One of the most popular factorization is QR decomposition which generally employed to invert a matrix. The applications of QRD for solving the least square problems include adaptive antenna and MIMO systems etc. To achieve QRD via triangular systolic array, $(N^2 + N)/2$ processor cells are required for a $N \times N$ square matrix. Although the systolic array enable highly-parallel computation, a direct implementation of systolic array would not be practical because of high operation complexity.

In [6], the QRD using GR was applied to RLS problems to find a weight vector of adaptive beamforming systems. In the approach, a triangular array of $(2m^2+3m+1)$ cells can be mapped onto a linear architecture of $(m+1)$ processors comprising a BC and m ICs. In this paper, we utilize the mapping technique of QR systolic array based on MGS algorithm for LS problem in MIMO systems. Fig. 1 depicts the architecture of QR processing unit comprising processor elements (PEs), data bus, and control unit. The unit is based on mapping the triangular systolic array onto the linear array architecture. All the BC operations of a systolic array are assigned to one processor, while all the IC operations are implemented on a row of separated processors. Even if the number of ICs is reduced less than one row compared with

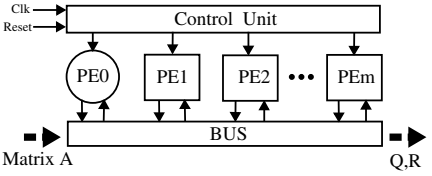


Fig. 1. MGS-QRD processing unit

the conventional systolic array, at least one BC is required. According to fig. 1, the PE0 is BC and others PE are ICs. Each PE performs for more than one clock cycle. This is due to mapping the conventional systolic array onto a linear array architecture. The number of the PEs depends upon the matrix dimension. The control unit is used for exchanging data among PEs via the data bus. To achieve the efficient QR processing unit, we consider not only the technique of reducing the number of processor cells but also the hardware architecture of BC and IC as shown next below.

The structure of BC and IC proposed in [8] computes only the real-value matrix. In this paper, the hardware architecture of a MGS-QRD processing unit for a complex-value matrix is implemented. The QR decomposition can be separated into two steps. First, the diagonal line elements (r_{ij}) of the upper triangular matrix R and the unitary matrix Q are computed. The diagonal line element is obtained from the square root of the summation of the square of each element in the column j of $\mathbf{a}(\mathbf{a}_j)$. This process is indicated by circular of BC in fig 1. Fig. 2 illustrates a structure of the BC comprising two multipliers, two adders, two dividers and one square root. Because of iterative structure of BC, the $N+1$ delay units inserted at the uppermost and lowest of the structure is used to delay the matrix input for computing q_j. Note that the BC structure contains two dividers as seen in the figure. Generally, a divider would consume too many hardware resources. Thus, the division, namely A/B, is modified as (1/B)A which is accomplished by using a look up table (LUT) and multiplier. Since one part of the modified division is a multiplier, it can be done by sharing hardware with another multiplier when the multiplier does not handle computation. Fig. 3 shows the modified BC structure of MGS-QRD. Note that one of the dividers is eliminated. The division for $1/r_{ij}$ is easily accomplished by a LUT, it needs small resource for hardware implementation.

Second, $\mathbf{q}_j$ value from BC is sequentially feed to IC and the non-diagonal line element r_{ij} of matrix $\mathbf{R}$ is computed. The resulting r_{ij} is then used to compute a new matrix input $\mathbf{A}(\mathbf{a}_i^p)$ as seen in the fig. 4 in which the structure of an IC is illustrated. In the figure, the new matrix input $\mathbf{a}_i^p$ is obtained from the subtraction of the matrix input and the product of r_{ij} and $\mathbf{q}_i$. Multiplexer and demultiplexer are employed to select operation for r_{ij} and $\mathbf{a}_i^p$ in order to reduce the utilized resource of implementation. Complex multiplication in the IC generally comprises four multipliers and two adders. Following the derivation in [9], operation for complex multiplication can be modified as three multiplier and five adders. The technique is therefore applied to our proposed architecture.

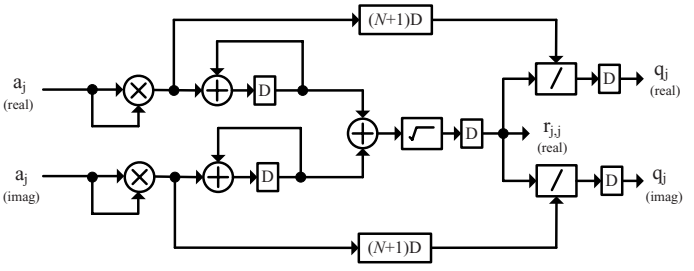


Fig. 2. Structure of BC

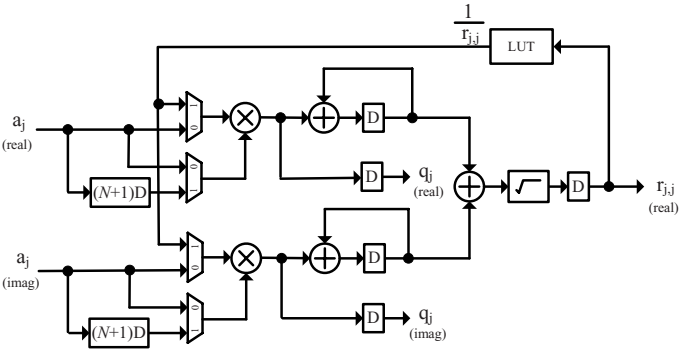


Fig. 3. Structure of a modified BC

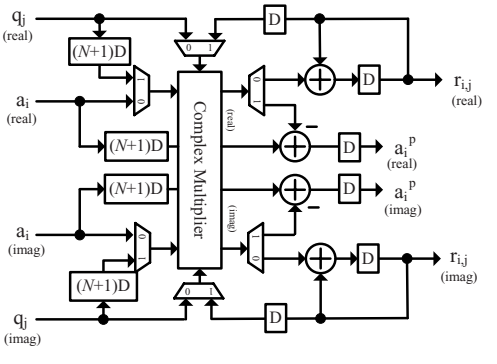


Fig. 4. Structure of an IC

3 FPGA Implementation Results

To examine the FPGA resource utilization, the designed architecture of MGS-QRD was implemented on the Xilinx Virtex2-Pro XC2VP30 device. The FPGA resource utilization of BC with and without sharing multiplication is compared. We classify multipliers used in BC into four different implementation techniques i.e. 1. multiplier block without sharing multiplication in BC 2. parallel multiplier without sharing multiplication in BC 3. multiplier block with sharing multiplication in BC and 4. parallel multiplier without sharing multiplication in BC. We found that the utilized resource for BC using and not using multiplier block is almost equal. In addition, the use of hardware sharing of multipliers consumes smaller resource than that of without sharing of multipliers.

The utilized resource for FPGA implementation of conventional and modified complex multipliers is also investigated. It is clear that the resource obtained by the use of modified complex multipliers is smaller than that of conventional complex multipliers. The resource utilization of IC with modified complex multipliers can be reduced about 15.5% of equivalent gates.

The designed architecture according to fig. 1-4 is implemented on the Xilinx Virtex2-Pro XC2VP30 device. The architecture is based on discrete mapping discussed above. Table 1 shows comparison of utilized FPGA resource and performance of MGS-QRD processing unit with different input-matrix dimensions. Note that the equivalent gate count of the unit for odd and even matrix dimension, for example 2x2 matrix and 3x3 matrix, is equal because of the same number of PEs. In total, it takes $(2N-1)(2N+1)-(2N-2)(N-1)$ and $(2N)(2N+1)-(2N-1)(N-1)$ cycles of the linear architecture to complete one specific QR update for odd and even matrix dimension, respectively. Furthermore,the processing unit proposed in [8] comprises four processing elements (including BC and ICs) and takes 44 clock cycles to compute QRD for a 4×4 square matrix. In paper, it take only three processing elements (including BC and ICs) and take 55 clock cycles. Although, the cycle time of computing QRD in our approach is much than that in existing architecture, the throughput of

Table 1. Comparison of utilized FPGA resource and performance of MGS processing unit with different input-matrix dimensions

$N \times N$	No.PE	Time Unit	Total Cycle	Equivalent gate	Throughput (M)
2×2	2	4	17	16,820	7.22
3×3	2	5	27	16,820	3.82
4×4	3	8	51	24,058	2.40
5×5	3	9	67	24,058	1.66
6×6	4	12	101	31,297	1.22
7×7	4	13	123	31,297	0.94
8×8	5	16	167	38,535	0.74
9×9	5	17	195	38,535	0.60
10×10	6	20	249	45,634	0.50

the proposed architecture implemented on FPGA (2.4 Msps)is higher than that of the existing architecture implemented on ASIC (1.47 Msps). This is becauce that there is a repetitive section meaning that the next input can be feed to the processor before QRD calculation is completed [6].

4 Conclusions

The MGS-QRD based on linear mapping proposed in [6]-[7] has been implemented on FPGA. The structure of BC with multiplication sharing and IC with modified complex multiplication is shown. The implementation results have shown that the MGS-QRD with designed architecture consumes smaller resource than that with conventional architecture (without mapping, hardware sharing and modified complex multiplier).

Acknowledgments. Monai Krairiksh thanks the Thailand Research Fund (TRF) which provides financial assistance under contract No. RTA 5180002.

References

1. Gesbert, D., Shafi, M., Shiu, D., Smith, P.J., Naguib, A.: From Theory to Practice: An Overview of MIMO Space-Time Coded Wireless Systems. IEEE Journal on Selec. Areas in Commu. 21(3), 281–302 (2003)
2. El-Amawy, A., Dharmarajan, K.R.: Parallel VLSI Algorithm for Stable Inversion of dense matrics. IEEE Proceeding on Computer and Digital Technique 136(6), 60–75 (2005)
3. Golub, G.H., Van Loan, C.F.: Matrix Computation. Johns Hopkins, New York (1996)
4. Bjork, A.: Numerical Methods for Least Square Problem. AP-SIAM, Philadelphia (1996)
5. Sing, C.K., Prasad, S.H., Balsara, P.T.: A Fixed-point Implementation for QR Decomposition. In: IEEE Proceeding Dallas Workshop Circuits Systems, pp. 75–78 (2006)
6. Lightbody, G., Woods, R., Walke, R.: Design of a Parameterizable Silicon Intellectual Property Core for QR-Based RLS Filtering. IEEE Trans. on very large scale integration (VLSI) systems 11(4), 659–678 (2003)
7. Lightbody, G., Walke, R., Woods, R., McCanny, J.: Novel mapping of a linear QR architecture. In: ICASSP 1999, pp. 1933–1936 (1999)
8. Lin, K., Lin, C., Chang, R.C., Huang, C., Chen, F.: Iterative QR Decomposition Architecture Using the Modified Gram-Schmidt Algorithm. In: ISCAS 2009, pp. 1409–1412 (2009)
9. Benhamid, M., Othman, M.: FPGA Implementation of a Canonical Signed Digit Multiplier-less based FFT Processor for Wireless Communication Applications. In: ICSE 2006, pp. 641–645 (2006)

Memory-Centric Communication Architecture for Reconfigurable Computing

Kyungwook Chang and Kiyoung Choi

Department of Electrical Engineering and Computer Science
Seoul National University
Seoul, Korea
{kyungwook222,kchoi}@poppy.snu.ac.kr

Abstract. This paper presents a memory-centric communication architecture for a reconfigurable array of processing elements, which reduces the communication overhead by establishing a direct communication channel through a memory between the array and other masters in the system. Not to increase the area cost too much, we do not use a multi-port memory, but divide the memory into multiple memory units, each having a single port. The masters and the memory units have one-to-one mapping through a simple crossbar switch, which switches whenever data transfer is needed. Experimental results show that the proposed architecture achieves 76% performance improvement over the conventional architecture.

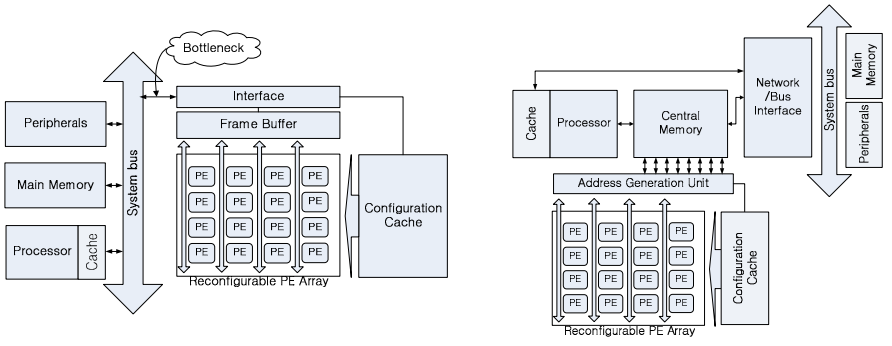
Keywords: memory-centric, CGRA, reconfigurable array architecture, communication overhead.

1 Introduction

Coarse-grained reconfigurable array architecture (CGRA) has been proposed to tackle both issues of flexibility and performance [1]. A typical CGRA has a reconfigurable array of processing elements (PEs) to boost its performance through parallel execution. Each PE is a small sized execution unit that can be configured dynamically and such dynamic configurability renders the flexibility of CGRA.

Fig. 1 (a) shows a generic template of conventional CGRA [1]. It has a two dimensional reconfigurable array of PEs for parallel computing. The PEs do not have address generation units to keep their size small. So they receive/send input/output data from/to a memory called frame buffer passively through a set of data buses running for each column of the array. As shown in Fig. 1 (a), a typical CGRA has two storage units. One is configuration cache and the other is frame buffer. Configuration cache feeds instructions which are executed by the PEs. It also contains information on the time when the data in the frame buffer should be loaded on the data bus and when the data on the bus should be stored into the frame buffer, so that the PEs are able to receive and send data without separate address generation units. These instructions can be stored only at system initialization stage, such that it does not affect the system performance.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 400–405, 2010.



(a) Conventional architecture (b) Proposed architecture

Fig. 1. The conventional CGRA and the proposed CGRA

Since the frame buffer is used to feed input data to the PEs in the array and to store output data from the PEs, enormous amount of data goes in and out of the frame buffer for most multimedia applications. This can be a serious burden to the system bus and controllers such as a processor or a DMA unit. Usually, in such a CGRA, the frame buffer and the PE array are tightly coupled with each other, and so the communication between them is not a problem. On the other hand, outside the array, reading/writing data from/to the frame buffer on the side of processor or main memory becomes a serious bottleneck for the entire system performance due to the limited bandwidth and latency of the system bus. According to our simulation result, the communication overhead can be five times longer than the computing time of the array, depending on the application.

To overcome such limitation, reconfigurable array architecture with a multiple banked frame buffer has been proposed [2][3]. It enables the architecture to do double buffering for hiding the communication overhead, which helps in some case. However, it does not work well when the application has a cyclic data dependency, and the limited bandwidth of the system bus can cause serious performance degradation. So it is crucial to reduce the communication overhead itself, to improve the overall system performance.

To cope with this limitation, in this paper, we propose a memory-centric communication architecture where data is transferred through a scratch pad memory (SPM) without passing through a traditional system bus. It improves the system performance significantly, especially when the communication overhead cannot be hidden due to cyclic data dependency.

2 Proposed Architecture

2.1 Architecture Overview

Fig. 1 (b) shows an overview of the proposed architecture. The whole system consists of a processor with caches, a PE array with configuration caches and address generation units, an interface to a network or bus which provides interconnection to the main memory and peripherals, and a central memory tightly coupled with each component

through a crossbar switch. The central memory works as an SPM to the processor and as a frame buffer to the PE array at the same time.

2.2 Central Memory

Fig. 2 shows the concept of the central memory. Data communication takes place through this memory instead of a system bus. The central memory is divided into multiple memory units, each having a single port. As shown in the figure, the masters (including the processor, the PE array, and the network/bus interface) and the memory units have one-to-one mapping, so that each master accesses only one memory unit at a time. For the communication between masters, the data can be transferred without copying it, but by just changing the mapping information.

The central memory is divided into two regions: shared region and exclusive region. Shared region contains configuration information for the central memory including the mapping information between masters and the memory units. Control signals (mainly for synchronization) or small sized data transferred between the masters are also stored in the shared region. On the other hand, data to be transferred in a large quantity from one master to another is stored in the exclusive region. The exclusive region is divided into K units according to the number of masters connected to the central memory. These K units share the same logical addressing space, but use different physical memory blocks. Since no system bus is involved in the data transfer, this memory structure enables masters to simultaneously access the exclusive regions to fetch input data and store output data without bus contention, in contrast to the conventional architecture. Unlike other masters, the PE array needs to fetch multiple data concurrently within a cycle and so each exclusive region is again divided into multiple (N) blocks with each block attached to a column of the PE array and with the bit-width of each block equal to that of a PE's datapath.

Regarding to the implementation, the shared region is implemented with a multi-port register file and the exclusive region is implemented with multiple one-port memories. Because the cost and power consumption of the multi-port register file increase rapidly with the size, the size of the shared region should be minimized. On the other hand, the exclusive region of the central memory uses multiple one-port

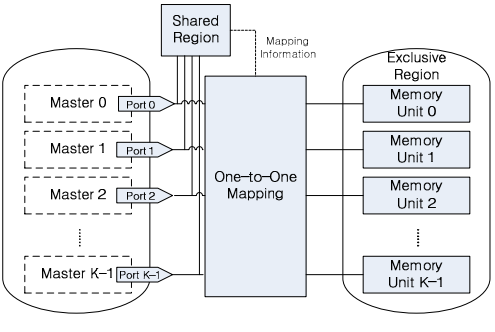


Fig. 2. The concept of the central memory

memories instead of one big multi-port memory not to increase area and power overhead. According to our experiments, total area of the proposed memory structure with two one-port memories (including the crossbar switch) is 22.57% smaller than that of the dual-port memory on average, and the dynamic power and leakage power is reduced by 43.09% and 22.15%, respectively.

In terms of performance, one may expect slight degradation since the communication requires changing the mapping configuration in the proposed memory structure. However, the change can be done within one cycle, and it is not a problem for data communications in a large quantity (for word-by-word or small-sized data communications, one can still rely on the shared region). Even if we use one big multi-port memory, there will be an overhead of changing the base address to the next chunk of data, which actually takes more cycles than just switching the mapping.

Moreover, by dividing exclusive regions physically and using a crossbar switch, a master can only access one memory unit at a time. Thus we can avoid memory conflict or race condition and also eliminate the risk of corrupting other master's memory. For a safe use of multi-port memory, it is desirable to add extra memory protection hardware to prevent memory conflict or race condition, which is not needed in the proposed memory structure.

In the proposed multiple one-port memory structure, we limit the number of masters to four, since we are using a crossbar switch which is not quite scalable.

2.3 PE Array

In the proposed architecture, to read/write data, PEs should invoke a read/write request to the address generation unit. Since the PEs have bit width limitation, they may not be able to cover full address space. Thus instead of sending the full address for data transfer directly to the central memory, they pass two arguments – one for the index to the data address table and the other for the offset – to the address generation unit, and the address generation unit does actual memory access. The address generation unit contains a table of base addresses to the data to be accessed by the PEs. The slots of the table can be filled up during system initialization or while the system is running. When it receives a request, it finds the base address in the table using the index passed through the data bus from the PE as one of the arguments. Then it adds the base address to the offset which is passed as another argument, generating the full address to the central memory for read or write operation.

3 Experiments

The experiments have been performed on a conventional CGRA as well as the proposed CGRA, both modeled using SoC Designer 7.0 [4] with cycle accurate components. The conventional architecture consists of an ARM7TDMI with 64KB cache as the processor, a DMA controller, and a PE array with a frame buffer whose size is 6KB. The proposed architecture consists of an ARM7TDMI with 64KB cache as the processor, a central memory, a PE array (without frame buffer), and a network/bus interface. The central memory has three units, 2KB each, for the exclusive region and 64B for the shared region.

Table 1. Simulation Result

			Conventional			Proposed			Speed up	
			Total Cycles	Comm. overhead Cycles	Comm. overhead Ratio	Total Cycles	Comm. overhead Cycles	Comm. overhead Ratio	Total	Comm. Decreased
Non-cyclic	JPEG Decoder		4321	0	0.0%	4278	0	0.0%	1.01	-
Cyclic	MPEG-4 Decoder	Intra	16796	5905	35.2%	10863	0	0.0%	1.55	-
		Inter	20694	11205	54.2%	12858	1337	10.4%	1.61	88.1%
	Physics Engine		5952892	3038673	51.1%	2805502	785783	28.0%	2.12	74.1%
	Average		-	-	46.8%	-	-	12.8%	1.76	81.1%

Both non-cyclic data dependent application and cyclic data dependent application have been used for the experiments. For the non-cyclic data dependent application, JPEG decoder has been chosen. Dequantization and IDCT process of the JPEG decoder have been mapped to the PE array and all other processes have been mapped to the processor. However, for Huffman decoding process, since general purpose processor is not suitable for bit manipulation, the method proposed in [5] has been applied.

For the cyclic data dependent application, MPEG-4 decoder and 3D crash-wall simulation physics engine have been used. As in the JPEG decoder, the method in [5] has been used for the Huffman decoding process in the MPEG-4 decoder. And the interpolation, dequantization, and IDCT processes are mapped to the PE array. All other processes including post process and header decoding process are mapped to the processor. The 3D physics engine actually consists of floating point operations, for which we have adopted the method in [6]. We have mapped compute-intensive tasks to the PE array, and control-intensive tasks to the processor.

Table 1 shows the simulation result. For the JPEG decoder, since it does not have cyclic data dependency, it is possible to use double buffering in the conventional architecture to hide communication overhead. Thus, as the simulation results show, the communication overhead is hidden into the computation cycles of the processor to which the Huffman decoding is mapped. However, because the conventional architecture uses the system bus as a communication medium, there can be bus contention. This is why the conventional architecture takes more execution cycles than the proposed architecture.

In the MPEG-4 decoder, since the task data dependency is somewhat different when we decode intra-prediction frames and inter-prediction frames, we measure the simulation result separately. The numbers in cycles represent average number of cycles taken to decode one macroblock. As the simulation result for the intra-prediction shows, in the conventional architecture, the processor spends more than 35% of the total execution time on busy-waiting for the completion of input/output data copy. However, in the proposed architecture, it removes the communication overhead completely achieving 55% overall performance improvement. For the inter-prediction whose data dependency is much denser than that of the intra-prediction, in the conventional architecture, it suffers from a serious memory bandwidth bottleneck problem. The simulation result shows that the processor spends over 54% of the total execution time to wait for transferring data from the main memory to the frame buffer. On the other hand, in the proposed architecture, the processor idle time is measured to be only 10.4% of the total execution time.

In the simulation of 3D graphics physics engine, more than 50% of the total execution time is spent on transferring input data and output data in the conventional architecture. For the proposed architecture, the communication overhead is reduced to about 28%.

4 Conclusion

In this paper, we propose memory-centric CGRA, which reduces communication overhead by establishing a direct communication channel through a central memory. Instead of copying data between main memory and frame buffer, which is done in conventional CGRAs, the proposed approach uses only the central memory as the communication channel. It avoids using costly multi-port memory by switching the mapping between masters and memory units. The simulation results with multimedia applications show that our approach reduces communication overhead by over 80% and improves overall performance by 76% on average.

Acknowledgement

This work was supported by KOSEF under NRL Program Grant (R0A-2008-000-20126-0) funded by MEST, by IITA under ITRC support program (IITA-2008-C1090-0804-0009) funded by MKE, and by Nano IP/SoC Promotion Group under Seoul R&BD Program (10560).

References

1. Hartenstein, R.: A decade of reconfigurable computing: a visionary retrospective. In: Proc. DATE (2001)
2. Singh, H., et al.: MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive application. IEEE Trans. on Computers 49(5) (2000)
3. Kim, et al.: Design and evaluation of coarse-grained reconfigurable architecture. In: Proc. ISOCC (2004)
4. Carbon SoC Designer, `http://www.carbondesignsystems.com/Products/SoCDesigner.aspx`
5. Lee, H., Choi, K.: Multi-codec variable length decoder design with configurable processor. In: Proc. ISOCC (2008)
6. Jo, M., et al.: Implementation of floating-point operations for 3D graphics on a coarse-grained reconfigurable architecture. In: Proc. IEEE SOCC (2007)

Integrated Design Environment for Reconfigurable HPC

Lilian Janin, Shoujie Li, and Doug Edwards

School of Computer Science, The University of Manchester,
Manchester M13 9PL, United Kingdom
{lilian.janin,shoujie.li,doug.edwards}@manchester.ac.uk

Abstract. Using FPGAs to accelerate High Performance Computing (HPC) applications is attractive, but has a huge associated cost: the time spent, not for developing efficient FPGA code but for handling interfaces between CPUs and FPGAs. The usual difficulties are the discovery of interface libraries and tools, and the selection of methods to debug and optimize the communications. Our GALS (Globally Asynchronous Locally Synchronous) system design framework, which was originally designed for embedded systems, happens to be outstanding for programming and debugging HPC systems with reconfigurable FPGAs. Its co-simulation capabilities and the automatic re-generation of interfaces allow an incremental design strategy in which the HPC programmer co-designs both software and hardware on the host. It then provides the flexibility to move components from software abstraction to Verilog/VHDL simulator, and eventually to FPGA targets with automatic generation of asynchronous interfaces. The whole design including the generated interfaces is visible in a graphical view with real-time representation of simulation events for debugging purpose.

Keywords: hardware-software interface generator, asynchronous, GALS.

1 Introduction

Using FPGAs to accelerate HPC applications involves a huge cost in time. Developing efficient FPGA code is far from being the most time-consuming part of the process. The main problem usually comes from handling the interface between CPU and FPGA: figuring out which libraries to use and how to debug the communications. In fact the difficulties are:

- the choice and description of the interface between the main software and the FPGA implementation;
- how to setup an environment that enables the software programmer to co-design and debug his HPC+FPGA application.

One feature that is usually difficult to achieve is the ability to design and debug the HPC+FPGA application on a separate non-HPC host. This leads to tremendous increases in efficiency as the programmer is free from the HPC constraints: remote text-based terminal, delays for processes to be scheduled, and dynamic compute node

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 406–413, 2010.

allocation. Of course, the debugging environment also needs, at some point, to be able to target directly the HPC environment as specific bugs may appear at that stage only. The design environment proposed here allows HPC designers to:

- Design and debug their whole design on a non-HPC host, by linking the software code to Verilog or VHDL simulators;
- Remotely target the real HPC system while keeping debugging feedback in the IDE;
- Move components one by one from software abstractions to hardware with the new interfaces being automatically regenerated.

1.1 Background

In order to open up the FPGA market to software programmers, a variety of C-like programming languages have appeared and are available to the HPC programmer [1]: Mitrion-C [2], Celoxica with Handle-C [3], and Nallatech Dime-C [4] being the main ones. These languages provide a higher level of abstraction than conventional Verilog or VHDL and appear more familiar to software HPC developers, leading to shorter development times. However, one of the real benefits of these languages is that they are provided with complete integrated design environments, pre-configured for specific HPC systems. These environments are able to handle the complexity of interfacing CPUs to FPGAs themselves, freeing the user from what we believe is the most difficult task.

The design environment presented in this paper provides similar benefits for interfacing automatically the software running on CPUs to FPGA code described in Verilog or VHDL. Programming in Verilog and VHDL also has some desirable properties: code efficiency and control of the implementation details are sometimes necessary as the FPGA clock runs ten times slower than CPU clocks. Although these HDL languages may require significantly longer development times, their availability for small efficient accelerators is important. With experience, shorter design times can also be achieved as pre-programmed IPs and open-source modules are also available for re-use in these hardware languages.

2 GALAXY Design Framework

The GALAXY framework [5] was originally designed for GALS (Globally Asynchronous Locally Synchronous) embedded system design. It aims at providing an environment of development for iterative design and prototyping of embedded systems where the circuit designer can refine the system description from high levels of abstraction to lower levels, and from software simulation to FPGA prototyping. Components are handled independently at any level of abstraction, targeting any simulator, and the communication interfaces between components, between abstractions and between simulators are automatically regenerated for each simulation. Through the use of various FPGA prototyping boards, we discovered that the GALAXY IDE and tools could help greatly in the task of HPC acceleration.

The framework provides a graphical IDE where software and hardware components are represented as entities (Fig. 1). Each of these components can have multiple

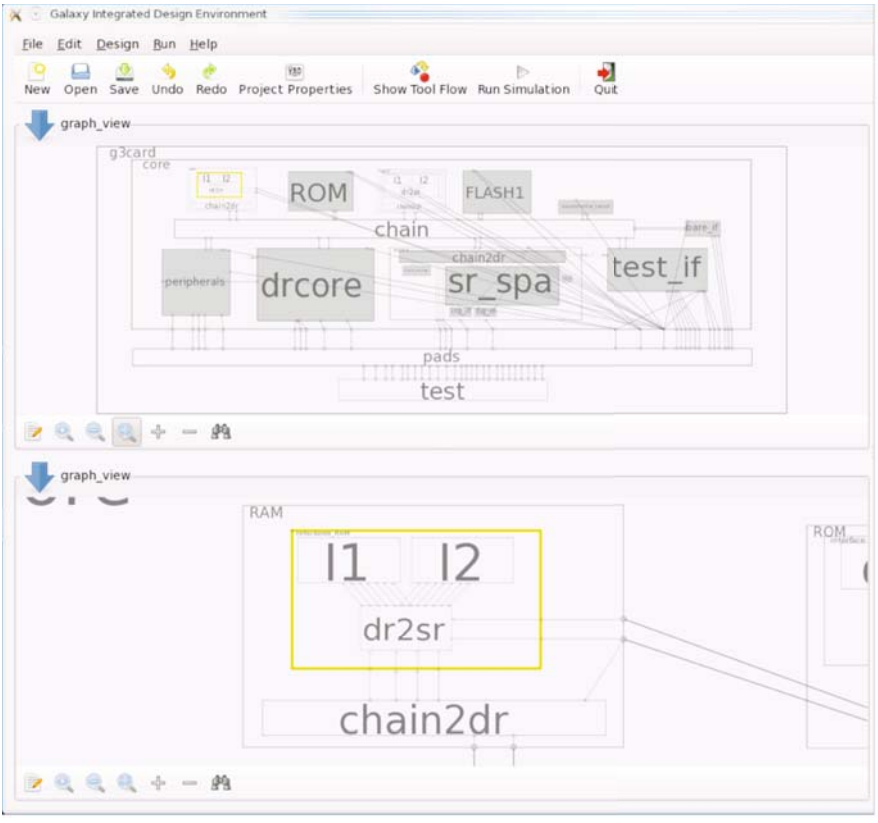


Fig. 1. GALAXY Integrated Design Environment – Dual Graph View

implementations (for example an FFT component can have a software implementation in C calling a library and a hardware implementation in Verilog) and implementations can be switched from one to another at the click of a mouse.

For each component the user selects a "simulation target" (where simulation actually includes anything from software execution to FPGA boards) dependent on its source code description: a C description can be executed on host, whereas a Verilog description can be simulated in a Verilog simulator or synthesized and sent to an FPGA. If C synthesizers are available and added to GALAXY's tool flow system, Verilog simulators and FPGA targets become automatically available to components described in C (this is also applicable to any other language).

When two connected components are set to different simulation targets, the communication links are replaced by asynchronous components following a delay-insensitive protocol. This allows the user to experiment with several architectures before optimizing the critical paths, and appears to be an efficient way to proceed.

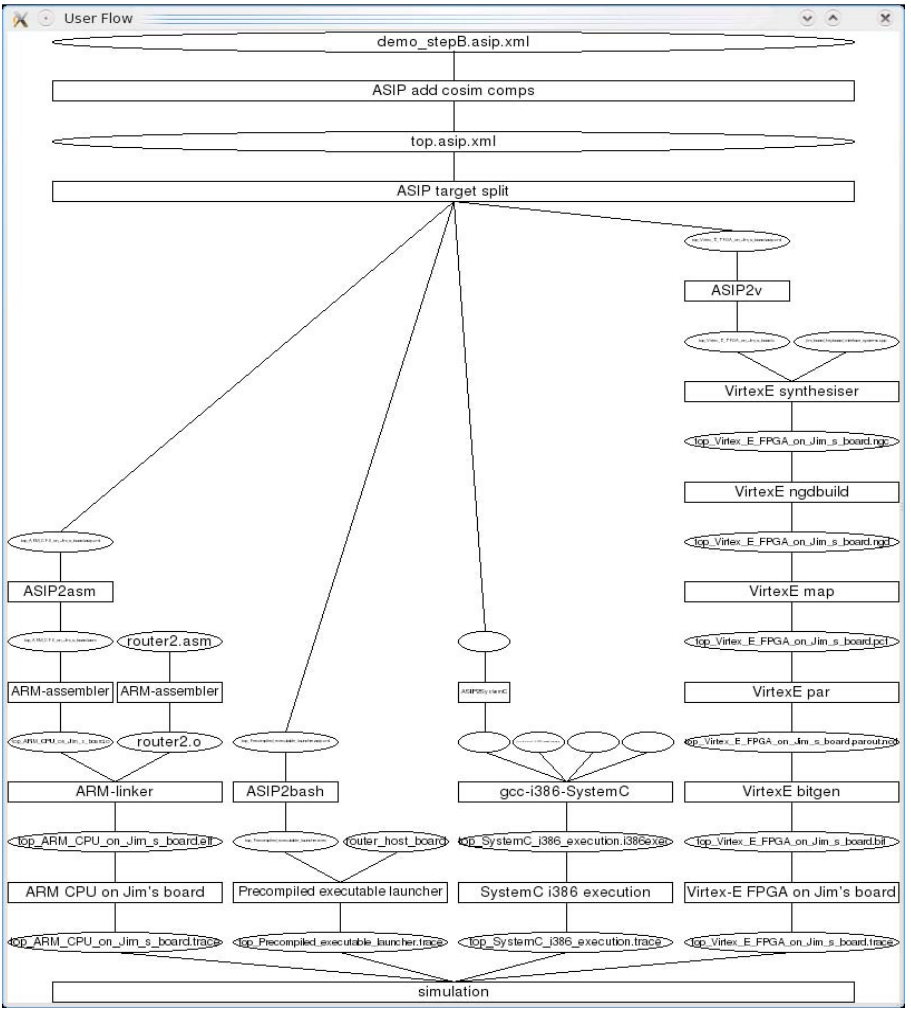


Fig. 2. GALAXY Tool Flow View showing back-end tools

2.1 The ASIP (Asynchronous-Synchronous IP) XML Format

ASIP is a standard component-based description format describing the tree of components making up a circuit. It was developed as part of the GALAXY project. Starting from a top-level component, each component is described in a standard hierarchical way, with a specification of its interfaces, sub-components and connections between sub-components. The leaf components contain a description of their associated source code in any HDL.

What makes ASIP interesting is a set of special constructs, which define the assumptions made to be able to refine a system into asynchronous parts and the constraints on the system description these assumptions impose:

- Asynchronous channels: In the description of the interface and connections, the standard wire and TLM socket types are available, but also an asynchronous channel type. Asynchronous channels are associated to asynchronous protocols. This new channel type allows further design exploration, where the designer can try out various protocols, for example to check for link efficiency. Protocol adapters are automatically added in the GUI where necessary, clearly showing to the designer where bottlenecks may arise.
- Multiple implementations: For each ASIP component, the designer can provide multiple implementations. They must describe the same behaviour and share the same interface, but can be in different languages and at different levels of abstraction.
- Multiple interfaces + transactors: This feature was brought in after long discussions, and makes the GALAXY framework unique: for each ASIP component, the designer can provide multiple interfaces! The problem is that other components in the system might expect one particular interface, and letting the user switch a component's interface could invalidate some connections. For this reason, some constraints apply: proper use of multiple interfaces can be achieved by providing interfaces that are functionally equivalent. The ASIP format encourages this by requiring transactors between the various interfaces. In practice, interfaces are supports for the same communications at different levels of abstraction, with/without debugging signals, and to support synchronous-asynchronous IP wrapping in the GALS context.

An ASIP component can therefore have many implementations and multiple interfaces. Adapters between the different interfaces are included in the ASIP description as transactors, letting the designer select an interface at one level of abstraction for the IP, and a different level of abstraction for its implementation. For example, if an IP is included in a system using its pin-level interface, and the architect needs to simulate it at the SystemC TLM level, an inconsistency is raised and the transactor "pin-level to TLM" is automatically inserted to simulate the component using its TLM description and adapt the transactions to the pin-level interfaces of the connected components. The ability to provide and switch easily between multiple implementations and interfaces, together with the presence of transactors between interfaces, allow a very efficient design space exploration, where IPs can be switched between levels of abstraction in a single operation. In most cases, the transactors can even be automatically generated, for example when TLM sockets are mapped to asynchronous channels.

2.2 GALAXY Back-End Tool Flow

ASIP descriptions are processed by a collection of back-end tools to achieve the regeneration of interfaces for transparent co-simulation for every change in the system architecture. The ASIP flow is as follows (see also an example with four simulation targets in Fig. 2):

- *Asip-add-cosim-comps* inserts co-simulation components in the ASIP file. Each time two connected components are set to be simulated on a different simulator, the connection is split into two and a co-simulation component is added at each end. The co-simulation components behave as if they were

wirelessly transferring the data to each other. After this step, the graphs of components corresponding to each simulator are fully disconnected from each other, even though they are still all contained in the same ASIP file. In a HPC environment, these components are programmed to use the HPC communication libraries provided with the FPGA system.

- *Asip-target-split* creates one ASIP file per simulator. All the components targeting a specific simulator are copied to the corresponding file. Incidentally, a flattening of the ASIP structure is also performed at this stage. All the hierarchy is removed and each resulting file is made of one top level components containing directly all the leaf components.
- Each ASIP file is then processed by the code generators *asip2systemc, asip2v, asip2vhdl, asip2asm* or *asip2bash* to generate the top-level source code in an appropriate language for the selected simulator. Keeping the same structure as the input ASIP file, the generated source code is a top-level code instantiating and linking together the various IP source codes. Due to the ASIP flattening occurring in *asip-target-split*, the generated source code is actually a top-level procedure instantiating user-written modules. Most of the time, intermediate modules are created to rename and reorder the signals in order to match the user-defined component interfaces.
- Finally, each generated source code undergoes its own flow as shown in each branch in Fig. 2, specific to the targeted simulator. It can be compiled, synthesized, placed and routed, sent to FPGAs, executed on the host or interpreted. The information about the available tools is stored in the Tool Flow System.

2.3 Tool Flow System: Execution of Tool Flows

The GALAXY IDE also serves as a front-end to launch all the tools, internal backend tools or external tool flows. A tool flow window allows the user to control these tools, and an execution window provides means of interaction with these tools.

The Tool Flow System is the gateway to link external (vendor or open-source) tools to the GALAXY framework. It has a well-defined interface to facilitate the integration of new tools and tool flows. Its aim is to generate, from the knowledge of available tool flows, an execution sequence of tools to go from the ASIP and IP source codes to the simulators. To achieve this, the tool flow database is made of three main sections:

- available file formats (including how to recognise them, e.g. from their extension);
- available simulators, specifying which input file formats they require;
- available translators (tools able to convert one file format into another, such as compilers and synthesizers), with their required input and produced output file formats.

From this information, a graph of tools and file formats is created (Fig. 3), and appropriate execution sequences are generated when the user desires to “simulate file X with simulator Y”.

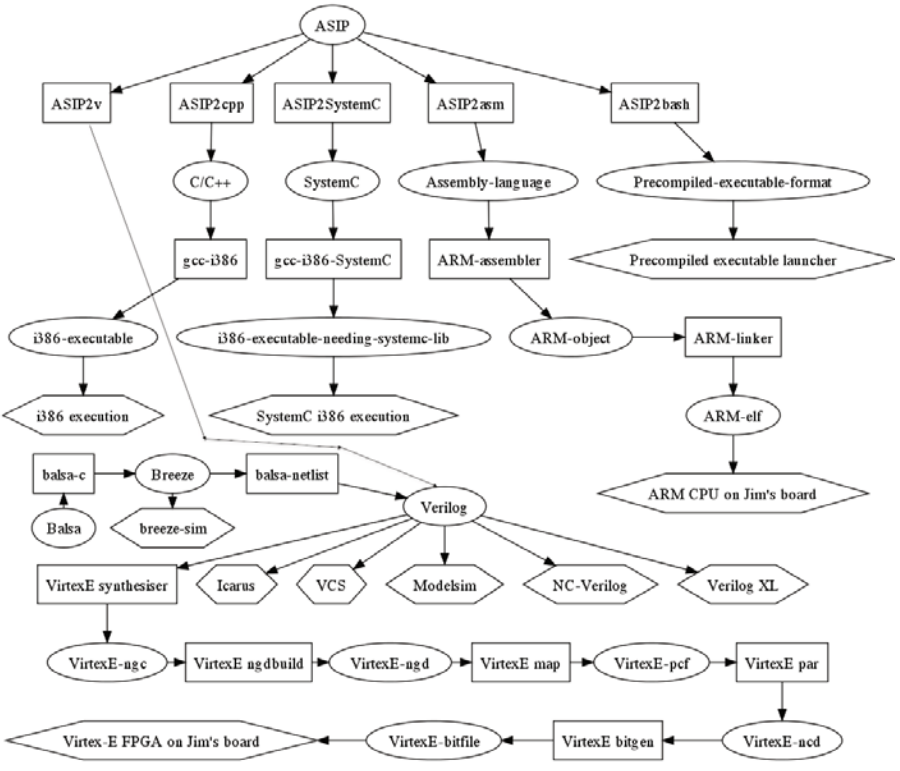


Fig. 3. GALAXY Internal and External Tool Flow

2.4 Ability to Design and Debug on a Separate non-HPC Host

The GALAXY IDE is usually running on a host with graphical display, therefore not directly on the HPC system. In the first stage of the design process, the HPC designer can design and simulate the HPC application on the non-HPC host, giving him the advantage of faster access and direct control over the hardware simulators. This leads to tremendous increases in efficiency, as the programmer is free from the HPC constraints: remote text-based terminal, delays for processes to be scheduled, dynamic compute node allocation.

A good way to run and debug the system on a single non-HPC host is to have the FPGA code simulated in a Verilog simulator, or even better: a graphical debugger like ISE.

However, parallel HPC code cannot always be compiled and executed on any host, due to the links to MPI libraries, but a non-MPI test harness is usually an acceptable way to start the design of the FPGA code. Of course, the debugging environment also needs, at some point, to be able to target directly the HPC environment, as specific bugs may appear at that stage only. This is achieved by the Tool Flow System after configuration of scripts to handle the transfers between the host and the HPC system.

Scripts can include the ability to access HPC systems behind gateways and submit jobs in queues.

3 Conclusions and Further Work

The GALAXY framework, originally designed for GALS embedded system design, happens to be mature for the design of accelerated HPC applications. It allows HPC programmers to apply an iterative strategy for the use of HPC FPGA boards. They can design and debug their whole design on a non-HPC host, by linking the software code to Verilog or VHDL simulators, and then remotely target the real HPC system while keeping debugging feedback in the IDE. Components can be moved one by one from software abstractions to hardware with the new interfaces being automatically regenerated.

The generated asynchronous interfaces happen to be efficient enough for a first version of the user's HPC application. They allow the user to experiment with several architectures before optimizing the critical paths, and appear to be an efficient way to proceed.

Although no benchmark is available yet, the GALAXY framework presented in this paper has been augmented for HPC by using a Cray XD1 with Xilinx Virtex 4 FPGAs. The Xilinx tool flow has been integrated in the tools, and we are now working on a demonstrator.

Acknowledgements

This project is funded by the European Seventh Framework Programme (FP7).

We would like to express our thanks to the CCLRC Daresbury Laboratory and their great team for providing access to their Cray XD1.

References

1. Wain, R., Bush, I., Guest, M., Deegan, M., Kozin, I., Kitchen, C.: An overview of FPGAs and FPGA programming; Initial experiences at Daresbury. Computational Science and Engineering Department, CCLRC Daresbury Laboratory (2006)
2. Mohl, S.: The Mitrion-C programming language. Mitrionics Inc. (2006), http://www.mitrionics.com/
3. Celoxica Ltd., http://www.celoxica.com
4. Nallatech Dime-C.: http://www.nallatch.com
5. http://www.galaxy-project.org

Architecture-Aware Custom Instruction Generation for Reconfigurable Processors

Alok Prakash, Siew-Kei Lam, Amit Kumar Singh,
and Thambipillai Srikanthan (Senior Member IEEE)

Center for High Performance Embedded Systems
School of Computer Engineering
Nanyang Technological University
{alok0001,assklam,amit0011,astsrikan}@ntu.edu.sg

Abstract. Instruction set extension is becoming extremely popular for meeting the tight design constraints in embedded systems. This mechanism is now widely supported by commercially available FPGA (Field-Programmable Gate Array) based reconfigurable processors. In this paper, we present a design flow that automatically enumerates and selects custom instructions from an application DFG (Data-Flow Graph) in an architecture-aware manner. Unlike previously reported methods, the proposed enumeration approach identifies custom instruction patterns that can be mapped onto the target FPGA in a predictable manner. Our investigation shows that using this strategy the selection process can make a more informed decision for selecting a set of custom instructions that will lead to higher performance at lower cost. Experimental results based on six applications from a widely-used benchmark suite show that the proposed design flow can achieve significantly higher performance gain when compared to conventional design approaches.

1 Introduction

The rapidly increasing time-to-market pressure and demand for higher performance is consistently pushing FPGA-based systems to the forefront of embedded computing. One of most popular ways of using this FPGA space, is by adding Custom Instructions to the system. Although a lot of research has been done in the area of custom instruction generation, commercial FPGA tools still lack a consolidated framework for automatic implementation of custom instructions for a given application code. There are typically two major steps in custom instruction implementation namely, custom instruction (pattern) *identification* and custom instruction (pattern) *selection*. A commonly adopted approach in pattern identification is pattern enumeration, which tries to identify all the legal custom instruction patterns within an application. Pattern selection then selects a set of non-overlapping patterns for final implementation based on certain constraints such as area and performance. In certain design flows, a design exploration step, which takes into consideration the area required by each pattern, is undertaken to propose a set of custom instructions for a given performance area constraint.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 414–419, 2010.

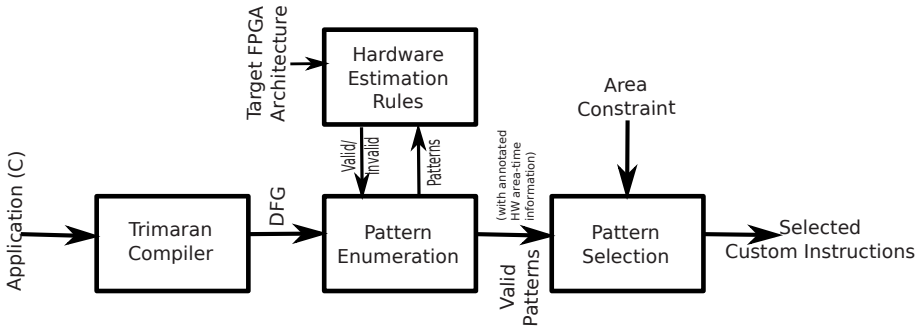


Fig. 1. Proposed Instruction Set Extension Methodology

There has been a number of previous work in the area of pattern identification and selection. Atasu et al. in [1] described a branch and bound algorithm that enumerates all the legal patterns from an application DFG, while pruning off the ones, which violate the fundamental micro-architectural constraints. Their method produced significantly better performance when compared to the state of the art techniques at that time. However, the complexity of the branch and bound algorithm in their work increases rapidly with the size of the DFG.

Clark et al. presented a complete design framework for automatic identification of pattern candidates within a DFG and a compiler framework to take advantage of such custom units [2]. In their selection stage, a greedy heuristic was used to select the candidate with largest (speedup/area) ratio before selecting any other patterns.

Chen et. al. presented a novel algorithm for rapid identification of custom instructions for extensible processors which was especially useful for large DFGs [3]. Li et. al. [4] proposed further enhancements in Chen's algorithm and thus achieving up to 50% faster tool runtime for enumerating patterns with single-output constraint.

Lam et. al. [5] presented a custom instruction generation design flow that incorporates a novel way for estimating the area of custom instructions in the final hardware by *clustering* the candidate patterns [5].

All the enumeration algorithms described above take the micro-architectural properties of the underlying hardware as a fundamental set of constraints, for example the number of input or output ports available for custom instruction implementation. Although these constraints must be used during enumeration, we propose a custom instruction generation strategy that incorporates the target architecture information in the pattern enumeration and selection phases as shown in Fig. 1.

In particular, the proposed pattern enumeration approach generates only patterns that can be fully mapped onto the logic elements of the target FPGA architecture. As the hardware area-time of the custom instruction patterns from the enumeration phase are implied, effective pattern selection can be performed to choose a set of custom instructions that can lead to higher performance at lower

cost when compared to conventional design approaches that focus on selecting large and recurring custom instructions.

The rest of the paper is organized as follows. Section 2 discusses the proposed pattern enumeration algorithm, while section 3 explains the pattern selection phase. Section 4 deals with the experiments and results. Section 5 concludes this paper.

2 Pattern Enumeration

Pattern enumeration is defined as the process of identifying all the legal patterns (subgraphs) within a DFG. The legal patterns of an application must obey a set of constraints that is compatible with the micro-architectural constraints of the underlying architecture [1], [3]. These micro-architectural constraints are used to prune away the illegal patterns. In addition, memory and branch operations are generally not allowed in a custom instruction.

The proposed pattern enumeration technique in this paper extends the method in [3] and [4], which have been shown to be orders of magnitude faster than other existing algorithms for large DFGs. In particular, we have incorporated additional constraints, which ensures that the enumerated patterns can be mapped onto the target FPGA in a predictable manner. In the following sub section we will describe the proposed architecture-aware pattern enumeration method.

2.1 Architecture-Aware Pattern Enumeration

In order to perform architecture-aware enumeration, we have introduced additional architecture specific constraints during pattern enumeration phase to limit the number of enumerated patterns to those that would lead to efficient mapping on the target FPGA and achieve high performance. Similar to [5], we have used Trimaran to compile the C Code of the benchmark application. The output of the Trimaran compiler is called a "Rebel" file which is used as the input to our enumeration stage. The "Rebel" file obtained from Trimaran gives the application representation using the basic operations of the Trimaran instruction set architecture. We mentioned before that we have used the enumeration algorithm from [3] and [4] in our work. This enumeration algorithm works in the following manner: It searches for valid patterns, starting with an empty pattern P, the convex DFG G of a basic block and a redundancy guarding node referred to as r_g. The algorithm recursively invokes the enumeration procedure, which consists of three functions, *select_node*, *unite* and *split*. Function *select_node* returns a selected node from the remaining node set *(G-P)*. The function *unite* handles the condition of addition of valid nodes into the pattern to create a new pattern, while the *split* function handles the situation where the selected node cannot be added to the pattern. The function *unite* can merge multiple nodes if the node selected by the function *select_node* is of high quality. It is in this *unite* step, wherein we can check the extra constraints for our purpose during the merging of a selected node with the existing pattern. Meanwhile, the function *split*

can decompose the current DFG *G(V,E)* into one or two sub-graphs by splitting them when necessary, thus reducing the depth of recursive search. Using this methodology the output of the *unite* phase always produces a valid pattern which is ready for the selection stage. The details of the algorithm can be found in [3] and [4].

Lam et. al. in [5] showed a novel method for estimating the area of a given pattern. They applied various architecture specific constraints to ensure that a candidate pattern can be fully implemented within a logic group (a set of FPGA logic elements with the same hardware configuration) on the Xilinx Virtex 4 FPGA. In particular, after the pattern selection phase, they partitioned each of the selected pattern into clusters, wherein each cluster could be implemented in one logic group. This enabled them to estimate the total number of logic groups required to implement the selected patterns on the target architecture. A detail description of the constraints used by them for *Clustering* can be found in [5]. We incorporated these rules along with the previously mentioned micro-architecture constraints, during our enumeration phase in order to enumerate only "clusters". This ensured that all the patterns we enumerated could be implemented in the logic blocks of the FPGA. It is noteworthy that although we have demonstrated the proposed approach for the Virtex 4 FPGA, our idea is generic enough to be used with other FPGA families.

Once the valid clusters are enumerated, we begin the selection phase. In this phase, we select the most profitable patterns to be implemented as custom instructions. In the next section we will explain the pattern selection step in detail.

3 Pattern Selection

After enumerating all the possible legal patterns from the enumeration phase, we perform graph-subgraph isomorphism using "VFLib" to find the similar patterns and group them together. Each pattern group is defined as a template, whereby each template corresponds to a set of similar patterns. In the next step, pattern selection is performed to find a set of non-overlapping patterns for final implementation.

We have used the approach described in [6] for pattern selection. A conflict graph is created based on the enumerated patterns.The objective of creating this conflict graph is to facilitate choosing a set of non-overlapping patterns for final implementation.

Once the conflict graph is created, we use the steps described in [5] to compute the local Maximum Independent Set (MIS) in order to obtain a set of non-overlapping patterns within each template. This aims to find the largest set of vertices in every template of the conflict graph that are mutually non-adjacent. The process of computing the local MIS relies on a similar heuristic used in [6] and [7], which aims to select large and recurring custom instructions. The metric for this heuristic is calculated based on the product of the number of nodes in a template and the number of corresponding patterns in the DFG.

After the local MIS is calculated, the templates are arranged in decreasing order of their MIS weight. In order to perform constraint-aware pattern

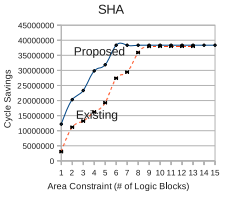
selection, the template with the largest MIS weight is evaluated first. It is selected for implementation if the area constraint is not violated. If this template violates the area constraint, we move on to evaluate the template with the next largest MIS weight and so on, unless a suitable template is found and selected. Once a template is selected, the corresponding patterns of the template are implemented as custom instructions. These selected patterns are then removed from the conflict graph, and the remaining area for custom instruction implementation is updated. The algorithm repeats to find a new set of local MIS for each template. The algorithm terminates when either the conflict graph becomes empty or the area constraint is violated.

4 Experiments and Results

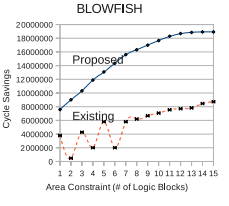
In this section, we compare the results obtained using our proposed architecture-aware custom instruction design flow with a conventional approach that aims to select large and recurring custom instructions without considering how they can be mapped onto the target FPGA. The conventional methodology has been implemented using the methods described in [3], [4] for enumeration, [5], [6] for selection and [8] for design space exploration.

The cycle savings after using the extended instruction set has been calculated using the following formula [5] for both the conventional and proposed methods:

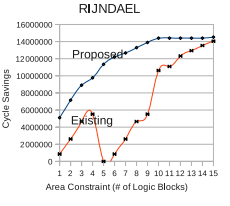
$$cycle\ savings = [\sum_1^n Num\_of\_Nodes * Dynamic\_Occurance] - [\sum_1^n Critical\_LUT\_Path * Dynamic\_Occurance] \quad (1)$$



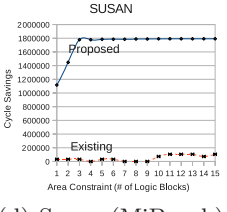
(a) Sha (MiBench)



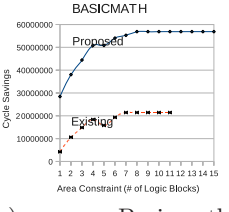
(b) Blowfish (MiBench)



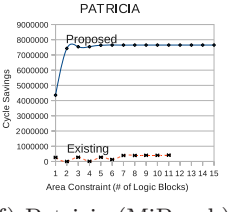
(c) Rijndael (MiBench)



(d) Susan (MiBench)



(e) Basicmath (MiBench)



(f) Patricia (MiBench)

Fig. 2. Performance vs. Area Curves for Benchmark Applications

The first term in the above equation 1 represents the software execution time of a custom instruction whereas the second term depicts the time in number of clock cycles if the custom instruction is executed in the hardware.

The graphs in Fig. 2 clearly show the advantage of the proposed architecture-aware custom instruction generation process. This methodology is especially useful for Area-Constrained FPGA designs.

5 Conclusion

In this paper, we proposed a design flow for automated custom instruction generation that takes into account the target FPGA architecture in both the custom instruction enumeration and selection phase. In particular, the proposed enumeration approach incorporates a set of rules that identifies custom instruction patterns which can be fully mapped onto the logic elements of the FPGA architecture. As the hardware area-time of the candidates are available after the enumeration process, the selection process can effectively choose a set of custom instructions that lead to high performance at low cost. Experimental results based on six applications from the MiBench benchmark suite show significant performance improvement of up to 5 orders of magnitude over an existing approach.

References

[1] Atasu, K., Pozzi, L., Ienne, P.: Automatic application-specific instruction-set extensions under microarchitectural constraints. In: DAC 2003: Proceedings of the 40th annual Design Automation Conference, pp. 256–261 (2003)

[2] Clark, N., Zhong, H., Mahlke, S.: Automated custom instruction generation for domain-specific processor acceleration. IEEE Trans. on Computers 54(10), 1258–1270 (2005)

[3] Chen, X., Maskell, D.L., Sun, Y.: Fast identification of custom instructions for extensible processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26(2), 359–368 (2007)

[4] Li, T., Jigang, W., Deng, Y., Srikanthan, T., Lu, X.: Fast identification algorithm for application-specific instruction-set extensions. In: International Conference on Electronic Design, ICED 2008, pp. 1–5 (December 2008)

[5] Lam, S.K., Srikanthan, T.: Rapid design of area-efficient custom instructions for reconfigurable embedded processing. J. Syst. Archit. 55(1), 1–14 (2009)

[6] Guo, Y., Smit, G.J., Broersma, H., Heysters, P.M.: A graph covering algorithm for a coarse grain reconfigurable system. In: LCTES 2003: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, pp. 199–208. ACM, New York (2003)

[7] Bonzini, P., Pozzi, L.: Recurrence-aware instruction set selection for extensible embedded processors. IEEE Trans. VLSI Syst. 16(10), 1259–1267 (2008)

[8] Cong, J., Fan, Y., Han, G., Zhang, Z.: Application-specific instruction generation for configurable processor architectures. In: FPGA 2004, pp. 183–189. ACM, New York (2004)

Cost and Performance Evaluation of a Noise Filter for Partitioning in Co-design Methodologies

Victoria Rodellar, Elvira Martínez de Icaya, Francisco Díaz, and Virginia Peinado

Grupo de Investigación Aplicada a la Señal e Imagen (GIAPSI)
Facultad de Informática, Universidad Politécnica de Madrid,
Campus de Montegancedo s/n, Boadilla del Monte, 28660, Madrid-Spain
victoria@pino.datsi.fi.upm.es, {emicaya,vpeinado,fdiaz}@eui.upm.es

Abstract. We have studied a Noise Canceller filter to estimate its performance parameters of speed, area and power consumption in different implementations. They have been oriented to find solutions for two kinds of applications: low-power or high speed. The results may be used as inputs to a partitioning algorithm in Co-design methodologies. The algorithm for filter implementation presents a big dispersion in the upper and lower bounds of the variables. The performance dependence by using the same data-format size for all the variables and multiple data-format size adjusted ad hoc for each one of them was evaluated. The results from the Virtex-4 and Stratix II families and TMS320C5510 and TMS320C6416 microprocessors are presented and discussed.

Keywords: Cost Evaluation, Performance Evaluation, HW/SW Co-design, Metrics, Data format Optimization, Software Profiling, FPGAs implementation, Noise cancellation.

1 Introduction

In Hardware/Software Co-design Methodologies the space design of a given problem is explored in order to find a good solution accordingly to its performance/cost restrictions. The partition stage guides the designer in which parts should be implemented in hardware or in software or automatically splits the problem into them. Representative examples of available tools and design environments as Ptolemy [1], Polis/Metropolis [2], CoWare [3], etc, restrict the design exploration to predefined library-based components. These components are characterized by a given metric and fed into the partition algorithms. In this paper we will focus on the evaluation of an Adaptive Noise Canceller Filter for different implementations in commercial DSP microprocessors and FPGA's which may be included in the predefined component library to be used in the design space exploration. The Adaptive Noise Canceller studied here, can be used for Speech Enhancement Interfaces in adverse environments with high levels of noise.

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 420–425, 2010.

2 Filter Description

The recording scheme and computational structure of the filter can be found in [4]. It shows a very good behavior in highly non-stationary environments having a competitive performance compared which other filters [5]. The algorithm demands 9 additions, 15 multiplications and 6 divisions per stage. Then, the number of operations involved per sample will be 126 additions, 210 multiplications and 84 divisions, considering 14 stages filter. The algorithm is computationally costly specially taking into consideration that most of the DSP microprocessors and all FPGA's lack of the division operation implemented by hardware. This fact is an obvious disadvantage but the algorithm presents the advantage of requiring very small arrays of memory for the speech input data instead of large buffers because the processing of the speech samples can be done as soon as they become available. Nevertheless, its high computational cost makes the problem suitable to explore different alternatives of implementation. The filter design space exploration requires taking into consideration its implementation in FPGA's. High level synthesis tools have the limitation of using only integer data types, which has important implications on the accuracy. There are tools which convert from floating-point to fixed-point formats [6] and optimize word-length [7]. These tools are very helpful but the results are not good enough when accuracy is critical, as is the case of the algorithm under study. Then word-length estimation was carried out in [4]. A large dispersion on the variable bounds was observed. It was found that some of the parameters have values below one and other exceed the range of representation for integer numbers for 32 bits. Then, the optimization of word-length for the different variables was carried out, which is NP-hard [8]. An exhaustive simulation study was done to determine the minimum number of bits to represent each variable having in mind the accuracy and the quality of spoken-commands intelligibility after filtering. The criterion to validate the results consisted in estimating the errors between the results in floating-point arithmetic and different integer-data formats using a scale factor. The longest word-length was obtained for the forward and backward residual errors (40 bits), the shortest word-length was 16 bits to estimate the noise and clean signals and the word-lengths for the rest of the variables was included among those extreme values [4].

3 Platforms, Devices and Tools

Having in mind embedded applications, it was decided to estimate the problem for two extreme solutions, that is, for applications with performance restrictions in speed or in power consumption. Fixed-point processors as TMS320C5510 oriented to low-power applications and TMS320C6416 oriented to high performance applications were chosen for the evaluations. Concerning the election of FPGA platforms the Stratix II and Virtex-4 families recommended for DSP applications have been considered. The estimations will be done with the pairs EP2S30F484C-4 and 4VLX25sf363:11, and EP2S15F484C-4 and 4VLX15sf363:11. The granularity chosen to study the problem was at the level of algorithm and ANSI-C

modeling was used. The software synthesis to assembler was carried out by means of the Code Composer Studio from Texas Instruments. The transformation from C-code (ImpulseC) to RTL-VHDL code was done by Co-Developer from Impulse Technologies. The hardware synthesis was carried out by Quartus II from Altera and ISE from Xilinx. PowerPlay from Altera and XPower from Xilinx were the tools for power estimation. Another fact that affects the quality of the results is the design tool settings; in our case default values have been used.

4 Results

The estimation results have been calculated under two different points of view. First, all the variables of the algorithm have been coded with the number of bits of the most costly variable; then the format is fixed to 40 bits for all of them. And second, the variables are coded in multiple formats adapted to their optimal word-length. The arithmetics implemented has been fixed-point and two's complement in both cases. The operations of multiplication and division have been also programmed in C. The reason is mainly due to the use of the Co-Developer tool which limit the data format for fixed-point preprogrammed operations to 32 bits size. The multiplication algorithm developed is the classical Booth algorithm [9] and the division is implemented as the Goldschmidt algorithm [10] based on the calculation of successive multiplications.

4.1 Metrics

The metrics is the key point to make the evaluation results useful when fed in a partitioning algorithm. The algorithm should be characterized by its estimation of area, time and power parameters in hardware and software implementations. Each parameter must summarize in single number significant information (like its fingerprints) of that particular solution implementation in order to make it comparable with other possible predefined library components and also to assure its portability among different Co-design environments to facilitate its reusability. Then, two main questions arise. How to make the results comparable among microprocessors or FPGA's which have architectures too different? What are the units better describing the parameter quantization? The area of software (a_s) will be evaluated by the number of memory bits occupied by the program code and data, assuming that the demand of space in memory for each instruction is different in the two considered microprocessors. Meanwhile the hardware area quantification (a_h) will be based on the logic structure of the FPGA's, Adaptive Logic Modules (ALM) for Altera and Slices for Xilinx. The units will be expressed as the number of LUT of four inputs. The factor to convert the estimated resources from one to other architecture has been taken from the information provided by Altera [11], no other source of information about this conversion has been found in the literature for time being. These consider the equivalence of one ALM from StratixII matching to 2,6 times 4 LUT's and one Slice from Virtex-4 equaling to 2 times 4LUT's. The hardware time (t_h) will be

expressed as the number of samples processed per second (s/s). The hardware power consumption (p_h) results will be given in terms of the dynamic power (mW/MHz) only. The results presented were obtained for an 80% signal toggle rate. The software power (p_s) estimation is calculated with the spreadsheet provided by Texas Instruments for its DSP processors taking into consideration CPU activity and the current and the voltage of operation.

4.2 Estimation

Four different cases were evaluated in the option of 40 bits fixed format. The main difference among them resides in the way of implementing the algorithm and the resource restrictions for the variables when mapping them on the FPGA. In Case I, all the variables are stored in registers only and the devices used were EP2S30F484C-4 and 4VLX25sf363:11, meanwhile for the rest of the evaluated cases, the implementation was done on the devices EP2S15F484C-4 and 4VLX15sf363:11. In Case II and Case III, implementations on single-port and dual-port RAM memories respectively were considered. And finally Case IV, was the least restrictive of them all, allowing the use of dual-port RAM memories and internal DSP block multipliers from the FPGA's. The results are summarized in Table 1 (1) for Altera and (2) for Xilinx. Obviously, the performance obtained for the different cases is in agreement to resource restrictions. Generally speaking, Case IV may be considered the best, Case I the worse and Case II and Case III solutions between them. Analyzing these two last cases in detail, we may observe in dual-port RAM memory realizations that the speed decreases for both Altera and Xilinx, the area and power increases in Altera's and they maintain around the same values in Xilinx's.

Table 1. Hardware estimation results for 40 bits fixed format

	Speed (1) (s/s)	Area (1) (4LUT's)	Power(1) (mW/Mhz)	Speed (2) (s/s)	Area (2) (4LUT's)	Power (2) (mW/Mhz)
Case I	2769	28584	1.25	2665	21000	2.18
Case II	2741	15839	1.78	2676	11312	1.73
Case III	2494	16203	2.22	2660	11300	1.72
Case IV	103842	7680	1.7	92575	8280	1.2

These results evidence the computational structure of the algorithm being different from other kind of filters or FFT algorithms where dual-port memories structures are much appropriated for these implementations. Then, it may be concluded that the better performance is achieved for Case IV in Altera's implementation. The word-length adjusted ad hoc after optimization for each variable is studied next. The implementation was done with dual-port RAM memory and DSP block multipliers to make the results comparable with Case IV. These results show an important improvement in area and power consumption but they are worse in speed when compared with fixed format Case IV in both Altera and

Table 2. Hardware estimation for word-length adjusted ad hoc

	Speed (s/s)	Area (4LUT's)	Power (mW/Mhz)
Altera	50342	4430	0.93
Xilinx	47808	3818	0.62

Table 3. Software estimation results with 40 bits format and optimal number of bits for the variables

Microprocessor	Format	Execution time (t_s) (s/s)	Area (a_s) (bits)	Power (p_s) (mW/MHz)
TMS320C5510	40 bits	36.36	156144	0.92
	Optimal	43.48	153416	0.91
TMS320C6416	40 bits	158.73	286272	1.54
	Optimal	196.07	288000	1.52

Xilinx realizations. The poor behavior of this last parameter is due to the delay suffered by the variables caused by size adaptation to internal DSP multipliers in the FPGA's. It may be concluded that the best choice for applications with speed restrictions is the Altera implementation in 40 bits-data format, meanwhile if the restriction is power consumption the data format should be implemented ad hoc in Xilinx FPGA's, in both cases using dual port RAM memories and DSP block multipliers.

A similar study carried out for hardware estimation has been done for software estimation. First the algorithm considering 40 bits length has been implemented, later on adjusting ad hoc the number of bits for each variable. The results are shown in Table 3.

They are in coherence with the characteristics of the microprocessors. Obviously, the 5510 microprocessor presents better results for low-power and the 6416 for high speed applications. Which is interesting to analyze here is the impact of the ad hoc data formats in the parameters estimation. It may be observed that the execution time improves in 16,4% for the 5510 microprocessor and in 19,1% for the 6416. The ad hoc variables format has different effects in the resulting area for the platforms studied. The 5510 experiments a decrement and an increment is observed in the 6416. This is due to the differences of assembler programming. The memory positions for the program instructions are fixed to two bytes in the 6416, which make it faster. Meanwhile each instruction in the other microprocessor occupies different number of bits, being more suitable for the ad hoc format. In both platforms there is a slight reduction in the power consumption when ad hoc formats are considered. As a conclusion, it may be said that the ad hoc format improves speed and power consumption but the format influence in software area results depends on how many bytes each instruction in memory occupies.

5 Conclusions

A digital filter for noise cancellation has been described and its performance/cost parameters have been estimated from different implementations in hardware and software. The objective of these implementation characterizations is to use the resulting parameters as inputs in a partition algorithm in Co-design Methodologies. The first decision which had to be adopted was what platforms, devices and design tools to use. The first problem faced was to define a metric representative enough to make the comparisons possible among elements having different architectures making comparisons very difficult in a precise manner. The solution implemented with multiple data format sizes improves the results of time and power obtained with single data format sizes but the behavior of the area parameter depends on the microprocessor architecture. Moreover, improvements are attained in the parameters of area and power and losses of about a half in speed in hardware implementations.

Acknowledgments. This work was supported by grants TEC 2006-12887-C02-00 from Plan Nacional de I+D, Ministry of Education and Science and by Project HESPERIA (http://www.proyecto-hesperia.org) from the Program CENIT, Ministry of Industry, Spain.

References

1. Ptolemy Environment, `http://ptolemy.eecs.berkeley.edu/`
2. Metropolis Environment, `http://www.gigascale.org/metropolis/index.html`
3. CoWare Platform, `http://www.coware.com/products/platformarchitect.php`
4. Rodellar, V., Alvarez, A., Martínez, E., Gonzalez, C., Gómez, P.: FPGA Implementation of an Adaptive Noise Canceller for Robust Speech Enhancement Interfaces. In: 4th Southern Conference on Programmable Logic (SPL 2008), pp. 13–18 (2008)
5. Haykin, S.: Adaptive Filter Theory. Prentice Hall, New York (1996)
6. Hill, T.: AccelDSP Synthesis Tool Floating-Point to Fixed-Point Conversion of MATLAB Algorithm Targeting FPGAs. Xilinx Whitepaper, WP239-V1.0 (April 2006)
7. Chang, M.L., Hauck, S.: Precis: A Usercentric Word-Length Optimization Tool. IEEE Design and Test of Computers, 349–361 (July-August 2005)
8. Constantinides, G.A., Woeginger, G.J.: The complexity of multiple word length assignment. Applied Mathematics Letters 15(2), 137–140 (2002)
9. Omo, R.: Computer Arithmetic System: Algorithms, Architectures and Implementations. Prentice Hall, Englewood Cliffs (1999)
10. Kilts, S.: Advanced FPGA design: Architecture, implementation and optimization. Wiley and Sons, New Jersey (2007)
11. Stratix II vs. Virtex-4, `http://www.altera.co.jp/literature/po/ss-s2density.pdf`

Towards a Tighter Integration of Generated and Custom-Made Hardware

Harald Devos, Wim Meeus, and Dirk Stroobandt

Hardware and Embedded Systems Group, ELIS Dept., Ghent University,
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{harald.devos,wim.meeus,dirk.stroobandt}@elis.UGent.be

Abstract. Most of today's high-level synthesis tools offer a fixed set of interfaces to communicate with the outer world. A direct integration of custom IP in the datapath would often be more beneficial than an integration using such communication interfaces. If a certain interface protocol is not offered by the tool, either translation blocks (wrappers) are needed or the code should be written at a lower level. The former solution may hurt the performance, while the latter one is often impossible using an untimed high-level description.

In this paper interface protocols or sets of IP core accesses are first described at a low level as sets of operations with scheduling information (macros). During the synthesis process, corresponding function calls are mapped to these macros. This facilitates the integration of custom-made hardware and hardware generated by high-level synthesis tools.

1 Introduction

The increasing number of transistors that can be put on a single chip, has led to a growing design gap. Therefore, hardware design has to be lifted to a higher level, as exemplified by the growing interest in high-level synthesis (HLS) techniques [5]. However, hardware designers like to have full control over critical design parts, which hinders the adoption of current HLS environments.

In the software world high-level languages have become mainstream since long. However, for critical pieces of code, manually written assembler code may still outperform compiled high-level code. *In-line assembler* allows to exploit the full power of certain processor instructions while still enjoying the benefits of high-level languages for the less critical program code. It would be very interesting to have a similar option for hardware design. The limitations of current HLS tools will be more tolerable if the designer can take over control for those parts of a design where the synthesis results are not satisfactory.

Many HLS tools offer a built-in library of macros to communicate with the *outer world*. The inclusion of a custom-made block should typically be specified as communication with an *external block*, that has to fit a certain interface. A translator block, called a wrapper, may be used to translate one interface into another (Fig. 1), but this may create an overhead (see Sect. 4). Describing the communication at a lower level, e.g., using multiple statements in the high-level

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 426–434, 2010.

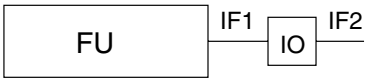


Fig. 1. An IO hardware block translates the interface of the functional unit (IF1) to the desired interface (IF2)

input language to describe one external transaction, may often be impossible since the concurrency of signal assignments cannot be forced in untimed C code.

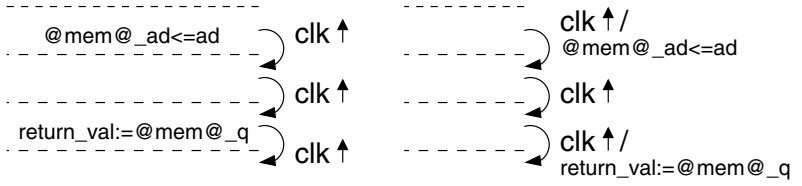
This paper presents a method to describe interface protocols as a sequence of schedule[1] steps with transition information (Sect. 2), such that it can be dealt with on equal terms with the built-in macros or interfaces of the HLS tool. Macros may also be used to interact with custom-made IP blocks in a more integrated way. Our scheduler is able to pipeline transactions and operations without specific directives (Sect. 3). Hereby, we still have the advantages of a modular design flow: portability and a clear separation for the designer of communication and functional behavior, but with less overhead thanks to the tighter integration of these two in the generated hardware.

2 Macro Schedule Description

A macro is a set of operations described at a cycle- and pin-accurate level that together with the external hardware it communicates with implements a functionality corresponding to one high-level construct, e.g., a memory access, a mathematical operation performed by an IP core, ... (In this section we restrict ourselves to well known memory examples. Real IP-cores are used in Sect. 4.2). Such a macro can be described as a finite state machine (Fig. 2 and Fig. 3). The area between two dashed lines corresponds to one state. Transitions, indicated with arrows, are triggered by events (clk ↑), possibly guarded by conditions put between `[ ]`. Operations that are executed at a state transition are put behind a '`/`'. State charts only execute operations at state transitions. We use a more compact notation where operations inside a state represent operations that are executed at each outgoing transition as explicitly shown in Fig. 2(a and b). For each macro a corresponding C function is written. During synthesis, calls to these functions are mapped to the corresponding macros.

From this information, our scheduler derives which signals are used by a macro instance in which cycles (Fig. 4(a and b)). It is able to overlap several instances of one macro (Fig. 4(c)), but also instances of different macros sharing ports (Fig. 4(d)). If in a clock cycle a macro does not apply a value to an output, that output can be used by another macro execution. Else, it receives a default safe value (Fig. 2(c)). That these default values are not forced by macros that do not use a certain output is essential to the ability of automatically pipelining macro instances.

[1] Scheduling in HLS is the task of assigning operations to clock cycles (or control steps) in a static way, i.e. in the hardware generation process.



(a) (b) Alternative representation of (a)

```
defaults (
  @mem@_ad <= (others => '-');
  @mem@_d <= (others => '-');
  @mem@_we <= '0';
  )
```

(c) Default values of outputs

```
schedule:
  @mem@_ad <= ad;
  <nc>
  <nc>
  return_val := @mem@_q;
  <end>
```

(d) Textual schedule representation

Fig. 2. Protocol description for a single cycle memory read. The `@mem@` substring is a placeholder for the name of the memory instance. <nc>= new cycle.

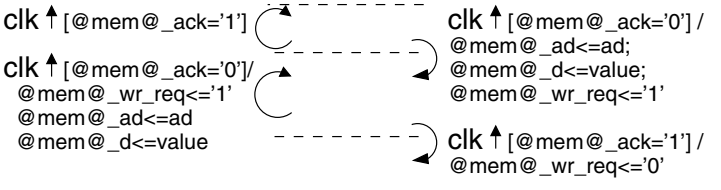


Fig. 3. Protocol description for a memory write with a full handshake

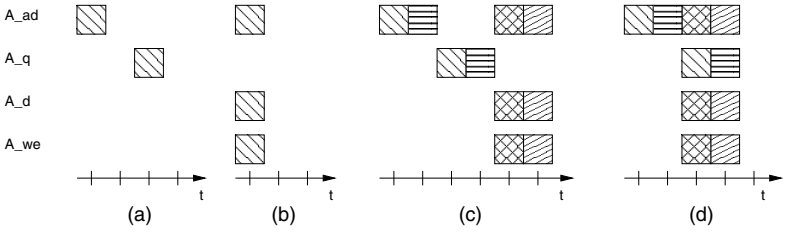
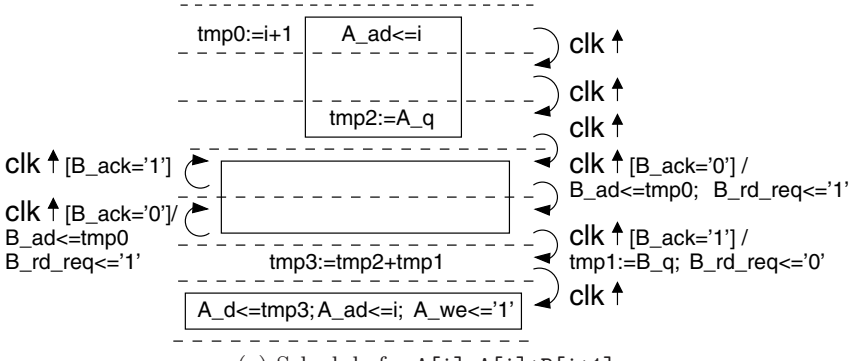


Fig. 4. Occupation of ports for single-cycle memory transactions (a) read, (b) write. Both reads and writes can be pipelined with initiation interval 1 (c), but also a mix of reads and writes (d).

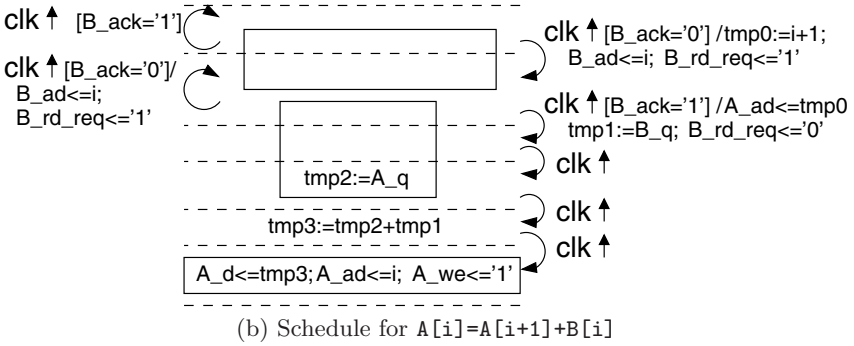
Conditional state transitions (Fig. 3) result in an indeterministic execution time. Our scheduler is able to combine macros (and built-in constructs) with both deterministic and non-deterministic execution time (Sect. 3).

3 The Scheduling Algorithm

Scheduling operations with fixed time intervals can be done using behavioral templates [11]. A behavioral template is a set of nodes of a control data flow graph (CDFG) coupled with cycle offsets. If one of the nodes of the template is scheduled, the schedules of all the other nodes are defined by the relative offsets. Behavioral templates for a single cycle read and write are recognized in the first and last cycle of the schedule in Fig. 5(a). We extended the concept of behavioral templates with conditional state transitions that introduce an indeterministic execution time (E.g., handshake read of array `B` in Fig. 5.). A conditional state transition cannot be scheduled in parallel with a template that has a fixed cycle offset, unless some kind of micro-threading is used (which is



(a) Schedule for `A[i]=A[i]+B[i+1]`



(b) Schedule for `A[i]=A[i+1]+B[i]`

Fig. 5. Behavioral templates (boxes) take care of the relative schedule constraints (offsets) of different operations of a macro during the scheduling process

```
while nodes remain to be scheduled {
    ready list := all nodes that do not depend on unscheduled nodes
    for each node in the ready list {
        find earliest execution cycle of this node based on dependencies
        while conflicting resource constraints
            increase execution cycle of this node
    schedule this node at the found execution cycle
        update consumed resources
    }
}
```

Fig. 6. The scheduling algorithm. A *node* is a behavioral template consisting of several CDFG operations, or a single operation that is not included in any behavioral template (could be regarded as a singleton behavioral template).

kept as future work). This is regarded as a *conflicting resource constraint*. However, single cycle operations can be made on the conditional state transitions that are taken exactly once, e.g., `tmp0:=i+1` in Fig. 5(b).

Our scheduling algorithm (Fig. 6) schedules operations as soon as possible. If two operations are ready to be scheduled but have conflicting constraints the one that comes first in the input C code is selected. In future work we will explore more advanced scheduling heuristics. The outer loop does not iterate over cycles, but over the nodes to be scheduled. This differs from classical algorithms. The nodes in the algorithm are behavioral templates that may consist out of multiple operations scheduled together or operations not being part of a behavioral template, regarded as singleton behavioral templates (that can be combined with conditional transitions).

4 Experiments

Our high-level synthesis environment, JCCI [6] (Java C to CLooG [3] Interface), reads in macro descriptions (in a textual representation, cf. Fig. 2(d)). Function calls in the C input code that correspond to macros are recognized and replaced with the scheduling information of the corresponding macros.

4.1 Memory and FIFO Interfaces for a Sobel Edge Detector

The kernel of the Sobel edge detector [8] algorithm contains a 3×3 sliding window operation. Five conceptually different designs were described in C.

1: The standard textbook algorithm. 2: Variables store the input values that are reused in the next two column iterations. This reduces the number of memory reads with a factor three. 3: Vertical reuses are exploited by storing two lines of the image in FIFO buffers. This reduces the memory reads with roughly another factor three. The FIFO buffers are created as IP cores (Altera MegaWizard) and

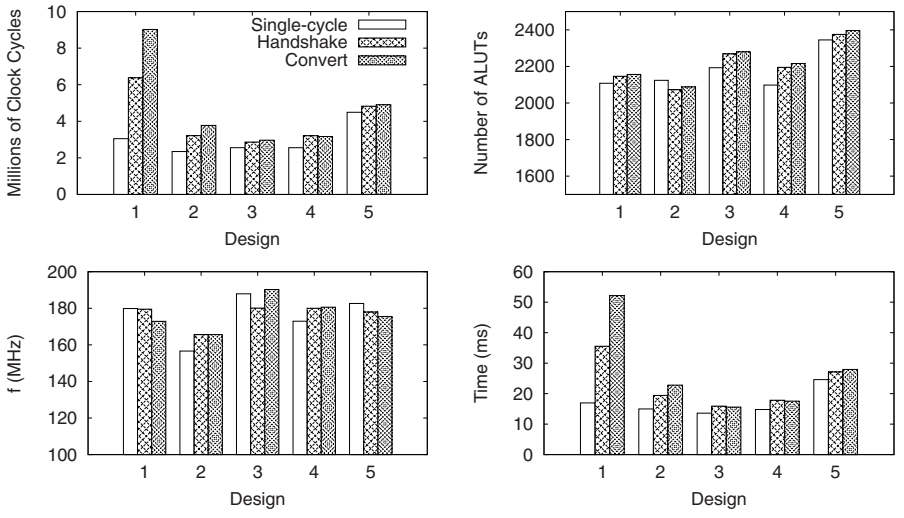


Fig. 7. Simulation and synthesis results for 5 implementations of a Sobel edge detector on an Altera Stratix III (EP3SL150F1152C2) for different interfaces. ALUT = Adaptive Look-Up-Table. Execution time and number of clock cycles for a 320×320 input image. The number of DSP and memory blocks used are not plotted since they do not depend on the IO interface.

accessed using macros. Designs 2 and 3 are similar to the designs listed in [7,12]. 4: similar to 3, but the FIFO is made using single port memory blocks, with a macro that implements the address increments with wrap around. 5: no FIFO macros used. A C description of FIFOs is used to buffer the two lines of pixels.

For each of these designs we generated three variants with a different memory interface (Fig. 7). A first variant uses a single-cycle memory (cf. Fig. 2). A second one uses a memory with a full handshake protocol (cf. Fig. 3). A third variant is identical to the second extended with a block that converts the full handshake protocol into a single-cycle protocol, similar to Fig. 1 (with IF1 the handshaking and IF2 the single-cycle protocol). This mimics the situation where a protocol is not part of the HLS tool's library and a wrapper is inserted.

Results and Discussion. The interface conversion blocks introduce (on the average) an overhead in area and clock cycles (*Convert* vs. *Single-cycle* in Fig. 7). In some cases a wrapper augments the clock frequency, but this is counterbalanced by the increased number of cycles. It is thus beneficial to include the proper protocol directly. The usage of macros to describe the FIFOs gives better results than the equivalent description in C code. Probably, other HLS tools may generate line buffers in a much more efficient way. However, the point we want to make here is that the user can take control over the synthesis process when the HLS techniques do not give satisfactory results.

4.2 Integration of Floating-Point Megacores

The Altera Floating Point Megafunctions [1] implement IEEE 754-compliant floating-point operations. The cores are pipelined and have a known latency (add/sub: 7 cycles, mul: 5 cycles, div: 6 cycles). The addition and subtraction use the same block. A select signal is used to choose between the two. After writing a macro description (applying the input values in the first cycle and reading the output a fixed number of cycles later), a C function that performs one floating-point operation can be synthesized using the corresponding IP core. Our scheduler is able to pipeline all operations and it knows that a sub and an add cannot be started at the same time. We tested the macros by implementing the Newton-Raphson algorithm to find a zero of a fifth degree polynomial:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{a_5x_n^5 + a_4x_n^4 + a_3x_n^3 + a_2x_n^2 + a_1x_n + a_0}{5a_5x_n^4 + 4a_4x_n^3 + 3a_3x_n^2 + 2a_2x_n + a_1} \quad . \quad (1)$$

The time needed to set up this design was less than two hours from C-code to synthesisable VHDL: 15' to create the Altera floating point megacores with the Altera QuartusII MegaWizard Plug-In Manager; 15' to convert all floating point operations in the C code into function calls, such as fp_add, fp_sub, ... (E.g. `x2=x*x;` becomes `x2=fp_mul(x,x);`); 30' to describe the macro schedules for each of these floating point operations; some additional time for constructing the top-level and testbench, and testing. On an Altera Stratix III (EP3SL150F1152C2), the design can clock at 150.60 MHz, and does one evaluation of (1) in 69 cycles, i.e. 458 ns. The design uses 20 DSP blocks (5% of the FPGA), 4608 Memory bits (< 1%), 2147 combinational ALUTs (2%), 26 memory ALUTs (< 1%) and 3079 dedicated logic registers (3%). On an Intel Core2 Quad CPU Q6600 at 2.40 GHz the same computation takes 43 ns or roughly 90 cycles. The focus of this experiment, however, is not the comparison of the FPGA and processor implementations, but the ease of integration of the floating-point cores.

5 Related Work

Some HLS tools focus on the datapath and have little attention for the communication interface. SPARK [9], e.g., generates 1 port for each array element. Impulse-C [10] offers a set of macros for the communication with hardware blocks, e.g., FIFOs and channels, but no user-specified interfaces can be defined.

The approach of Mentor Graphics' Catapult C [4] is similar to ours. Catapult C accepts a pure untimed C++ description as its input. The user can define its own interfaces with a library builder. The interface resource consists of an *I/O hardware block* connected to a standard *interface* (similar to the architecture in Fig. 1). Property mappings are used to describe the delay (for combinational logic) or initiation delay (needed for pipelining) of a transaction. This allows to pipeline different instantiations of the same transaction as in Fig. 4(c). We doubt, however, if it would be possible to detect a pipeline option as in Fig. 4(d).

Transaction Level Modeling (TLM) using SystemC is used for system-level design and architecture exploration. TLM-RTL transactors connect blocks with interfaces at different abstraction levels: a TLM interface, which is a function call, and an RTL interface. For simulation, this is similar to the concept of macros. However, our focus is on the hardware generation and not on the modeling.

SpecC has modeling capabilities similar to SystemC but has a designer-assisted tool flow to RTL [13]. Communication is implemented with protocol adapters, similar to [2], where an IO block is inserted for interface translation.

6 Conclusions

The growing complexity of digital system design has raised the need for more flexible HLS tools. This paper presents a method to extend such a tool with the ability to generate custom interface protocols or IP core integration, without the overhead that would be caused by wrappers. This enables a tighter integration of generated and custom-made hardware and may create a design flow in which the strengths of HLS tools and manual hardware design can be combined.

Acknowledgements. This research is supported by the I.W.T. grant 060068. Our research group at Ghent University is a member of the HiPEAC Network of Excellence. We would like to thank Altera for donating a Stratix III board.

References

1. Altera. Floating-Point Megafunctions User Guide, 1.0 edn. (March 2009)
2. Balarin, F., Passerone, R.: Functional verification methodology based on formal interface specification and transactor generation. In: Design, Automation and Test in Europe (DATE), pp. 1013–1018 (2006)
3. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2004: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, Juan-les-Pins, September 2004, pp. 7–16 (2004)
4. Bowyer, B.D., Gutberlet, P.P., Waters, S.J.: Interactive interface resource allocation in a behavioral synthesis tool, US Patent 0077906 (March 2008)
5. Coussy, P., Morawiec, A. (eds.): High-Level Synthesis: From Algorithm to Digital Circuit. Springer, Heidelberg (2008)
6. Devos, H., Meeus, W., Stroobandt, D.: CLooGVHDL and JCCI. DATE University Booth (CD–ROM), Nice, France, April 2009, pp. 1–2 (2009)
7. Devos, H., Van Campenhout, J., Verbauwhede, I., Stroobandt, D.: Constructing application-specific memory hierarchies on FPGAs. Transactions on High-Performance Embedded Architectures and Compilers 3(3) (2008) (to appear in LNCS)
8. Green, B.: Edge detection tutorial (2002), `http://www.pages.drexel.edu/~weg22/edge.html` (last accessed 2009.02.24)
9. Gupta, S., Gupta, R., Dutt, N., Nicolau, A.: SPARK, A Parallelizing Approach to the High-Level Synthesis of Digital Circuits. Kluwer Academic Publishers, Dordrecht (2004)

10. Impulse Accelerated Technologies. Impulse C, http://www.impulsec.com/ (last accessed 01.2009)
11. Ly, T., Knapp, D., Miller, R., MacMillen, D.: Scheduling using behavioral templates. In: Proceedings of the 32nd Design Automation Conference (DAC), pp. 101–106. ACM, New York (1995)
12. Moore, C., Devos, H., Stroobandt, D.: Optimizing on-chip memory systems for FPGAs. In: International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA (2009)
13. Shin, D., Gerstlauer, A., Dömer, R., Gajski, D.D.: An interactive design environment for C-based high-level synthesis of RTL processors. IEEE Trans. on VLSI Systems 16(4), 466–475 (2008)

Pipelined Microprocessors Optimization and Debugging

Bijan Alizadeh, Amir Masoud Gharehbaghi, and Masahiro Fujita

VLSI Design and Education Center (VDEC), University of Tokyo and CREST, Tokyo, Japan
{alizadeh,amir}@cad.t.u-tokyo.ac.jp,
fujita@ee.t.u-tokyo.ac.jp

Abstract. This paper proposes a methodology based on formal correspondence checking to automatically debug and also optimize pipelined microprocessors including reconfigurable processors with timing error recovery techniques. Since formal verification analyzes the design exhaustively, it may give good insights into not only debugging but also optimization of hardware designs with complicated control structures. The paper gives two main contributions, 1) modeling and formal verification of pipelined microprocessors including reconfigurable processors with timing error recovery techniques and 2) an approach to debug and optimize the implementation using the UCLID system as a correspondence checker. Using our method, the debug time can be reduced significantly. In addition, the implementation can be optimized by removing unnecessary signals and components while the correctness of the design is guaranteed.

Keywords: formal verification, architecture level debugging and optimization, pipelined microprocessors, reconfigurable processors.

1 Introduction

Reconfigurable computing is becoming increasingly attractive for a broad range of applications due to achieving much more flexibility in hardware and improving performance in comparison with software-only implementations. In such reconfigurable systems, the correctness of reconfigurable microprocessors is a key for their applications and should be checked before developing and debugging the application. Moreover, the increased parallelism provided by superscalar and out-of-order mechanisms in advanced microprocessors has made correctness statements for verification more complicated. Several formal verification techniques including symbolic model checking [1], theorem proving [2-3] and approaches based on decision procedures [4-5] have been proposed to verify such micro-architectures.

On the other hand there are some approaches [8-10] [12] based on SAT or SMT solvers to debug hardware designs using the erroneous traces. Those methods depend on existence of such traces and also their corresponding correct results. Usually the existence of the erroneous and also the correct traces rely on a testbench that is able to reveal the bugs. In [12] a diagnosis method is presented which extends the SAT-based diagnosis for the RTL designs. The design description as well as error candidate signals are specified in the word level and therefore word-level multiplexers are added

P. Sirisuk et al. (Eds.): ARC 2010, LNCS 5992, pp. 435–444, 2010.

into error signals. Finally to solve the resulting formula, a word-level solver is used. None of those methods address automatic debugging or optimization issues.

In this paper, we propose a methodology which utilizes formal verification techniques to debug and optimize a given pipelined processor more efficient. In our methodology, an RTL code of the implementation and the Instruction Set Architecture (ISA) as the specification are given. We assume that the RTL design is incrementally developed which is an implication to design reconfigurable processors. In this way we are able to easily define a list of error candidate signals which is one of the inputs to our debugging and optimization algorithms (see *VarList* in Fig. 3 and Fig. 5) and formal verification can effectively and incrementally verify design changes. First of all the RTL Verilog code is abstracted to the term-level in UCLID using *v2ucl* tool [11]. Then a correspondence checking between the specification and the implementation is performed until we obtain a golden model of the implementation. After that, the implementation is systematically modified and then symbolically simulated until we get an optimized model of the implementation. Our contributions in this paper are: 1) presenting a technique to locate and correct the bugs while correct outputs are derived from the specification instead of being provided by the user, 2) presenting a method to systematically optimize the design by looking for unnecessary control signals and avoiding functional redundancy and 3) utilizing the specification rather than one counter-example at a time to check all possible paths in the implementation by employing a symbolic simulator.

The rest of this paper is organized as follows. We explain our methodology in Section 2. In Section 3 we introduce our case study of a timing error recovery technique and present how to debug and optimize it. Finally we conclude in Section 4.

2 Microprocessor Debugging and Optimization Methodology

This section presents our methodology to debug a given design and improve its performance which is based on correspondence checking technique supported by UCLID [6]. Note that architecture-level verification and optimization are even important for reconfigurable microprocessors to guarantee the absence of hardware errors and to escape from bugs through reconfiguration.

The main phase of debugging or optimizing a design is to systematically modify it. To do so, an extra logic is added to the implementation. Fig. 1(a) shows a state variable *Var* which is connected to *m* blocks (B_1 to B_m). To change the value of *Var*, *m* multiplexers are inserted so that the original variable *Var* is attached to the 0-input of the multiplexers and the multiplexers' outputs are connected to the *Var*'s fanout cone

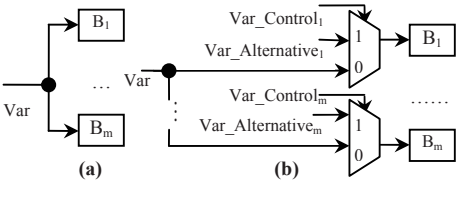


Fig. 1. Systematic modification

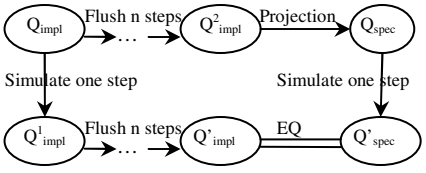


Fig. 2. Correspondence checking

as shown in Fig. 1(b). In addition, for each block B_i ($i=1$ *to* m) two new variables $Var\_Alternative_i$ and $Var\_Control_i$ are attached to multiplexer's 1-input and select-input respectively. Although this idea was applied to the problem of RTL debug [8], we utilize it at a higher level of abstraction where the design is implemented using uninterpreted functions and predicates.

The UCLID system utilizes the logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU) to specify and verify systems which have infinite or unbounded state [6]. A flushing refinement proof is used to check whether the state of a pipelined machine (implementation) is equivalent to its corresponding ISA state (specification) by completing the partially executed instructions in the pipeline. First the implementation (Q_{impl}) is symbolically simulated with an arbitrary input combination for one step ($Q_{impl} \rightarrow Q^1_{impl}$ in Fig. 2). Then the pipeline is flushed for n steps until all partially executed instructions can be completed and the programmer-visible parts of the implementation are saved ($Q^1_{impl} \rightarrow Q'_{impl}$). After that, we re-initialize to the start state and in order to have a corresponding state of the implementation in the specification, first the implementation is flushed for n steps ($Q_{impl} \rightarrow Q^2_{impl}$), and then the state of the implementation is projected into that of the specification ($Q^2_{impl} \rightarrow Q_{spec}$). After symbolically simulating the specification for one step ($Q_{spec} \rightarrow Q'_{spec}$), the programmer-visible components such as *Program Counter (PC), Register File (RF)* and *Memory (Mem)* in the Q'_{impl} should be equivalent to those of Q'_{spec}.

2.1 Debugging Technique

In order to check the correspondence between the implementation (*Impl*) and the specification (*ISA*) as discussed before, we check the following correspondence property:

$$\textbf{P1:}\ (\text{Impl.PC} = \text{ISA.PC} \ \&\ \text{Impl.RF} = \text{ISA.RF} \ \&\ \text{Impl.Mem} = \text{ISA.Mem}) \quad (1)$$

If the formula is valid, *Impl* is taken into account as a golden model of the implementation which can further be optimized using our optimization technique. If the formula is not valid, instead of checking the counterexample generated by the UCLID which is a time-consuming and difficult task, we apply our proposed debugging method. Fig. 3 depicts the proposed semi-automatic debugging technique. We assume that a buggy model of the implementation (*Impl*), the specification (*ISA*), and a list of variables which may be the root-causes of the bugs in the implementation (*VarList*) are given. Although *VarList* can contain all state variables in the implementation or a subset of them which can be automatically generated, since we have employed an incremental development methodology to implement the design, we select those variables which are incrementally added to the original design and may cause the design

fails. There is an assumption that a single failure can be corrected by adding a single multiplexer.

Our debugging technique starts by enriching the buggy implementation (*Impl*) in such a way that each variable in *VarList* is replaced with a multiplexer (*Enriched_Impl*) as explained in Fig. 1 (lines 1-2 in Fig. 3) when the correspondence property between *Enriched_Impl* and the *ISA* model is as follows:

P2: (Enriched_Impl.PC = ISA.PC & Enriched_Impl.RF = ISA.RF & Enriched_Impl.Mem = ISA.Mem) (2)

```
Debug_Technique(Impl, ISA, VarList, n)
Inputs: Impl: implementation; ISA: specification;
        VarList: list of variables;
Output: golden model of the implementation
1  For all variables in VarList
2     Enriched_Impl = Modify Impl based on Fig. 1;
3  Valid = Simulate (Impl, Enriched_Impl, ISA);
4  Look for variables with Var_Control=1;
5  Impl = Fix the bug;
6  Update VarList;
7  Valid = Simulate (Impl, ISA);
8  If (Valid)          return Impl;
9  ElseIf (VarList is empty)   Bug is not locatable!
10 Else go to step 1;
```

Fig. 3. Debug method

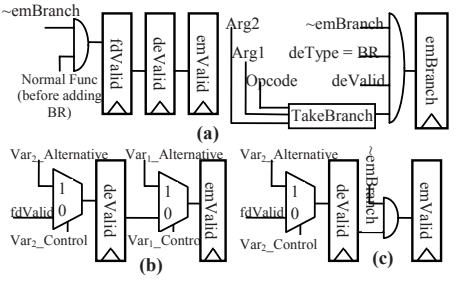


Fig. 4. (a) A buggy implementation, (b) first enriched implementation, (c) second enriched implementation

After that, two implementations, i.e. the buggy implementation (*Impl*) and the modified implementation (*Enriched_Impl*), are compared with the specification (*ISA*) using the UCLID system (line 3). To do so, the validity of *~(~P1 & P2)* is checked to indicate under what conditions the buggy implementation fails (because of ~P1 in the formula), while the modified one works correctly (due to using *P2* in the formula). At this point, the UCLID generates a counterexample which can easily be traced to locate the bug. We just need to look for those variables whose *Var_Control* signals are set to 1 (line 4). In other words, that part of the counter example shows the difference between the buggy implementation and the enriched one. Therefore the user can easily identify the reasons for failure by checking the state of the design at the $k\text{-}1^{th}$ and k^{th} steps and perform appropriate modification. According to the value of $Var_i\_Alternative$ when $Var_i\_Control$ is 1, the buggy implementation is modified and *VarList* is updated accordingly (lines 5-6). Then the correspondence between the *Impl* and the *ISA* model is checked (line 7). If the correspondence formula holds *Impl* is returned as a golden model (line 8). Otherwise if *VarList* is not empty the algorithm goes to step 1 to repeat the process (line 10 in Fig. 3) until the formula is valid or there is no variable in *VarList*.

As an example, consider a DLX processor with two *RR* and *RI* instructions. Fig. 4(a) shows a part of the implementation that has changed due to adding *BR* instruction, where *VarList* = {*deValid*, *emValid*}. Note that *emBranch* in Fig. 4(a) indicates whether or not a branch is taken, where *TakeBranch* is an uninterpreted predicate. Fig. 4(b) illustrates the enriched implementation after adding multiplexers into *deValid* and *emValid* state variables. After correspondence checking between two implementations and the specification, i.e., checking *~(~P1 & P2)*, we have found out

that Var_1_*Control* is activated at the 2nd step while a *BR* instruction is being executed. During this step, Var_1_*Alternative* is set to 0 which causes *emValid* to become 0 at the next step while in the buggy implementation *emValid* becomes 1. In other words, the execute stage (*emValid*) should be invalidated when a branch is taken (*emBranch* is true). Hence we have modified *emValid* state variable as shown in Fig. 4(c). The fourth row in Table 1, *DLX-DBG2*, gives the results.

It is interesting to note that after simulating the new implementation with the specification (according to line 7 in Fig. 3), the formula (*P1*) was not valid and therefore we have applied our debugging technique again. The *~(~P1 & P2)* formula is checked again when *VarList* = {*deValid*} and the enriched implementation is as shown in Fig. 4(c). In this case, Var_2_*Control* is activated at the 2nd step when the instruction is *BR* again. Moreover, Var_2_*Alternative* is set to 0 which causes *deValid=0* at the next step while in the buggy implementation *deValid=1*. It means that the decode stage (*deValid*) must be invalidated when a branch is taken. The fifth row in Table 1, *DLX-DBG3*, reports the results. Finally the correspondence checking between the implementation and the specification is successfully done.

2.2 Optimization Technique

Fig. 5 depicts the proposed optimization technique. We assume that a golden model of the implementation (*Impl*), the specification (*ISA*), and a list of variables (*VarList*) which may make the implementation more optimized are given. First of all, *Impl* is modified by replacing each variable in *VarList* with a multiplexer (*Enriched_Impl* in lines 1-2). Then a variable Var_i is selected from *VarList* (line 3) and Var_i_*Alternative* $\neq Var_i$ as an invariant is added to the enriched implementation (line 4). After that, the MUXs added into all variables except Var_i are deactivated by setting Var_j_*Control=0*, where $j \neq i$ (line 5). Finally the correspondence between two implementations (*Impl* and *Enriched_Impl*) and the specification (*ISA*) is checked (line 6). Although *P1* and *P2* are similar to those used by the debugging technique (see (1) and (2)), the correspondence formula for the optimization is *~(P1 and ~P2)* that generates conditions in which the golden implementation works correctly (*P1*), while the modified one fails (*~P2*). If the formula is valid, it means that there exists functional redundancy so that we can further optimize the implementation by removing this redundancy (line 7). If the formula is not valid, we can conclude that no optimization with respect to Var_i is possible. After removing Var_i from *VarList* (line 8), if *VarList* is empty *Opt_Impl* is returned as an optimized model of the implementation (line 9). If *VarList* is not empty, *Opt_Impl* is considered as *Impl* and the algorithm goes to step 1 to repeat the optimization process for another variable (line 10).

Let us consider the previous example again. Fig. 6(a) shows a part of the implementation after adding *BR* instruction, where *VarList* = {*mwValid*}. In order to see whether the functionality of *mwValid* can be optimized or not, its inputs are replaced with multiplexers and then activated one by one. When a multiplexer is added into *~emBranch* input as shown in Fig. 6(b) and Var_1_*Alternative* $\neq$ *~emBranch* is considered, the UCLID indicates that *~(P1 & ~P2)* is valid. In other words, *~emBranch* is redundant and can be removed. The last row in Table 1, *DLX-OPT3*, gives the results.

Optimization_Technique(Impl, ISA, VarList, n)
Inputs: *Impl*:implementation; *ISA*:specification; *VarList*: list of variables;
Output: *Opt_Impl*: optimized model of the implementation

```
1    For all variables in VarList
2        Enriched_Impl = Modify Impl based on Fig. 1;
3    Select Var_i from VarList;
4    Consider Var_i_Alternative ≠ Var_i as an invariant;
5    Deactivate MUXs of all variables except that of Var_i;
6    Valid = Simulate (Impl, Enriched_Impl, ISA);
7    If (Valid)  Opt_Impl = remove redundancy;
8    Remove Var_i from VarList;
9    If (VarList is empty)        return Opt_Impl;
10   Else Impl = Opt_Impl and go to step 1;
```

Fig. 5. Optimization method

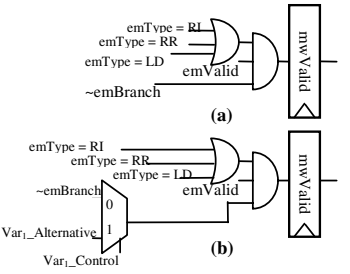


Fig. 6. (a) A part of implementation before optimization, (b) enriched implementation

3 Case Study: Pipelined Timing Error Recovery Techniques

In order to demonstrate the effectiveness of our method, we have considered timing error recovery (TER) technique in pipelined microprocessors. Parameter variation in integrated circuit devices has become a major challenge so that it may cause sections of a chip to be slower than others. Hence, logic paths in those sections may take longer delay to propagate signals and hence induce timing errors. In [7] a novel architecture, *Razor*, has been proposed to detect and correct occasional errors. The main idea is to append a shadow latch to each main flip-flop. The shadow latch uses a delayed clock to sample the outcome of the combinational logic one more time and to compare it with the data sampled earlier by the main flip-flop at the edge of the normal clock. If the data arrives late with respect to the normal clock, the main flip-flop stores a wrong value while the shadow latch may store the correct one. In this case a timing error has occurred. Although the detection is done in the Razor flip-flops, the timing errors are corrected through architectural replay mechanism. To model Razor-II technique, three things should be implemented: 1) timing error detection 2) bubble or no-operation (NOP) insertion and 3) re-execution or replay. Fig. 7 illustrates how they have been implemented with UCLID. Corresponding to each stage there are two variables 1) *TE* (timing error) which is the output of an uninterpreted predicate to specify if a timing error has occurred and 2) *Valid* as a state variable to indicate whether the related data is valid or not. For instance *fdValid* in Fig. 7(a) is the *Valid* register between the fetch and decode stages and shows whether the data provided by the fetch stage is valid or not. Also the TE_1 and TE_5 state whether the fetch stage and the write-back stage have a timing error problem, respectively. In addition, the $Error_i$ (i=1 to 5) state variable passes along the pipeline as shown in Fig. 7(b) to inform the next stages ($i+1$,…) if a timing error was detected at the stage k ($k=1$ *to* i). In Fig. 7(a), if ($Error_5$ *or* TE_i) is activated, the related *Valid* signal will be zero. Otherwise the normal function is done. Fig. 7(c) shows how an errant instruction can be re-executed. It can be seen in this figure that the PC is passed through the pipeline until we reach the write-back stage. If an error was detected in one of the stages in the pipeline, $Error_5$ will be asserted and therefore the PC in the *WB* stage, i.e., *mwPC*, is written into the PC in the fetch stage.

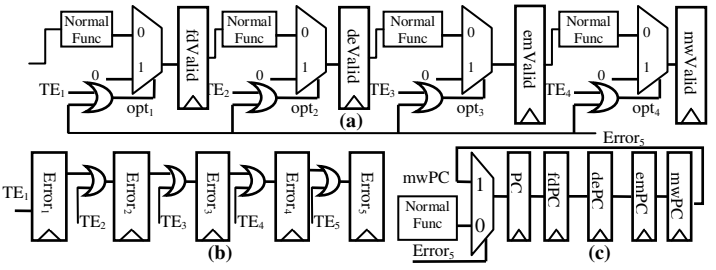


Fig. 7. Implementation of Razor-II timing error recovery technique

Table 1. Preliminary experimental results

Processor	#line	#var	#truth	#term	#bool	CNF-var	CNF-cls	time (s)
DLX-FV	597	190	3048	1782	43152	40952	121678	1.61
DLX-DBG1	896	203	4068	1966	64020	61336	181348	2.48
DLX-DBG2	672	179	3234	1573	49518	47327	139501	1.91
DLX-DBG3	651	179	3024	1596	49961	47767	140923	1.93
DLX-OPT1	896	268	5679	3066	105811	102063	302668	4.08
DLX-OPT2	896	219	4206	2356	73947	71028	210268	2.87
DLX-OPT3	664	154	2841	1218	38551	36696	108112	1.47

The experiments were performed on a 3.33GHz Intel Core2 Duo with 3 GB memory running Linux. zChaff has been used as the SAT solver within UCLID. Table 1 summarizes the results for different cases discussed in the following subsections. In this table, columns *#line* and *time* indicate the number of lines in the UCL file and the CPU time taken by the decision procedure in seconds respectively. In addition, the number of variables, the number of nodes corresponding to truth-expressions, i.e. Boolean expressions, the number of nodes corresponding to term-expressions, i.e. word-level expressions, the size of Boolean formula obtained after translating a CLU formula to a propositional formula, the number of CNF variables and clauses are reported in columns *#var*, *#truth*, *#term*, *#bool*, *CNF-var*, and *CNF-cls* respectively. Note that out-of-order architectures challenge existing verification techniques because the number of cycles required to empty large reorder buffer completely is so large and the logical formulas to be constructed and manipulated are too complex.

In order to formally verify this timing error recovery technique, we have implemented it into a 5-stage single-issue DLX processor which has five types of instructions: store word, load word, conditional branch, three-registers ALU instructions and ALU immediate instructions. The implementation is symbolically simulated with an arbitrary input combination for one step and then the pipeline is flushed for 5 steps until all partially executed instructions are completed. On the other hand, first the implementation is flushed for 5 steps and then the state of the implementation is projected into the state of the specification. Finally the specification is symbolically

simulated for one step. After that the two models are compared to see whether they are equivalent or not. One issue for verifying TER techniques is the fact that we are not able to model multiple clocks in the UCLID system. To alleviate this problem, we have considered an uninterpreted predicate *TErr* to randomly activate the error signal of stage *i* (TE_i *: i=1 to 5*). The second row, *DLX-FV*, in Table 1 shows the results for verifying 5-stage DLX processor with error recovery technique. Although our methodology is flexible enough that can be applied to out-of-order superscalar microprocessors, we leave it as a future work to show an automatic mechanism for debugging and optimization of wide-issue out-of-order microprocessors. Note that out-of-order architectures challenge existing verification techniques because the number of cycles required to empty large reorder buffer completely is so large and the logical formulas to be constructed and manipulated are too complex.

3.1 Semi-automatic Debugging

After implementing the timing error recovery techniques in the UCLID system, using our debugging methodology we found out some enhancements to our implementation as well as some bugs in the implementation of our case study. We have observed real bugs which were not false negative. For instance, one of bugs happened when a load instruction followed by two consecutive store instructions. In order to identify the failure reason, we have replaced $\sim next[Error_4]$ with m_Error_4 in Fig. 8(b) and considered $Error_4\_Alternative \neq \sim next[Error_4]$ as the invariant explained before. After simulating the design, we obtained a counterexample in which $Error_4\_Control$=1 and $Error_4\_Alternative$=1 at the 5th step of simulation. Based on the formula *~(~P1 and P2)* discussed in Fig. 3, the counterexample gives an example in which *P1* is not valid while *P2* is valid. It shows that if a timing error occurs in the execution stage (TE_3*=1*) and $\sim next[Error_4]$*=1* at the 5th step, two implementations behave differently. After more investigations, we figured out the main reason for the problem. The problem occurs when the second store instruction saves an incorrect data into the memory while the first store instruction has a timing error and the destination of the load instruction is equal to the source of the store instructions. In Fig. 8(a) the second store instruction writes something into the memory because the select-line of the multiplexer is true (*emType=ST & emValid=true &* $next[Error_4]$*=false*). In this way we could find out that $\sim next[Error_4]$ should be replaced by $\sim Error_4$. The third row in Table 1, *DLX-DBG1*, gives the results.

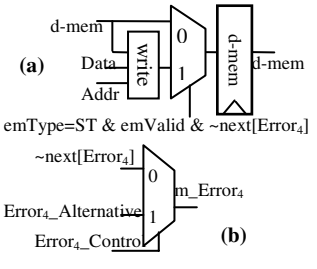


Fig. 8. An incorrect write into the memory – debugging example

3.2 Semi-automatic Optimization

After obtaining a golden model of the implementation, we have tried to optimize the implementation by removing unnecessary parts in an interactive manner. In other words, we have looked for some sections of the algorithm which are not necessary to be supported in special cases. We have applied the optimization method in Fig. 5 to the opt_i and $Error_i$ (i=1 to 5) variables in the design, where opt_i is the select-input of the multiplexer in Fig. 7(a). We have modified the above-mentioned variables one by one at different simulation steps when various timing errors have been taken into account. This way we could find some special cases which do not need to be considered:

1. When a timing error happens in the execution stage ($Timing\_error_3$), it is not necessary to insert bubble in the first stage (*fdValid*). *DLX-OPT1* row in Table 1 reports the results.
2. We have also observed that if a timing error occurs the instruction and the PC in the fetch stage are not important. The seventh row in Table 1, *DLX-OPT2*, gives the results.
3. Moreover, if multiple timing errors occur, it is sufficient to only consider the nearest one to the WB stage.

4 Conclusion and Future Work

In this paper, we have proposed a methodology for debugging and optimizing of pipelined microprocessors which is based on formal verification techniques. We have applied our method to timing error recovery techniques in pipelined processors in order to find out the source of the bugs as well as unnecessary signals and components. Although we have discussed our methodology based on the UCLID system, any other symbolic simulator can be utilized. As a future work, we are going to apply our methods to out-of-order superscalar microprocessors.

References

1. Jhala, R., McMillan, K.: Microarchitecture Verification by Compositional Model Checking. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 396–410. Springer, Heidelberg (2001)
2. Arons, T., Pnueli, A.: A Comparison of Two Verification Methods for Speculative Instruction Execution. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, p. 487. Springer, Heidelberg (2000)
3. Hosabettu, R., Gopalakrishnan, G., Srivas, M.: Verifying Advanced Microarchitectures that Support Speculation and Exceptions. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
4. Burch, J., Dill, D.: Automatic Verification of Pipelined microprocessor Control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994)
5. Velev, M.N.: Using Rewriting Rules and Positive Equality to Formally Verify Wide-issue Out-of-order Microprocessors with a Reorder Buffer. In: Design, Automation and Test in Europe (DATE), pp. 28–35 (2002)

6. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 78. Springer, Heidelberg (2002)
7. Das, S., Tokunaga, C., Pant, S., Ma, W.-H., Kalaiselvan, S., Lai, K., Bull, D., Blaauw, D.T.: RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. IEEE Journal of Solid-State Circuits 44(1), 32–48 (2009)
8. Smith, A., Veneris, A., Fahim Ali, M., Viglas, A.: Fault Diagnosis and Logic Debugging using Boolean Satisfiability. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems 24(10), 1606–1621 (2005)
9. Mangassarian, H., Veneris, A., Safarpour, S., Benedetti, M., Smith, D.: A Performance-Driven QBF-Based Iterative Logic Array Representation with Applications to Verification, Debug and Test. In: Int'l Conference on Computer-Aided Design (ICCAD), pp. 240–245 (2007)
10. Sulflow, A., Wille, R., Fey, G., Drechsler, R.: Evaluation of Cardinality Constraints on SMT-based Debugging. In: 39th International Symposium on Multiple-Valued Logic (ISMVL), pp. 298–303 (2009)
11. `http://uclid.eecs.berkeley.edu//v2ucl`
12. Mirzaeian, S., Zheng, F., Cheng, K.-T.: RTL Error Diagnosis Using a Word Level SAT-Solver. In: International Test Conference (ITC), pp. 1–8 (2008)

Author Index