Philip Brisk
José Gabriel de Figueiredo Coutinho
Pedro C. Diniz (Eds.)

LNCS 7806

Reconfigurable Computing: Architectures, Tools, and Applications

9th International Symposium, ARC 2013
Los Angeles, CA, USA, March 2013
Proceedings

Springer

Philip Brisk
José Gabriel de Figueiredo Coutinho
Pedro C. Diniz (Eds.)

Reconfigurable Computing: Architectures, Tools, and Applications

9th International Symposium, ARC 2013
Los Angeles, CA, USA, March 25-27, 2013
Proceedings

Springer

Volume Editors

Philip Brisk
University of California, Riverside
Department of Computer Science and Engineering
Riverside, CA 92521, USA
E-mail: philip@cs.ucr.edu

José Gabriel de Figueiredo Coutinho
Imperial College London, Department of Computing
SW7 2AZ London, UK
E-mail: gabriel.figueiredo@imperial.ac.uk

Pedro C. Diniz
University of Southern California
USC Information Sciences Institute
Marina del Rey, CA 90292, USA
E-mail: pedro@isi.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36811-0 e-ISBN 978-3-642-36812-7
DOI 10.1007/978-3-642-36812-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013931851

CR Subject Classification (1998): B.8, B.2, B.7, C.0, C.1.1-4, C.3, C.4, F.2.2

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

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Reconfigurable computing platforms offer increased performance gains and energy efficiency through coarse-grained and fine-grained parallelism coupled with their ability to implement custom functional, storage, and interconnect structures. As such, they have been gaining wide acceptance in recent years, spanning the spectrum from highly specialized custom controllers to general-purpose high-end programmable computing systems. The flexibility and configurability of these platforms, coupled with increasing technology integration, has enabled sophisticated platforms that facilitate both static and dynamic reconfiguration, rapid system prototyping, and early design verification. Configurability is emerging as a key technology for substantial product life-cycle savings in the presence of evolving product requirements, standards, and interface specifications.

The growth of the capacity of reconfigurable devices, such as FPGAs, has created a wealth of new research opportunities and intricate engineering challenges. Within the past decade, reconfigurable architectures have evolved from a uniform sea of programmable logic elements to fully reconfigurable systems-on-chip with integrated multipliers, memory elements, processors, and standard I/O interfaces. One of the foremost challenges facing reconfigurable application developers today is how to best exploit these novel and innovative resources to achieve the highest possible performance-and energy-efficiency; additional challenges include the design and implementation of next-generation architectures, along with languages, compilers, synthesis technologies, and physical design tools to enable highly productive design methodologies.

The International Applied Reconfigurable Computing (ARC) Symposium series provides a forum for the dissemination and discussion of ongoing research efforts in this transformative research area. The series of editions started in 2005 in Algarve, Portugal. The second edition of the symposium (ARCÕ06) took place in Delft, The Netherlands, during March 1–3, 2006, and was the first edition of the symposium to have selected papers published as a Springer LNCS (*Lecture Notes in Computer Science*) volume. Subsequent editions of the symposium have been held in Rio de Janeiro, Brazil (ARCÕ07), London, UK (ARCÕ08), Karlsruhe, Germany (ARCÕ09), Bangkok, Thailand (ARCÕ10), Belfast, UK (ARCÕ11), and Hong Kong, China (ARCÕ12).

This LNCS volume includes the papers selected for the ninth edition of the symposium (ARCÕ13), held in Los Angeles, California, USA, during March 25–27, 2013. The symposium attracted a large number of very good papers, describing interesting work on reconfigurable computing-related subjects. A total of 41 papers were submitted to the symposium from 20 countries: The Netherlands (4), France (3), Germany (4), Republic of South Korea (12), Brazil (14), Republic of China (8), Denmark (1), Mexico (2), Portugal (2), South Africa (1), Lebanon (1), Australia (1), Republic of Ireland (2), Puerto Rico (1), Spain (5), UK (2), India (2), Japan (5), Poland (1), and Greece (1). Submitted papers were evaluated by at least three members of the Program Committee. After a careful selection, 28 papers were accepted as full papers (acceptance rate of 38.9%) and 10 as short papers (global acceptance rate of 52.8%). These accepted papers led to a very interesting symposium program, which we consider to constitute a representative overview of ongoing research efforts in reconfigurable computing, a rapidly evolving and maturing field.

Several persons contributed to the success of the 2013 edition of the symposium. We would like to acknowledge the support of all the members of this year's symposium Steering and Program Committees in reviewing papers, in helping with the paper selection, and in giving valuable suggestions. Special thanks also to the additional researchers who contributed to the reviewing process, to all the authors that submitted papers to the symposium, and to all the symposium attendees. Last but not least, we are especially indebted to Alfred Hofmann and Anna Kramer from Springer for their support and work in publishing this book and to Jüergen Becker from the University of Karlsruhe for the strong support regarding the publication of the proceedings as part of the LNCS series.

January 2013

Philip Brisk
José Gabriel de Figueiredo Coutinho
Pedro C. Diniz

Organization

The 2013 Applied Reconfigurable Computing Symposium (ARC 2013) was organized by the Information Sciences Institute (ISI) of the University of Southern California (USC), in Marina del Rey California, USA.

Organizing Committee

General Chair

Pedro C. Diniz	USC/ISI, USA

Program Chairs

Philip Brisk	University of California Riverside, USA
José G.F. Coutinho	Imperial College London, UK

Finance Chair

Pedro C. Diniz	USC/ISI, USA

Sponsorship Chair

Ray C.C. Cheung	City University of Hong Kong, China

Proceedings Chair

Pedro C. Diniz	USC/ISI, USA

Special Journal Edition Chairs

José G.F. Coutinho	Imperial College London, UK
João M.P. Cardoso	FEUP/University of Porto, Portugal

Local Arrangements Chair

Jeremy Abramson	USC/ISI, USA

Steering Committee

George Constantinides	Imperial College London, UK
João M.P. Cardoso	FEUP/University of Porto, Portugal
Koen Bertels	Delft University of Technology, The Netherlands

Mladen Berekovic	Technische Universität Braunschweig, Germany
Pedro C. Diniz,	USC Information Sciences Institute, USA
Stamatis Vassiliadis	Delft University of Technology, The Netherlands
Walid Najjar	University of California Riverside, USA

Program Committee

José Carlos Alves	FEUP/University of Porto, Portugal
Jeff Arnold	Strech Inc., USA
Michael Attig	Xilinx Research Labs, San Jose, USA
Zack Backer	Los Alamos National Lab., Los Alamos, USA
Jürgen Becker	University of Karlsruhe (TH), Germany
Khaled Benkrid	University of Edinburgh, UK
Neil Bergmann	University of Queensland, Australia
Koen Bertels	Delft University of Technology, The Netherlands
Jean-François Boland	École de Technologie Supérieure, Canada
Christos-Savvas Bouganis	Imperial College London, UK
Stephen Brown	Altera, Toronto University, Canada
Mihai Budiu	Microsoft Research, USA
João M.P. Cardoso	FEUP/University of Porto, Portugal
Ray C.C. Cheung	City University of Hong Kong, China
Paul Chow	University of Toronto, Canada
George Constantinides	Imperial College London, UK
Steven Derrien	INRIA Rennes, France
Florent de Dinechin	École Normale Supérieure de Lyon, France
Pedro C. Diniz	USC Information Sciences Institute, USA
Robert Esser	Apple Inc., USA
Suhaib Fahmy	Nanyang Technological University, Singapore
António Ferrari	University of Aveiro, Portugal
João Canas Ferreira	FEUP/University of Porto, Portugal
Kris Gaj	George Mason University, USA
Carlo Galuzzi	Delft University of Technology, The Netherlands
Guy Gogniat	Université de Bretagne Sud, France
Diana Göhringer	Karlsruhe Institute of Technology, Germany
Maya Gokhale	Lawrence Livermore Laboratories, USA
Yajun Ha	National University of Singapore, Singapore
Frank Hannig	University of Erlangen-Nurenberg, Germany
Jim Harkin	University of Ulster, UK
Reiner Hartenstein	University of Kaiserslautern, Germany
Roman Hermida	Universidad Complutense, Madrid, Spain
Christian Hochberger	Technical University of Dresden (TU), Germany

Michael Hübner	University of Bochum, Germany
Masahiro Iida	Kumamoto University, Japan
Yasushi Inoguchi	Japan Advanced Institute of Science and Technology, Japan
Tomonori Izumi	Ritsumeikan University, Japan
Ryan Kastner	University of California, San Diego, USA
Taemin Kim	Intel Corp., USA
Andreas Koch	Technical University of Darmstadt (TU), Germany
Ram Krishnamurthy	Intel Corp., USA
Philip Leong	University of Sydney, Australia
Mingjie Lin	University of Central Florida, USA
Wayne Luk	Imperial College London, UK
Terrence Mak	Newcastle University, UK
Eduardo Marques	University of São Paulo, Brazil
Konstantinos Masselos	University of the Peloponnese, Greece
Sanu Mathew	Intel Corp., USA
John McAllister	Queen's University of Belfast, UK
Seda Memik	Northwestern University, USA
Takefumi Miyoshi	The University of Electro-Communications, Japan
Fearghal Morgan	National University of Ireland, Galway, Ireland
Katherine (Compton) Morrow	University of Wisconsin-Madison, USA
Walid Najjar	University of California, Riverside, USA
Vikram Narayana	George Washington University, USA
Brent Nelson	Brigham Young University, USA
Horácio Neto	INESC-ID/IST, Portugal
Smail Niar	University of Valenciennes, France
Elaine Ou	University of Sydney, Australia
Joon-seok Park	Inha University, Inchon, Republic of South Korea
Miquel Pericas	Tokyo Institute of Technology, Japan
Thilo Pionteck	University of Lübeck, Germany
Joachim Pistorius	Altera Corp., USA
Marco Platzner	University of Paderborn, Germany
Christian Plessl	University of Paderborn, Germany
Bernard Pottier	University of Bretagne, France
Patrice Quinton	INRIA Rennes, France
Sanjay Rajopadhye	Colorado State University, USA
Kyle Rupnow	Advanced Digital Sciences Center, Singapore
Tsutomu Sasao	Kyushu Institute of Technology, Japan
Yukinori Sato	Japan Advanced Institute of Science and Technology, Japan
Erkay Savas	Sabanci University, Turkey
Patrick Schaumont	Virginia Tech, USA
Farhana Sheikh	Intel Corp., USA

Pete Sedcole	Celoxica, France
Lesley Shannon	Simon Fraser University, USA
Yuchiro Shibata	Nagasaki University, Japan
Pedro Trancoso	University of Cyprus, Cyprus
Markus Weinhardt	Osnabrück University of Applied Sciences, Germany
Stephan Wong	Delft University of Technology, The Netherlands
Roger Woods	Queen's University of Belfast, UK
Yoshiki Yamaguchi	Tsukuba University, Japan
Peter Zipf	University of Kassel, Germany
Tim Todman	Imperial College London, UK
Henry Styles	Xilinx Research Labs, San Jose, USA
David Thomas	Imperial College London, UK
Qiang Liu	Tianjin University, China
Yuet-Ming Lam	Macau University of Science and Technology, P.R. China

Additional Reviewers

Fakhar Anjam	Delft University of Technology, The Netherlands
Falco Bapp	Karlsruhe Institute of Technology, Germany
Yves Blaquière	Université du Québec à Montréal, Canada
Anthony Brandon	Delft University of Technology, The Netherlands
Jon Butler	Naval Postgraduate School, USA
Jean Pierre David	École Polytechnique de Montréal, Canada
Heiner Giefers	Universität Paderborn, Germany
Junjun Gu	Altera, USA
Gerald Hempel	Technische Universität Dresden, Germany
José Arnaldo M. De Holanda	Universidade de São Paulo, Brazil
Yoshihiro Ichinomiya	Kumamoto University, Japan
Kazuki Inoue	Kumamoto University, Japan
Natalie Enright Jerger	University of Toronto, Canada
Tobias Kenter	Universität Paderborn, Germany
Alexander Klimm	Karlsruhe Institute of Technology, Germany
Changgong Li	Technische Universität Darmstadt, Germany
Nikos Leveris	Altera, USA
Junxiu Liu	University of Ulster, UK
Charles Lo	Xilinx, Canada
Leandro Martinez	Universidade de São Paulo, Brazil
Joachim Meyer	Karlsruhe Institute of Technology, Germany
Sascha Muehlbach	Center for Advanced Security Research Darmstadt, Germany

Lars Schaefers	Universität Paderborn, Germany
Roel Seedorf	Delft University of Technology, The Netherlands
Florian Stock	Technische Universität Darmstadt, Germany
Joseph Tarango	University of California, Riverside, USA
Claude Thibeault	École de Technologie Supérieure, Canada
Qian Zhao	Kumamoto University, Japan

Table of Contents

Heterogeneous Reconfigurable System for Adaptive Particle Filters in Real-Time Applications

Thomas C.P. Chau[1], Xinyu Niu[1], Alison Eele[3], Wayne Luk[1], Peter Y.K. Cheung[2], and Jan Maciejowski[3]

[1] Department of Computing, Imperial College London, UK
{c.chau10,niu.xinyu10,w.luk}@imperial.ac.uk
[2] Department of Electrical and Electronic Engineering, Imperial College London, UK
p.cheung@imperial.ac.uk
[3] Department of Engineering, University of Cambridge, UK
{aje46,jmm1}@cam.ac.uk

Abstract. This paper presents a heterogeneous reconfigurable system for real-time applications applying particle filters. The system consists of an FPGA and a multi-threaded CPU. We propose a method to adapt the number of particles dynamically and utilise the run-time reconfigurability of the FPGA for reduced power and energy consumption. An application is developed which involves simultaneous mobile robot localisation and people tracking. It shows that the proposed adaptive particle filter can reduce up to 99% of computation time. Using run-time reconfiguration, we achieve 34% reduction in idle power and save 26-34% of system energy. Our proposed system is up to 7.39 times faster and 3.65 times more energy efficient than the Intel Xeon X5650 CPU with 12 threads, and 1.3 times faster and 2.13 times more energy efficient than an NVIDIA Tesla C2070 GPU.

1 Introduction

Particle filter (PF), also known as sequential Monte Carlo (SMC) method, is a statistical method for dealing with dynamic systems having nonlinear and non-Gaussian properties. PF has been applied to real-time applications including object tracking [1], robot localisation [2], speech recognition [3] and air traffic control [4]. However, PF operates on a large number of particles resulting in long execution times, which limits the application of PF to real-time systems.

In this paper, an adaptive algorithmic and architectural approach using reconfigurable hardware is proposed for PF in real-time applications. An adaptive PF algorithm is employed to dynamically adjust the size of particle set for reduced computation complexity. A heterogeneous reconfigurable system (HRS) consisting of a multi-core CPU and an FPGA is developed for the adaptive PF algorithm. As most of the PF operations can be performed independently, the algorithm suits ideally for implementation in FPGA which consists of thousands of customisable resources and dedicated digital signal processing units to exploit parallel processing. The challenge is to meet real-time requirement by organising the operations in streaming manner to maximise throughput and hide latency.

The contributions of this paper include:

1. Adaptive PF algorithm: the computational complexity of PF is reduced through adapting the size of particle set dynamically, and the algorithm is optimised for hardware acceleration (Section 3).
2. Heterogeneous architecture: fully pipelined computations of the PF are streamed through the FPGA kernel, while control-oriented computations are handled by the host CPU (Section 4).
3. Energy saving by run-time reconfiguration: FPGA is reconfigured to low-power mode dynamically (Section 4).
4. Prototype: a robot localisation application is implemented on an FPGA based on the proposed adaptive PF approach. Compared to a non-adaptive implementation, the idle power is reduced by 34% and the overall energy consumption decreases for 26-34% (Section 5).

2 Background and Related Work

This section briefly outlines the PF algorithm. A more detailed description can be found in [5]. PF estimates the state of a system by a sampled-based approximation of the state probability density function. The state of a system in time step t is denoted by X_t. Sequences of control information and observations are denoted by U_t and Y_t respectively. Three pieces of information about the system are known a-priori: a) $p(X_0)$ is the probability of the initial state of the system, b) $p(X_t|X_{t-1}, U_{t-1})$ is the state transition probability of the system's current state given a previous state and control information, c) $p(Y_t|X_t)$ is the observation model describing the likelihood of observing the measurement at current state. PF approximates the desired posterior probability $p(X_t|Y_{1:t})$ using a set of P particles $\{\chi_t^i|i = 1, ..., P\}$ with their associated weights $\{w_t^i|i = 1, ..., P\}$. X_0 and U_0 are initialised. This computation consists of three iterative steps.

1. **Sampling:** A new particle set $\widetilde{\chi}_t^i$ is drawn from the distribution $p(X_t|X_{t-1}, U_{t-1})$, forming a prediction of the distribution of X_t.
2. **Importance:** Likelihood $p(Y_t|\widetilde{\chi}_t^i)$ of each particle is calculated. The likelihood indicates whether the current measurement Y_t matches the predicted state $\widetilde{\chi}_t^i$. Then a weight w^i is assigned to the particle. The higher the likelihood, the higher the weight.
3. **Resampling:** Particles with higher weights are replicated and the number of particles with lower weights are reduced. With resampling the particle set has a smaller variance. The particle set is then used in the next time step to predict the posterior probability subsequently. The distribution of the resulting particles χ_t^i approximates $p(X_t|Y_{1:t})$.

The parallelism of particles in PF means it can be accelerated using specialised hardware with massive parallelism and pipelining. In [1], an approach for PF on a hybrid CPU/FPGA platform is developed. Using a multi-threaded programming model, computation is switched between hardware and software during run-time to react to performance requirements. Resampling algorithms and architectures for distributed PFs are proposed in [6].

Adaptive PFs have been proposed to improve performance or quality of state estimation by controlling the number of particles dynamically. Likelihood-based adaptation controls the number of particles such that the sum of weights exceeds a pre-specified threshold [7]. Kullback Leibler distance (KLD) sampling is proposed in [8], which offers better quality results than likelihood-based approach. KLD sampling is improved in [9] by adjusting the variance and gradient of data to generate particles near high likelihood regions. The above methods introduce data dependencies in the sampling and importance steps, so they are difficult to be parallelised. An adaptive PF is proposed in [10] that changes the number of particles dynamically based on estimation quality. Our previous work [11] extends their techniques for multi-processor system on FPGA. The number of particles and active processors change dynamically but the performance is limited by soft-core processors. In [12], a mechanism and a theoretical lower bound for adapting the sample size of particles is presented.

3 Adaptive Particle Filter

This section introduces an adaptive PF algorithm where the size of the particle set is changed in each time step. The algorithm is inspired by [12], and we optimise the algorithm to exploit the capability of FPGAs to support streaming and deep pipelines.

Algorithm 1 shows the processing step of PF. The first part is performed on the FPGA because the operations can be scheduled sequentially to maximise throughput. The second part is done on the CPU because the operations involve non-sequential or random access of data, such as sorting and resampling, that cannot be mapped efficiently to FPGA's streaming architecture.

Sampling and Importance (line 4 to 5): In time step $t-1$, the number of particles is $P_{t-1} \leq P_{max}$. A particle has dimensions $d = 1, ..., dim$. Particle set $\{\chi^i_{d,t-1}\}$ is sampled to $\{\widetilde{\chi}^i_{d,t}\}$ and importance weight $\{\widetilde{w}^i\}$ is assigned, where $i = 1, ..., P_{t-1}$. For simplicity, we denote the set of $P_{d,t-1}$ particles $\{\widetilde{\chi}^i_{d,t}\}$ as a vector $\widetilde{X}_{d,t} = \{\widetilde{\chi}^1_{d,t}, \widetilde{\chi}^2_{d,t}, ...\widetilde{\chi}^{P_{t-1}}_{d,t}\}$. $\{\widetilde{X}_{d,t}\}$ and $\{\widetilde{w}^i\}$ give an estimation of the current system state at dimension d.

Calculate the Lower Bound of Particle Set Size (line 6): This step derives the smallest number of particles that are needed to bound the approximation error. This number, denoted as $\widetilde{P}_t$, is referred to as the lower bound of sampling size. It is calculated by Equation 1 to 5.

$$\widetilde{P}_t = \max_d^{dim} \widetilde{P}_{d,t} \tag{1}$$

Algorithm 1. Adaptive PF algorithm

```
1: $P_0 \leftarrow P_{max}$, $\{X_0\}$ ←random set of particles, $t = 1$
2: for each step $t$ do
3:     —On FPGA—
4:     Sample a new state $\{\widetilde{\chi}_t^i\}$ from $\{\chi_{t-1}^i\}$ where $i = 1, ..., P_{t-1}$
5:     Calculate unnormalised importance weights $\widetilde{w}^i$ and accumulate the weights as $w_{sum}$
6:     Calculate the lower bound of sample size $\widetilde{P}_t$ by Equation 1 to 5
7:     —On CPU—
8:     Sort $\widetilde{\chi}_t$ in descending $w^i$
9:     if $\widetilde{P}_t < P_{t-1}$ then
10:        $P_t = max\left(\lceil\widetilde{P}_t\rceil, P_{t-1}/2\right)$
11:        Set $a = 2P_t - P_{t-1}$ and $b = P_t$
12:        –Do the following loop in parallel–
13:        for $i$ in $P'$ do
14:            $\widetilde{\chi}_t = \frac{\chi_t^a w^a + \chi_t^b w^b}{w^a + w^b}$
15:            $w^i = w^a + w^b$
16:            $a = a + 1$ and $b = b - 1$
17:        end for
18:    else if $\widetilde{P}_t \geq P_{t-1}$ then
19:        $a = 0$ and $b = 0$
20:        for $i$ in $P_t - P_{t-1}$ do
21:            if $w^a < w^{a+1}$ and $a < P_t$ then
22:                $a = a + 1$
23:            end if
24:            $\widetilde{\chi}_t^{P_{t-1}+b} = \widetilde{\chi}_t^a/2$
25:            $\widetilde{\chi}_t^a = \widetilde{\chi}_t^a/2$
26:            $w^{P_{t-1}+b} = w^a/2$
27:            $w^a = w^a/2$
28:            $b = b + 1$
29:        end for
30:    end if
31:    Resample $\{\widetilde{\chi}_t^i\}$ to $\{\chi_t^i\}$ where $i = 1, ..., P_t$
32: end for
```

$$\widetilde{P}_{d,t} = \sigma_d^2 \cdot \frac{P_{max}}{Var(\widetilde{X}_{d,t})} \tag{2}$$

$$\begin{aligned}\sigma_d^2 = \frac{1}{(w_{sum})^2} \cdot \sum_{i=1}^{P_{t-1}} \left(\widetilde{w}^i \cdot \widetilde{\chi}_{d,t}^i\right)^2 - 2 \cdot E(\widetilde{X}_{d,t}) \cdot \sum_{i=1}^{P_{t-1}} (\widetilde{w}^i)^2 \cdot \widetilde{\chi}_{d,t}^i \\ + \left(E(\widetilde{X}_{d,t})\right)^2 \cdot \sum_{i=1}^{P_{t-1}} \left(\widetilde{w}^i\right)^2\end{aligned} \tag{3}$$

$$Var(\widetilde{X}_{d,t}) = \frac{1}{w_{sum}} \cdot \sum_{i=1}^{P_{t-1}} \widetilde{w}^i \cdot (\widetilde{\chi}_{d,t}^i)^2 - \left(E(\widetilde{X}_{d,t})\right)^2 \tag{4}$$

$$E(\widetilde{X}_{d,t}) = \frac{1}{w_{sum}} \cdot \sum_{i=1}^{P_{t-1}} \widetilde{w}^i \cdot \widetilde{\chi}_{d,t}^i \tag{5}$$

$\widetilde{w}^i$ is unnormalised. To calculate normalised weights w^i, a trivial approach is to stream data through the FPGA twice, one for accumulating the sum of weights

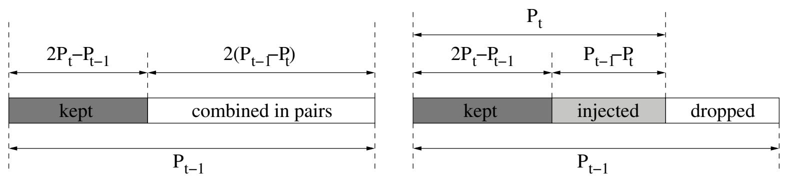
w_{sum} and one for dividing the weights $\widetilde{w}^i$ by w_{sum}. This method is inefficient as it reduces the throughput of the FPGA by half. Without degrading the performance, our design computes w_{sum} and $\widetilde{w}^i$ simultaneously as data stream through the FPGA. As shown in Equation 3 to 5, correct values of σ_d^2, $Var(\widetilde{X}_{d,t})$ and $E(\widetilde{X}_{d,t})$ appear at the last cycle by dividing the last piece of data by w_{sum}.

Meanwhile, Equation 3 to 5 involve accumulation which requires feedback of data in the previous cycle. If accumulation is performed in floating-point representation, the feedback path would take multiple clock cycles and greatly reduce throughput. Therefore, we use fixed-point data path such that the delay of the feedback path is kept in one clock cycle. The precision is designed to ensure that no overflow or underflow occurs.

Particle Set Size Tuning (line 8 to 30): The particle sample size is tuned to fit the lower bound P_t.

First, the particles are sorted in descending order according to their weight. Then, as the new sample size can increase or decrease, there are two cases.

- **Case I: Particle set reduction when $\widetilde{P}_t < P_{t-1}$**
  The lower bound P_t is set to $max\left(\lceil \widetilde{P}_t \rceil, P_{t-1}/2\right)$. Since the new size is smaller than the old one, some particles are combined to form a smaller particle set. Fig. 1 illustrates the idea of particle reduction. The first $2P_t - P_{t-1}$ particles with higher weights are kept and the remaining $2(P_{t-1} - P_t)$ particles are combined in pairs. As a result, there are $P_{t-1} - P_t$ new particles injected to form the target particle set with P_t particles. To remove loop dependency, we restrict that particles are combined deterministically. Therefore, each iteration of the loop can be processed independently and accelerated using multiple threads. The complexity of the loop is in $\mathcal{O}\left(\frac{P_{t-1}-P_t}{N_{parallel}}\right)$, where $N_{parallel}$ indicates the level of parallelism.



(a) Combining the last $2(P_{t-1} - P_t)$ particles with lower weights (b) P_t new particles are formed

Fig. 1. Particle set reduction

- **Case II: Particle set expansion when $\widetilde{P}_t \geq P_{t-1}$**
  The lower bound P_t is set to $\widetilde{P}_t$. Some particles are taken from the original set and are incised to form a larger set. The particles with larger weight would have more descendants. As shown in line 18 to 30, the process requires picking the particle with the largest weight at each iteration of particle incision.

Since the particle set is pre-sorted, the complexity of particle set expansion is $\mathcal{O}(P_t - P_{t-1})$.

Resampling (line 31): Resampling is performed to pick P_t particle from $\widetilde{X}_t$ to form X_t. The process has a complexity of $\mathcal{O}(P_t)$.

4 Heterogeneous Reconfigurable System

This section describes the proposed heterogeneous reconfigurable system (HRS) which makes use of run-time reconfiguration for power and energy reduction. The architectural diagram of HRS is shown in Fig. 2. The system consists of an FPGA board and a multi-threaded host CPU. The FPGA resources are customised to a deeply pipelined structure and the CPU performs coordination of particles. The FPGA has its own on board dynamic random-access memory (DRAM) because the amount of data is too large to be stored on-chip.

For the sampling and importance processes, the computation is independent for every particle. Therefore, particles are organised in a stream that is fed to the FPGA. In every clock cycle, one particle is taken from the FPGA's onboard DRAM. The FPGA kernel has a long pipeline that is filled with particles, and therefore many particles are processed at once. Fixed-point data representation is customised at each pipeline stage to reduce bit-widths and hence FPGA resource usage. One particle is written back to the DRAM in every clock cycle.

For lower bound calculation, particle set size tuning and resampling processes, the host CPU gathers all the particle data from the FPGA via PCI Express.

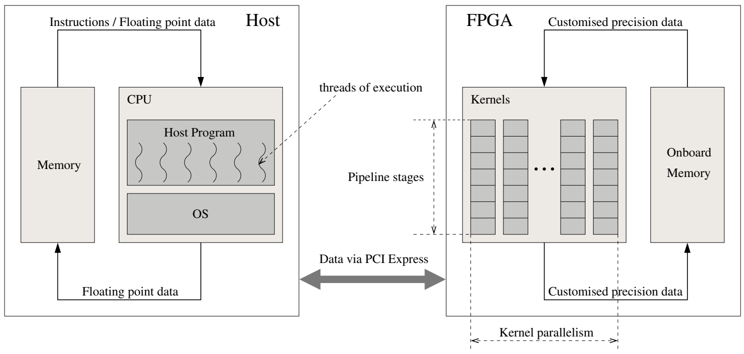


Fig. 2. Heterogeneous reconfigurable system

We derive a model to analyse the total computation time of the proposed system. The model helps us to design a configuration that can satisfy the real-time bound and, if necessary, amend the real-time application's specification. The model is validated by experiments in Section 5.

The total computation time of the system (T_{comp}) consists of three components.

$$T_{comp} = T_{kernel} + T_{host} + T_{io} \tag{6}$$

Kernel Time: It describes the time spent on the FPGA kernel for the sampling and importance processes. P_t denotes the number of particles at current time step and $freq_{kernel}$ denotes the clock frequency of the FPGA kernel. The sampling and importance processes can be repeated for N_{si} times before going to the resampling process. N_{kernel} denotes the level of parallelism as multiple kernels can be instantiated in the FPGA. L is the latency due to pipelining.

$$T_{kernel} = \frac{P_t \cdot N_{si}}{freq_{kernel} \cdot N_{kernel}} + L - 1 \tag{7}$$

Host Time: It describes the time spent on the host CPU. The clock frequency and number of threads of the host CPU are represented by $freq_{host}$ and N_{thread} respectively. par is an algorithm-specific parameter in the range of $[0, 1]$ which represents the ratio of CPU instructions that are parallelisable, and α is a scaling constant derived empirically.

$$T_{host} = \alpha \cdot \frac{P_t}{freq_{host}} \cdot \left(1 - par + \frac{par}{N_{thread}}\right) \tag{8}$$

IO Time: It describes the time of moving particle data between the FPGA's onboard DRAM and host memory. dim is the number of dimensions of a particle, e.g. if a particle represents coordinate (x, y) and weight, $dim = 3$. bw is the bit-width to represent one dimension. $freq_{pcie}$ is the clock frequency of PCI Express bus and $lane$ is the number of PCI Express lanes. Since PCI Express encodes data during transfer, the effective data is denoted by eff (in PCI Express gen2 the value is 8/10). The constant 2 accounts for the movement of data in and out of the FPGA.

$$T_{io} = \frac{2 \cdot P_t \cdot dim \cdot bw}{freq_{pcie} \cdot lane \cdot eff} \tag{9}$$

In real-time applications, the time for each step is fixed and is known as real-time bound T_{rt}. The derived model helps system designers to ensure that the computation time T_{comp} is shorter than T_{rt}. An idle time T_{idle} is introduced to represent the gap between finish time of computation and real-time bound.

$$T_{idle} = T_{rt} - T_{comp} \tag{10}$$

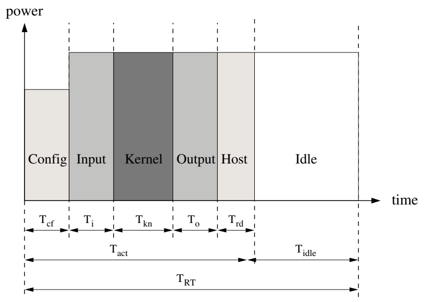
Fig. 3(a) shows the timing of system operations. It illustrates that the FPGA is still drawing power after the computation finishes. We propose a method to reduce dynamic power by using run-time reconfiguration of FPGA. During idle time, the FPGA is loaded with a low-power configuration which has minimal active resources and runs at a very low clock frequency. The idea is shown in

Fig. 3(b). Equation 11 describes the sleep time when the FPGA is idle and being loaded with the low-power configuration. If the sleep time is positive, it is always beneficial to load the low-power configuration.

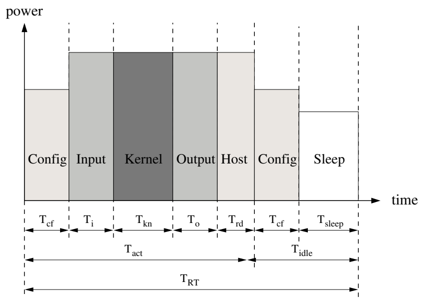
$$T_{sleep} = T_{idle} - T_{config} \tag{11}$$

Configuration Time (T_{config}): It describes the time needed to download a configuration bit-stream to the FPGA. $size_{bs}$ represents the size of bitstream in bits. $freq_{config}$ is the configuration clock frequency in Hz and $port_{config}$ is the configuration port width in bits.

$$T_{config} = \frac{size_{bs}}{freq_{config} \cdot port_{config}} \tag{12}$$



(a) Without reconfiguration



(b) With reconfiguration to low-power mode during idle

Fig. 3. Timing of the system's operations

5 Result and Evaluation

To evaluate the HRS and make comparison with other systems, a simultaneous robot localisation and people-tracking application is implemented. Given a priori learned map, a robot receives sensor values and moves at regular time intervals. In each time step, M people are tracked by the robot. The state of the whole system of robot and people is represented by a state vector X_t:

$$X_t = \{R_t, H_{t,1}, H_{t,2}, ..., H_{t,M}\} \tag{13}$$

R_t denotes the robot's pose at time t, and $H_{t,1}, H_{t,2}, ..., H_{t,M}$ denote the locations of the M people.

The PF uses the following equation to represent the posterior of robot and people locations:

$$p(X_t|Y_t, U_t) = p(R_t|Y_t, U_t) \prod_{m=1}^{M} p(H_{t,m}|R_t, Y_t, U_t) \tag{14}$$

Y_t is the sensor measurement and U_t is the control of the robot at time t. The robot path posterior $p(R_t|Y_t, U_t)$ is represented by a set of robot particles. The distribution of a person's location $p(H_{t,m}|R_t, Y_t, U_t)$ is represented by a set of people particles, where each people particle set is attached to one particular robot particle. Therefore, if there are P_r robot particles representing the posterior over robot path, there are P_r people particle sets, each has P_h particles.

In the application, the map has an area of 12m*18m. The robot makes a movement of 0.5m every 5s, i.e. $T_{rt} = 5s$. The robot can track 8 people at the same time. The system supports a maximum of 8192 particles for robot-tracking and each robot particle is attached with 1024 particles for people-tracking. Therefore, the maximum number of kernel cycles is 8*8192*1024=67M.

Experiment Settings: For the evaluation of HRS, we use the MaxWorkstation reconfigurable accelerator system from Maxeler Technologies. It has an FPGA board equipped with a Xilinx Virtex-6 XC6VSX475T FPGA which has 297,600 lookup tables (LUTs), 595,200 flip-flops (FFs), 2,016 digital signal processors (DSPs) and 1,064 block RAMs. The FPGA board is connected to an Intel i7-870 quad-core CPU clocked at 2.93GHz through a PCI Express link with a bandwidth of 2 GB/s. We develop the FPGA kernels using the MaxCompiler, which adopts a streaming programming model. At each pipeline stage, the fixed-point calculations are customised to different mantissa bit-widths.

There are two FPGA configurations: a) *sampling and importance configuration* is clocked at 100MHz, with 115961 LUTs (39%), 169188 FFs (28%), 967 DSPs (48%) and 257 block RAMs (24%) per kernel. b) *Low-power configuration* is clocked at 10MHz, with 5962 LUTs (2%), 6943 FFs (1%) and 12 block RAMs (1%). It uses the minimum amount of resources just to maintain communication between the FPGA and CPU.

The CPU performance results are obtained from an Intel Xeon X5650 CPU clocked at 2.66GHz. It is optimised by ICC with SSE4.2 and flag *-fast* enabled. OpenMP is used to utilise 6 physical cores (12 threads) of the CPU.

For the GPU performance results, we use NVIDIA Tesla C2070 GPU which has 448 cores running at 1.15GHz and has a peak performance at 1288GFlops.

Adaptive vs. Non-adaptive: Table 1 shows the breakdown of computation time using our model and experimental data. Initially, the maximum number of particles are instantiated for global localisation. For the non-adaptive scheme, the particle set size does not change. The total computation time is estimated to be 1.957s which is within the real-time bound. The remaining idle time is enough to reconfigure to sleep mode, since $T_{sleep} = (5 - 1.957 - 0.87)s = 2.173s$.

For the adaptive scheme, the number of particles varies from 70 to 8192, and the computation time scales linearly with the number of particles. Fig. 4 shows how the number of particles varies versus time. Both the model and experiment show 99% reduction in computation time.

Fig. 5 illustrates the localisation error of the mobile robot. The error is the highest during initial global localisation and it is reduced when the robot moves.

Table 1. Comparison of adaptive and non-adaptive PF on HRS (configuration with 1 FPGA kernel is used)

	Non-adaptive		Adaptive	
	Model	Exp.	Model	Exp.
No. of particles	8192		70-8192	
Kernel time T_{kernel} (s)	0.671	0.671	0.006-0.671	0.006-0.671
Host time T_{host} (s)	0.212	0.212	0.002-0.212	0.002-0.212
IO time T_{io} (s)	1.074	1.600	0.009-1.074	0.014-1.600
Total comp. time T_{comp}	1.957	2.483	0.017-1.957	0.022-2.483
Comp. speedup (higher is better)	1x	1x	1-115.12x	1-112.86x

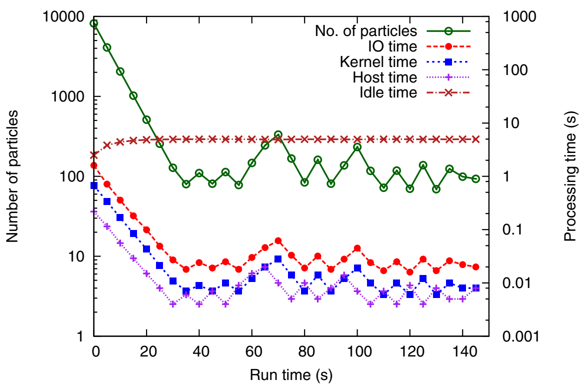


Fig. 4. Variation of particle set size and computation time

It is observed that the localisation error is not adversely affected by reducing the number of particles.

Performance Comparison: Table 2 shows the performance comparison of CPU, GPU and HRS. Considering the kernel computation only, which ignores the IO time and host time, the HRS is up to 19.94 times faster than the CPU, and is 2.65 times faster than the GPU. If the overall system performance is considered, the HRS is up to 3.26 times faster than the CPU, and is 1.74 times slower than the GPU. Meanwhile, the CPU needs 7.002s to process a step, so the real-time constraint of 5s is violated. Currently the performance of HRS is limited by the PCI Express bus between the FPGA and CPU, which has a bandwidth of 2GB/s. If PCI Express gen3 x8 (7.88GB/s) is used, which has comparable bandwidth as that on the GPU, the overall system performance of the HRS is 7.39 times faster than the CPU, and is 1.3 times faster than the GPU.

In real-time applications, we need to consider the energy consumption within the real-time bound. Fig. 6 shows the power consumption varies between computation and idle time, and a significant amount of energy is consumed during idle

Table 2. Comparison of using CPU, GPU and HRS

	CPU [a]	GPU [b]	HRS(1) [c]	HRS(2) [c]
Clock freq. (MHz)	2660	1150	100	100
Number of threads	12	448	1+8 [d]	2+8 [d]
Kernel time (s) [e]	0.058-6.780	0.008-0.892	0.006-0.671	**0.003-0.336**
Kernel speedup	1x	7.53x	10.11x	**19.94x**
Comp. time (s) [e]	0.060-7.002 [f]	0.011-1.236	0.021-2.483 [g] 0.011-1.283 [h]	0.018-2.148 [g] **0.008-0.948** [h]
Overall speedup	1x	5.67x	2.82x [g] 5.46x [h]	3.26x [g] **7.39x** [h]
Comp. power (W)	279	287	**135**	145
Comp. power eff.	1x	0.97x	**2.07x**	1.92x
Idle power (W)	133	208	**95**	**95**
Idle power eff.	1x	0.64x	**1.40x**	**1.40x**
Energy. (J) [e]	674-1954	1041-1138	489-587 [g] 489-539 [h]	489-595 [g] **489-535** [h]
Energy eff.	1x	0.64-1.72x	1.37-3.33x [g] 1.37-3.62x [h]	1.37-3.28x [g] **1.37-3.65x** [h]

[a] Intel Xeon X5650 @2.66GHz with 12 threads.
[b] NVIDIA Tesla C2070 and Intel Core i7-950 @3.07GHz with 8 threads.
[c] Xilinx XC6VSX475T and Intel Core i7-870 @2.93GHz with 8 threads. HRS(1) has one FPGA kernel while HRS(2) has two.
[d] Number of FPGA kernels and number of threads in the CPU.
[e] Cases for 70 and 8192 robot particles.
[f] Real-time bound is violated as the constraint is 5s.
[g] On our platform, the FPGA and CPU communicate through PCI Express with bandwidth 2GB/s.
[h] Assume the FPGA and CPU communicate through PCI Express gen3 x8, bandwidth 7.88GB/s.

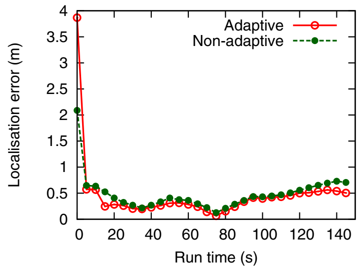


Fig. 5. Localisation error

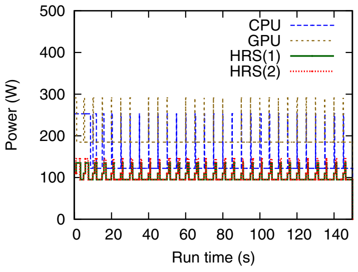


Fig. 6. Power consumption

time. Run-time reconfiguration reduces the idle power consumption of the HRS by 34%, from 135W to 95W. In other word, the energy consumption is reduced by 26-34%. For the case of 8192 particles, the HRS is up to 3.65 times more energy efficient than the CPU, and is 2.13 times more energy efficient than the

GPU. For the case of 70 particles, the HRS is 1.37 times more energy efficient than the CPU, and is 2.13 times more energy efficient than the GPU.

6 Conclusion

This paper presents an adaptive particle filter for real-time applications. The proposed heterogeneous reconfigurable system demonstrates a significant reduction in power and energy consumption compared with the CPU and the GPU. The adaptive particle filter reduces computation time while maintaining quality of results. Ongoing and future work includes applying the adaptive approach to larger systems with multiple FPGAs. Distributed resampling and data compression techniques are explored to mitigate the data transfer overhead between the FPGA and the CPU. More compute-intensive applications using PF would be of interest on the extended heterogeneous reconfigurable system.

Acknowledgment. This work is supported in part by the European Union Seventh Framework Programme under grant agreement number 248976, 257906 and 287804, by UK EPSRC, by Maxeler University Programme, by Xilinx, and by the Croucher Foundation. The author would like to thank Oliver Pell at Maxeler Technologies for valuable comments on the paper.

References

1. Happe, M., et al.: A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking. Journal Real-Time Image Processing, 1–16 (2011)
2. Montemerlo, M., et al.: Conditional particle filters for simultaneous mobile robot localization and people-tracking. In: Proc. IEEE Int. Conf. Robotics and Automation, pp. 695–701 (2002)
3. Vermaak, J., et al.: Particle methods for bayesian modeling and enhancement of speech signals. IEEE Trans. Speech and Audio Processing 10(3), 173–185 (2002)
4. Eele, A., Maciejowski, J.: Comparison of stochastic optimisation methods for control in air traffic management. In: Proc. IFAC World Congress (2011)
5. Doucet, A., et al.: Sequential Monte Carlo methods in practice. Springer (2001)
6. Bolic, M., et al.: Resampling algorithms and architectures for distributed particle filters. IEEE Trans. Signal Processing 53(7), 2442–2450 (2005)
7. Koller, D., et al.: Using learning for approximation in stochastic processes. In: Proc. Int. Conf. Machine Learning, pp. 287–295 (1998)
8. Fox, D.: Adapting the sample size in particle filters through kld-sampling. IEEE Trans. Robotics 22(12), 985–1003 (2003)
9. Park, S.-H., et al.: Novel adaptive particle filter using adjusted variance and its application. Int. Journal Control, Automation, and Systems 8(4), 801–807 (2010)
10. Bolic, M., et al.: Performance and complexity analysis of adaptive particle filtering for tracking applications. In: Proc. Asilomar Conf. Signals, Systems and Computers, vol. 1, pp. 853–857 (2002)
11. Chau, T.C., et al.: Adaptive sequential monte carlo approach for real-time applications. In: Proc. Int. Conf. Field Programmable Logic and Applications, pp. 527–530 (2012)
12. Liu, Z., et al.: Mobile robots global localization using adaptive dynamic clustered particle filters. In: IEEE/RSJ Int. Conf. Intelligent Robots and Systems, pp. 1059–1064 (2007)

Hardware Acceleration of Genetic Sequence Alignment

J. Arram[1], K.H. Tsoi[1], Wayne Luk[1], and P. Jiang[2]

[1] Department of Computing, Imperial College London, United Kingdom
[2] Department of Chemical Pathology, The Chinese University of Hong Kong, China

Abstract. Next generation DNA sequencing machines have been improving at an exceptional rate; the subsequent analysis of the generated sequenced data has become a bottleneck in current systems. This paper explores the use of reconfigurable hardware to accelerate the short read mapping problem, where the positions of millions of short DNA sequences are located relative to a known reference sequence. The proposed design comprises of an alignment processor based on a backtracking variation of the FM-index algorithm. The design represents a full solution to the short read mapping problem, capable of efficient exact and approximate alignment. We use reconfigurable hardware to accelerate the design and find that an implementation targeting the MaxWorkstation performs considerably faster and more energy efficient than current CPU and GPU based software aligners.

1 Introduction

DNA contains a long sequence of pairs of nucleotide bases which can be abstracted into a character string with an alphabet {'A', 'C', 'G', 'T'}. DNA sequencing is the process of identifying the order of the nucleotide bases in a DNA molecule. This process has been utilised in a wide range of applications; for example in medicine, analysis of the genetic information of a patient can be used in diagnosing hereditary diseases.

Next-generation sequencing (NGS) machines are able to rapidly and inexpensively produce sequenced data. To improve the throughput and measurement accuracy of these machines, shorter sequences are processed, allowing tens of billions of bases to be sequenced per day. These short sequences can be created by breaking the long DNA chain randomly. As a consequence of this action, the position and orientation information of the fragments with respect to the sample is lost. Based on the assumption that all DNA sequences within a species are similar, the sample DNA can be reconstructed by determining the location of the short fragments (the short reads) in a known reference genome of the species. An aligner system is used to find the possible positions of these short reads in the reference DNA. The performance of NGS machines has recently been improving at a rate faster than Moore's law. Large computer clusters are often used to process short read data generated by a single sequencing machine. However, the processing speed of computer clusters does not grow as fast as the speed of sequencing machines. As a result the performance of a software based aligner is usually the bottleneck of a bioinformatic analysis flow.

FPGA technology is a promising candidate for accelerating this application, which involves highly-parallel bit-oriented operations. However, irregular computation patterns can affect the efficiency of FPGA accelerators. Most hardware designs overcome this problem by running large portions of the alignment algorithm on a CPU. For example, in hardware designs based on Smith-Waterman local alignment, the traceback step is either run on a CPU or ignored. As a result the performance and functionality of these aligner designs are significantly reduced.

In this paper, we propose a hardware accelerated alignment processor based on a backtracking FM-index algorithm. The design represents a full solution to short read mapping, capable of both exact and approximate alignment. The design can be fully mapped into hardware, maximising the hardware efficiency, while reducing the runtime and costs. The major contributions of our work include:

- A hardware design for a novel sequence alignment processor based on a backtracking FM-index algorithm. Various optimisations, such as those for memory size, memory bandwidth and latency and are discussed (Section 3).
- An implementation of the proposed design, showing how the design can be realised on the Maxeler MAX3 board [1] (Section 4).
- Performance evaluation of the proposed design, with comparisons against some of the fastest software solutions on multi-core processors and GPUs, as well as hardware solutions on FPGAs (Section 5).

2 Background and Related Work

FM-Index. Our design is based on FM-index [2], a data structure that has inspired several software tools for analysing genetic sequences such as Bowtie [3] and SOAP2 [4]. This index combines the properties of suffix array (SA) with the Burrows-Wheeler transform (BWT) [5] , to provide an efficient method for finding all occurrences of a pattern within a long reference sequence. First the BWT of the reference sequence is generated using the following four steps:

1. Terminate the reference sequence R with a unique character: \$, which has the smallest lexicographical value.
2. Generate all rotations of R.
3. Sort the rotations lexicographically.
4. The BWT sequence is the last column of all the entries in the sorted list.

Table 1(a) illustrates an example of generating the BWT for the reference sequence R = ACTAGCTA. The character strings preceding the '\$' symbol in the sorted rotations column form a SA, which indicates the the starting position of each possible suffix in R.

After generating the BWT sequence and the SA representation of the reference sequence, the functions $c(x)$ and $s(x, i)$ are defined. $c(x)$ (frequency) is the number of symbols in the BWT sequence that are lexicographically smaller than x and $s(x, i)$ (occurrence) is the number of occurrences of the symbol x in the BWT sequence from the 0^{th} position to the i^{th} position. These functions are usually implemented as lookup tables using an array structure. Table 1(b) illustrates the $c(x)$ and $s(x, i)$ functions for the reference sequence R = ACTAGCTA.

Table 1. (a) BWT generation and SA representation of reference sequence R. **(b)** Values of functions $c(x)$ and $s(x, i)$ functions for the reference sequence R.

(a)

R = ACTAGCTA		
Index	Sorted Rotations:	SA
0	$ACTAGCTA	8
1	**A**$ACTAGCT	7
2	**AGCTA**$ACT	3
3	**ACTAGCTA**$	0
4	**CTA**$ACTAG	5
5	**CTAGCTA**$A	1
6	**GCTA**$ACTA	4
7	**TA**$ACTAGC	6
8	**TAGCTA**$AC	2
$BWT(R)$ = ATT$GAACC		

(b)

$BWT(R)$ = ATT$GAACC					
$s(x, i)$	x				
i	$	A	C	G	T
0	0	1	0	0	0
1	0	1	0	0	1
2	0	1	0	0	2
3	1	1	0	0	2
4	1	1	0	1	2
5	1	2	0	1	2
6	1	3	0	1	2
7	1	3	1	1	2
8	1	3	2	1	2
$c(x)$	0	1	4	6	7

The Suffix Array (SA) interval of a pattern Q is defined as $[k, l]$. The pointers k and l are respectively the smallest and largest indices in the suffix array which starts with Q. To search for a pattern Q within a reference sequence, k and l are initialised to the first and last indices of the suffix array table respectively. Using equations 1 and 2 the SA interval is updated for each character in Q, moving from the last character to the first (a backwards search).

$$k_{new} = c(x) + s(x, k_{current} - 1) \tag{1}$$

$$l_{new} = c(x) + s(x, l_{current}) - 1 \tag{2}$$

Figure 1 shows an example of searching the pattern Q = GCT in the reference sequence $R =$ ACTAGCTA. First, k and l are initialised to 0 and 8 respectively. Then equations 1 and 2 are applied three times, corresponding to the number of characters in Q. After the third iteration, k and l both become 6. Since $k \leq l$, the pattern can be found in the reference sequence. Note that if $k > l$ for an iteration, the symbol cannot be aligned to the reference sequence (a mismatch). The suffix array elements corresponding to each index within the SA interval give the location of the pattern in the reference sequence. Table 1(a) indicates that index 6 maps to position 4 in the reference sequence. Notice

R = ACTAGCTA $\quad$ Q = GCT

1st iteration: x = T	2nd iteration: x = C	3rd iteration: x = G
k_{new} = c(T) + s(T, -1)	k_{new} = c(C) + s(C, 6)	k_{new} = c(G) + s(G, 3)
= 0 + 7 = 7	= 4 + 0 = 4	= 6 + 0 = 6
l_{new} = c(T) + s(T, 8) - 1	l_{new} = c(C) + s(C,8) - 1	l_{new} = c(G) + s(G, 5) - 1
= 2 + 7 - 1 = 8	= 4 + 2 - 1 = 5	= 6 + 1 - 1 = 6

$\therefore$ SA interval = [6, 6]

Fig. 1. Example of searching the pattern Q = GCT in the reference sequence R

that the time complexity for finding *all* the matching locations is linear in length of the pattern and independent of the length of the reference sequence.

Related Work. There are several hardware accelerators for genetic sequence analysis. In [6], an FM-index based algorithm is proposed for FPGAs. It concludes that using a single large table for FM-index is more area efficient than splitting into multiple smaller tables. The performance is 1000 reads in 60.2 μs if mismatches are not allowed. In [7], a short read mapper is developed using a direct comparison approach. A single LUT is used to compare 2 bases from a streaming reference sequence and a stationary short read. Using a Xilinx XC6V-LX550T FPGA, the system achieves 500000 reads in 212 seconds (i.e. 2.36K reads per second). In [8], the short read alignment is performed by CPU and FPGA collaboratively. The indexing part is performed by CPU and then the short reads, as well as the corresponding reference segments, are sent to the FPGA for pairwise matching. The design achieves around 16 million reads in 110 seconds (i.e. 145.4K reads per second) using a Xilinx XC5V-LX330 FPGA. In [9], an FPGA based short read alignment accelerator is proposed based on indexing of the reference sequence, with Smith-Waterman alignment performed in FPGA. The main optimisation in this work is to reduce the size of the candidate alignment location (CAL) lookup table. The system, and also the CAL table, is partitioned into 8 Pico M-503 boards each with one XC6V-LX240T FPGA. This 8-FPGA system can map 50 million short reads in 34 seconds. Our work differs from previous designs since it is the first implementation of a hardware accelerated short read aligner based on a backtracking version of the FM-index. The proposed design represents a full solution to the short read mapping problem, capable of efficient exact and approximate alignment.

3 Alignment Processor Design

We propose an alignment processor design with the following features:

- A backtracking version of the FM-index algorithm with a data structure that supports both forward and backward search, which is capable of exact and approximate alignment (Section 3.1).
- A novel scheme to reduce the memory size of the FM-index occurrence array, allowing it to be stored directly on the accelerator board (Section 3.2).
- A new method to reduce external memory access frequency, while maximising memory bandwidth utilisation (Section 3.3).
- A novel scheme to process batches of short reads in parallel, maximising the throughput of the design (Section 3.4).

3.1 Backtracking Design

As discussed in Section 2, FM-index provides a method for finding all the occurrences of a pattern within a reference sequence. By incorporating backtracking into FM-index, the method is extended to allow for permutations of the pattern to be matched to the reference sequence. This allows the detection of Single Nucleotide Polymorphisms (SNPs), which represent a mismatch in a single nucleotide between the short read and reference sequence. The alignment processor supports the approximate alignment of short reads to a reference sequence within a permitted number of mismatches. This is a depth first search, in which the algorithm attempts to align each symbol in the short read to the reference sequence. When a symbol in the short read cannot be aligned (a mismatch) a different symbol is attempted. Only when the number of mismatches exceeds the permitted number does the algorithm backtrack to the previous symbol and attempts a different one. In the proposed design, only the first alignment solution (if any) is reported. The algorithm can be revised to report more solutions at the expense of performance.

The backtracking FM-index algorithm is extensively used in software aligners; however its use in hardware aligners is largely unexplored due to the complex mapping process. Backtracking algorithms are typically expressed as recursive functions in software. In hardware the circuit represents the entire computation, therefore recursive features such as branching statements are difficult to implement. We overcome this difficulty by using an iterative approach, where a stack is used to store the nodes in the current search path. For the FM-index, the items that require storage in a stack are:

– The SA interval indices k and l.
– The current number of mismatches.
– The next symbol to attempt if matching fails.

For a short read of length $|Q|$ and m permitted mismatches, the storage requirement for the stack is:

$$(2 \times |Q| \times 32bits) + (|Q| \times log_2(m)bits) + (|Q| \times 2bits)$$

For example, the storage requirement is 850 bytes when $|Q| = 100$ and $m = 4$. The pseudo code in Figure 2 describes the stack operations for one iteration of the backtracking FM-index algorithm.

As discussed in Section 2, for exact pattern matching the alignment processor is run for the same number of iterations as symbols in the short read. For approximate matching, the number of iterations to run the processor for is non-trivial and depends on the short read length, the number of mismatches and their position within the short read. The position of the mismatches has the greatest impact on the number of iterations. For example, if the mismatches are at the end of the short read, a large amount of the search space must be explored, requiring a large number of iterations. The challenge here is to reduce the number of iterations required for approximate matching, thus improving the design performance. We address this challenge by adapting the bi-directional BWT [10] for our purpose. This data structure allows searching in both directions (forward and backward search) and allows switching in search direction during the alignment of a pattern. As a result the impact of the mismatch position is minimised since the search can proceed in both directions. In this approach the BWT of the reversed reference sequence and the corresponding occurrence array, $s[x][i]$, are stored, doubling the external memory requirement. In the proposed design, the short reads are processed in stages. First, all the short reads are tested for exact alignment, then the unaligned reads are tested for one mismatch using a larger number of iterations (the average number

```
// backtrack condition
if number of mismatches > permitted mismatches
or all symbols in current node attempted
      pop stack;

// get symbol for alignment
if previous symbol successfully aligned
      get symbol from short_read;
else
      get symbol from stack and update top element to next symbol;

get k and l from top of stack;

// FM-index computation (update k and l)
apply equations (1) and (2)

if symbol successfully aligned and symbol from stack
    increment number of mismatches;

if symbol successfully aligned
      push k, l and number of mismatches on stack;
```

Fig. 2. Pseudo code describing one iteration of the backtracking FM-index algorithm

of iterations required for the number of mismatches and short read length). This reprocessing of the unaligned reads is continued until the maximum permitted number of mismatches is reached. Since approximately 70-80% of short reads in a typical data set can be exactly matched to the reference sequence, and over 90% is covered by allowing one mismatch, this method stops the processing of short read data from becoming the bottleneck.

3.2 Memory Size Optimisation

The occurrence array, $s[x][i]$, stores the number of occurrences of the symbol x in the BWT sequence from the 0^{th} position to the i^{th} position. Since human DNA has an alphabet of four symbols in approximately equal proportions, a 32 bit format is required to represent the occurrences. The total size required to store the occurrence array for a 3.2G base genome is $4 \times 3.2G \times 32bits = 51.2$G bytes. This is too large for memories in most FPGA platforms. The challenge here is to reduce the memory footprint of the occurrence array such that a) it can fit on most FPGA board memories and b) multiple independent copies of this array can be associated to multiple alignment processors.

We address this challenge by storing a full range value as a marker for every d elements in the occurrence array. To reconstruct the occurrence value of $s[x][i]$, the following components are summed: 1) the lower marker value relative to position i and 2) the result from counting the occurrence of symbol x in the BWT sequence between the lower marker position and i. In this approach we store the marker values and the BWT sequence. The required memory storage is then signifcantly reduced to

$$4 \times \frac{3.2G \times 32bit}{d} + 3.2G \times 2\ bits$$

For example, the memory footprint is reduced to 1.6G bytes when d = 64.

This technique is used in software aligners such as SOAP2 [4], but its use in hardware designs is largely unexplored. The cost of using this technique is having to count the occurrence value directly from the BWT sequence. Counting the occurrences sequentially increases the latency of the design by $d \times l$, where l is the latency of a single add operation. For large values of d this additional latency reduces the design performance. We can make use of the inherent parallelism of FPGAs to count the occurrences in parallel using a binary adder tree. As a result the additional latency of the design is reduced to $\log_2 d \times l$.

Having a larger d value will further reduce the memory footprint. However, the rate of reduction rapidly decreases as the marker array becomes smaller than the BWT sequence. On the other hand, the hardware resources and the latency of counting process increase when the value of d becomes larger. The architecture allows users to trade-off available external memory storage and available on-chip computation resources.

3.3 Memory Bandwidth Optimisation

By the nature of the FM-index algorithm, the indexing pattern to the occurrence array, $s[x][i]$, is random. So it is difficult to further reduce external memory access frequency by caching. Applying the memory size optimisation, each indexing iteration requires two 32-bit reads for the marker values and two 128-bit reads for the d-base sequence segment. Many FPGA based accelerators place a limit on the number of connections from the FPGA to external memory. The number of alignment processors that can populate the FPGA is therefore limited by the number of connections to external memory required by an alignment processor. Furthermore, many FPGA based accelerators, such as the Maxeler MAX3 board, support burst memory access such that hundreds or thousands of bits can be accessed from memory in a single read operation. Initial processor designs extract only the desired bits, discarding the remaining bits in the burst. For large burst sizes, this method of memory access results in poor utilisation of memory bandwidth. The challenges here are to: a) reduce the number of connections to external memory in order to allow a larger population of processors per FPGA and b) appropriately layout the data in external memory in order to make efficient use of the available memory bandwidth.

We address this challenge by interleaving the marker array and BWT sequence such that the occurrence array markers and the corresponding BWT sequence segments are grouped together in external memory. By interleaving, both the relevant marker and corresponding BWT sequence segments can be accessed in a single memory access, reducing memory access frequency and external memory connections by 50%. Furthermore, by adjusting the value of d, the size of interleaved segments can be optimised for burst access. For example, by adjusting d so that one interleaved segment is exactly the size of a burst, the memory bandwidth is fully utilised in each read operation.

3.4 Latency Optimisation

For a human DNA sequence, the occurrence array, $s[x][i]$, is so large that it must be stored in memory outside the FPGA device. The access to external memory usually increases the latency by many cycles. By the nature of FM-index, the computation for each iteration of the algorithm is dependent on the results from the previous iteration. This iteration interdependence, coupled with the design latency, creates a sub-optimal pipeline. The challenge here is to minimise the effect of the latency to ensure design performance is not limited by external memory access.

We address this challenge by interleaving the processing of multiple short reads. In this approach, the alignment processor contains a buffer storing a batch of short reads. For a design latency l, the buffer is able to store l short reads. In each clock cycle a symbol from a different short read is selected and propagates through the pipeline. After l cycles the result is available and the next symbol in the short read is processed. As a result, the design is fully pipelined with a throughput of one aligned base per cycle. This design achieves the equivalent of a multi-threaded program aligning multiple short reads in parallel, but with zero thread switching overhead.

This latency optimisation comes at the cost of BRAM resources required to store the additional short reads and stacks. By using only 2 bits to represent the four valid symbols in the DNA sequence rather than the standard 8-bit ASCII characters, the additional BRAM resources required for the short reads can be reduced. Furthermore, the traffic between the host and the FPGA accelerator is significantly reduced.

4 Design Architecture

Our design architecture consists of an FPGA populated with alignment processors, connected to the host processor through a software driver. Before alignment starts, the software driver transfers the BWT sequence, $s[x][i]$ and $c[x]$, to the accelerator board, where they are stored using on-chip BRAM or external DRAM. Short reads are then streamed to the alignment processors in batches given by the design latency. Each batch of short reads is processed for a number of iterations determined by the permitted number of mismatches and the short read length. The alignment results for each short read, including the SA interval, the cost and a string representation of the alignment are reported to the software driver. This architecture is illustrated in Figure 3.

Software Driver. The alignment processor design can be fully mapped to hardware, therefore the software driver has a minimal role in the design architecture. The reference sequence changes infrequently, therefore we assume that the BWT sequence, $s[x][i]$ and $c[x]$ are generated in advance. After transferring the data structures to the accelerator board and configuring the number of mismatches permitted, the short reads are streamed to the alignment processors populating the FPGA. The software driver then pauses until all the accelerator output is received. If a short read can be matched to the reference sequence the SA interval is mapped to positions in the reference sequence using a simple lookup table. This step can be performed in hardware, however we choose to perform it using a CPU in order to reduce the number of memory controllers required by each alignment processor. If a short read is unaligned it is streamed again to the alignment processors for testing with a higher number of permitted mismatches.

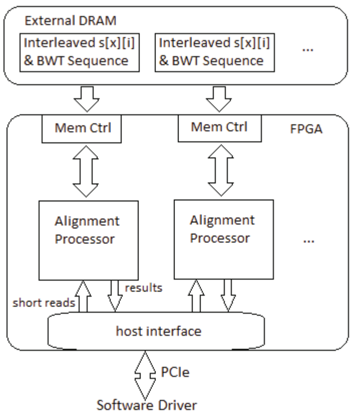


Fig. 3. Design architecture

Hardware Circuit. The FPGA is populated with alignment processors based on the backtracking FM-index algorithm described in Section 3.1. The challenge of designing the hardware circuit is mapping the iterative backtracking algorithm into hardware.

We address this challenge by implementing the stack using an array structure. In this approach, an array is used to hold the contents of the stack and the top of the stack is tracked using a pointer. The four stack items are stored in separate arrays using on chip BRAM. To implement the latency optimisation presented in Section 3.4, each array must store the stacks for l short reads, where l is the design latency. In our implementation the design latency is 58 cycles, therefore the storage requirement for the stacks is 50K bytes per processor. The pointer tracking the top element of the stacks (top), is implemented as a circular buffer using shift registers, so that its value propagates into the subsequent iteration. To push an item on the stack, top is incremented and to pop the stack top is decremented.

Using the memory size optimisation presented in Section 3.2, the occurrence array is stored in external memory with the marker values for every 64 bases. A binary tree adder is implemented to minimise the additional design latency from counting the occurrence directly from the BWT sequence. Without this optimisation the performance of the design would be reduced, since the BRAM resources required to store the additional short reads and stacks would limit the number of alignment processors able to populate the FPGA.

We implement the FM-index circuit developed in [6] for our design. The SA interval, cost and string representation of the alignment are output to the software driver when one of the following conditions is true: 1) the processor has run for the number of iterations determined by the number of mismatches permitted, 2) the entire search space has been explored and no alignment is found, or 3) an alignment solution is found. After each short read in the batch has been processed for the appropriate number of iterations, a new batch of short reads is streamed from the software driver. When all short reads have been processed, the hardware execution halts, returning control to the software driver.

5 Evaluation

In this work, we use the Human Genome version 18 [11] and short reads sampled directly from the reference sequence. We insert noise in random positions in the short reads to simulate mismatches between the short read and reference sequence. We implement the proposed design on the Maxeler MaxWorkstation containing a MAX3 dataflow engine (DFE). The MAX3 contains a Xilinx Virtex-6 SX475T FPGA and 24GB of fast DRAM. The system can also support up to 15 independent memory controllers. In this platform a design is described using the MaxJ language, an extension of the Java language. The MaxCompiler then maps the design into an FPGA implementation and enables its use from a host application. Table 2 shows the resource usage for the proposed design with three alignment processors implemented in the MaxJ language. The implementation supports up to 100 bases for a single short read.

Table 2. Resource usage

Design	Registers	LUTs	BRAM	DSP
Proposed Design	20%	25%	53%	0%

The resource usage values indicate that BRAM is the critical resource in the proposed design. The maximum number of alignment processors able to populate the FPGA is seven, since a) each alignment processor uses two memory controllers and the platform supports a maximum of 15 memory controllers per FPGA and b) the resource usage values for a single alignment processor, including the resources for place and route, indicate that only 7 alignment processors can fit on the FPGA, due to the large number of BRAM resources used.

Since different packages report their performance using different data sets, it is difficult to directly compare designs using the raw results. To better assess the performance of various designs, we define the bases aligned per second ($baps$) value as a normalised performance merit.

$$baps = \text{read size} \times \text{read count}/\text{process time}$$

In Table 3 we compare the performance of our design implemented on the MAX3 board with existing software and hardware designs. In all our testing we process short reads with 76 bases and allow up to 2 mismatches between the short read and reference sequence.

The $baps$ values in Table 3 indicate that the proposed design is faster than current software aligners. It is over 7 times faster than BWA on an Intel X5650 CPU and 3.5 times faster than SOAP3 on an NVDIA GTX 580 GPU. To make a fair comparison to the design in [9], we estimate the performance of the proposed design implemented on theMaxeler MPC-X Series rackmount system (8 FPGA devices). The $baps$ values in Table 3 indicate that the proposed design has a comparable performance to the design in [9], even with fewer cores and a lower clock frequency. With further work into the placement of the alignment processors on the FPGA, we could reach the upper bound population and exceed the performance of the design in [9].

Table 3. Aligner performance comparison

Design	Platform	Clock freq (MHz)	Devices	Cores	*baps* (millions)
Bowtie [3]	Intel X5650	2670	1	20	1.04
BWA [12]	Intel X5650	2670	1	20	1.76
SOAP2 [4]	Intel Xeon X5650	2670	1	20	1.59
SOAP3 [13]	NVIDIA GTX 580	900	1	512	3.84
Proposed Design on MaxWorkstation	Xilinx Virtex-6 SX475T	150	1	3	13.5
Design in [9]	Xilinx Virtex-6 LX240T	250	8	8	112
Proposed Design on MPC-X (estimate)	Xilinx Virtex-6 SX475T	150	8	3	108

The energy consumption of our design is estimated using the XPower utility offered by Xilinx. It allows power analysis and estimations based on FPGA resource usage and configuration. In Table 4 the energy consumption of the proposed design is compared with existing software and hardware designs when mapping 50 million short reads.

Table 4. Energy consumption comparison

Design/Platform	Energy (W-hr)
Bowtie	19
BWA	11
SOAP2	13
SOAP3	6.3
Design in [9]	5.0
Proposed Design on MaxWorkstation (3 cores)	0.078

The values in Table 4 indicate that the proposed design consumes significantly less energy than current software aligners. This is a result of the system only drawing a small amount of power (7W) coupled with a much shorter runtime.

In addition to runtime and energy efficiency, another important attribute of an aligner is sensitivity - the percentage of short reads able to be successfully mapped to the reference sequence.

For up to two mismatches our design has a comparable sensitivity ($\sim$100%) to current software aligners. For short reads with more than two mismatches, the sensitivity of the software aligners sharply decreases ($<$20%). This is a result of the aligner being unable to explore the large search space within the cut off time.

Since the number of iterations for which the alignment processors are run for is flexible and the performance of the proposed design is better than the software aligners, we are able to process short reads for a larger number of iterations and report better sensitivities than the software aligners. To estimate the appropriate number of iterations to run the approximate matching for, we measure the average number of iterations required to align short reads of a specific length with a specific number of randomly distributed mismatches. Adjustments can be made according to the desired sensitivity.

6 Conclusion

In this work we demonstrate that an alignment processor based on the backtracking FM-Index algorithm can achieve high throughput for short read alignment applications. The design is able to map short reads to a full genome faster than current software aligners, without compromising the alignment sensitivity. Furthermore, it consumes significantly less power than software aligners, making it a feasible alternative to computational clusters. Current and future research includes further optimisation of our approach, and its application in clinical procedures.

Acknowledgements. We thank Dennis Lo and Rossa Chiu for their timely assistance and encouragement in this area of research. We also thank Maxeler University Programme, particularly Oliver Pell, and Xilinx for their support. Thanks also to ESPRC for funding our work.

References

1. http://www.maxeler.com/products/
2. Ferragina, P., Manzini, G.: An experimental study of an opportunistic index. In: ACM-SIAM Symposium on Discrete Algorithms (SODA 2001), pp. 269–278 (2001)
3. Langmead, B., et al.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology 10(3), R25+ (2009)
4. Li, R., et al.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics 25(15), 1966–1967 (2009)
5. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Digital Equipment Corporation, Tech. Rep. (1994)
6. Fernandez, E., Najjar, W., Lonardi, S.: String matching in hardware using the FM-index. In: Proc. FCCM, pp. 218–225 (2011)
7. Preuer, T.B., Knodel, O., Spallek, R.G.: Short-read mapping by a systolic custom FPGA computation. In: Proc. FCCM, pp. 169–176 (2012)
8. Tang, W., et al.: Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator. In: Proc. FCCM, pp. 184–187 (2012)
9. Olson, C.B., et al.: Hardware acceleration of short read mapping. In: Proc. FCCM, pp. 161–168 (2012)
10. Lam, T., Li, R., Tam, A., Wong, S., Wu, E., Yiu, S.: High throughput short read alignment via bi-directional bwt. In: IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pp. 31–36 (November 2009)
11. http://hgdownload.cse.ucsc.edu/goldenpath/hg18/chromosomes/
12. Li, H., Durbin, R.: Fast and accurate short read alignment with burrows wheeler transform. Bioinformatics 25(14), 1754–1760 (2009)
13. Liu, C.-M., et al.: SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads. Bioinformatics 28(6), 878–879 (2012)

An FPGA Acceleration for the Kd-tree Search in Photon Mapping

Takuya Kuhara[1], Takaaki Miyajima[1], Masato Yoshimi[2], and Hideharu Amano[1]

[1] Graduate School of Science and Technology, Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama-shi, Kanagawa-ken 223-8522, Japan
[2] Graduate School of Informatics Systems,
The University of Electro-Communications,
1-5-1 Chofugaoka, Chofu, Tokyo 182-8585, Japan
kuhataku@am.ics.keio.ac.jp

Abstract. Photon mapping is a kind of rendering techniques which enables depicting complicated light concentrations for 3D graphics. Searching kd-tree of photons with k-near neighbor search (k-NN) requires a large amount of computations. As k-NN search includes high degree of parallelism, the operation can be accelerated by GPU and recent multi-core microprocessors. However, memory access bottleneck will limit their computation speed. Here, as an alternative approach, an FPGA implementation of k-NN search operation in kd-tree is proposed. In the proposed design, we maximized the effective throughput of the block RAM by connecting multiple Query Modules to both ports of RAM. Furthermore, an implementation of the discovery process of the max distance which is not depending on the number of Estimate-Photons is proposed. Through the implementation on Spartan6, Virtex6 and Virtex7, it appears that 26 fundamental modules can be mounted on Virtex7. As a result, the proposed module achieved the throughput of approximately 282 times as that of software execution at maximum.

Keywords: FPGA, Photon Mapping, kd-tree, Acceleration, k-NN.

1 Introduction

Photon mapping is a famous computer graphics methods proposed by Jensen as an extension of the ray-tracing[1] to represent complex light environments. Since the method requires large amount of computation which includes high degree of parallelism, various acceleration methods have been researched[2][3][4][5].

Since the photon mapping repeatedly executes K-nearest neighbor search(k-NN) by the number of pixels, the time to generate a 3D image is much influenced by the throughput of k-NN. Generally the computational cost of k-NN is reduced by a space-partitioning data structure called kd-tree[6] which organizes points in a k-dimensional space. The kd-tree is adopted by various applications including k-NN such as intersection computation in ray-tracing [7] [8], clustering images[9] [10], searching information [11], and machine learning[12]. Even though iterations of k-NN with kd-tree can be executed in parallel because they are independent of each other, frequency of memory access is increased to obtain the data corresponding to the node of kd-tree. The acceleration techniques for multi-core microprocessors and graphics processors will be suppressed by the capacity of cache memory which is generally smaller than the data size of kd-tree. On the other hand, dedicated hardware by FPGA may be a useful solution by following two reasons; (1) parallelized and pipelined memory architecture hides the memory access latency by a lot of the distributed small memories and (2) flexible design in response to the target application with various parameters.

In this paper, we propose a k-NN implementation using an FPGA, and evaluate that performance with some FPGA families. The rest of this paper is organized as follows. Section 2 overviews the photon mapping and kd-tree. Section 3 discusses relationship between the computation time of k-NN with kd-tree and the parameters of searching. Section 4 is about design and implementation of the k-NN for FPGA. Section 5 describes performance evaluation. Finally, Section 6 summarizes the paper with conclusion.

2 Photon Mapping and Kd-tree

2.1 Photon Mapping Algorithm

Photon mapping is a technique to generate an image which can be view the 3D models from a viewpoint. Even though general ray-tracing method traces only rays from the viewpoint, photon mapping adds the tracing photons from light source. First, a predefined number of *photons* are emitted in random directions from a light source with two data: position and light energy. The emitted photons reach a point in the space after repeating reflection, refraction and diffusion in several times. The reached point is stored in a database called photon map. kd-tree is used as a data structure for photon map. Second, rays are emitted in the space from a viewpoint. Similar to the photon, each emitted ray reaches a point in the space called *query*. The image is generated by searching the photons near the query to estimate color by density of photons and feature of the model. In the rest of paper, the emitted photon is called *Emit-Photon*, and the photon using estimation is called *Estimate-Photon*.

2.2 Kd-tree Algorithm

Kd-tree is a space-partitioning data structure for managing points in a k -dimensional space. Space division is expressed by adding an axis information to each node of tree. Fig. 1 shows how the photons distributed on the xy plane are stored into the tree structure. The photons are numbered 0-6 from left to right. KD-tree is built through circulating building a node of a binary tree based on the axis value. First in Fig. 1, photon 3 is selected to divide the space with left and right based on its value of axis x and stored as the root node of the kd-tree. Similarly, photon 1 is selected in the left space, and left subtree is built based

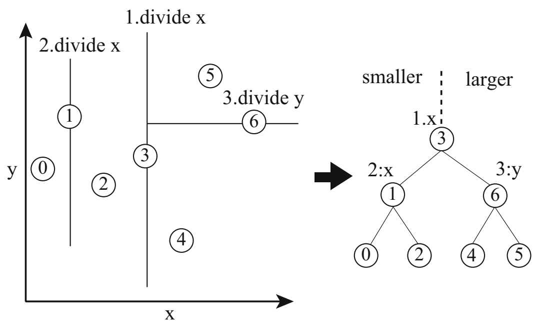


Fig. 1. Stored photon into kd-tree

on its value of axis y. In the right space, photon 6 is selected, and left subtree is rebuilt based on its value of axis x. By these steps, photon(0,2,4,5) become leaf nodes, and manage finer divided spaces.

In photon mapping, photons are aggregated within a certain range of the query by traversing kd-tree from the root node to a leaf node as following steps;

1. Photon p is obtained from kd-tree. At the start of travercing, the root node is obtained.
2. Absolute distance $D1$ between p and query q is calculated.
3. If $D1$ is within a radius R, the p and $D1$ are stored into a list of Estimate-Photons. Then, find the maximum value $D1_{\max}$ from the list. If the list is full and $D1$ is not larger than $D1_{\max}$, the photon with $D1_{\max}$ is overwritten by p and recalculate $D1_{\max}$.
4. If the list is full, R is updated by $D1_{\max}$.
5. The distance $D2$ between p and q on axis recorded in p is calculated.
6. If $D2$ is positive, the right child of p is adopted for the next traversal. Otherwise, the left child of p will be adopted.
7. Repeating steps from 1. to 5. until p reaches a leaf node.

While R is initialized with a large value at the start of traversing, it is updated with $D1_{\max}$ when the list of Estimate-Photon becomes full to avoid unnecessary search.

After reaching a leaf node, kd-tree is traversed again to obtain whether there is a photon closer to the query through following steps;

1. Photon P, which is the parent node of the current node, is extracted from kd-tree. At start of traversing, the current node is the leaf node.
2. Distance $D3$ between P and q with axis recorded in P is calculated.
3. If $D3$ is within R, traverse kd-tree to unsearched direction until leaf node.
4. Repeat steps from 1. to 3. until reaching the root node.

By these two traversing steps sequentially, the list of Estimate-Photon is obtained.

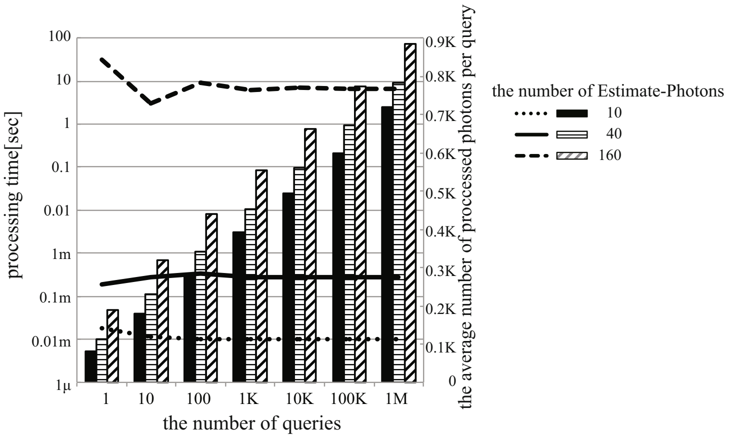


Fig. 2. Processing results versus # of query

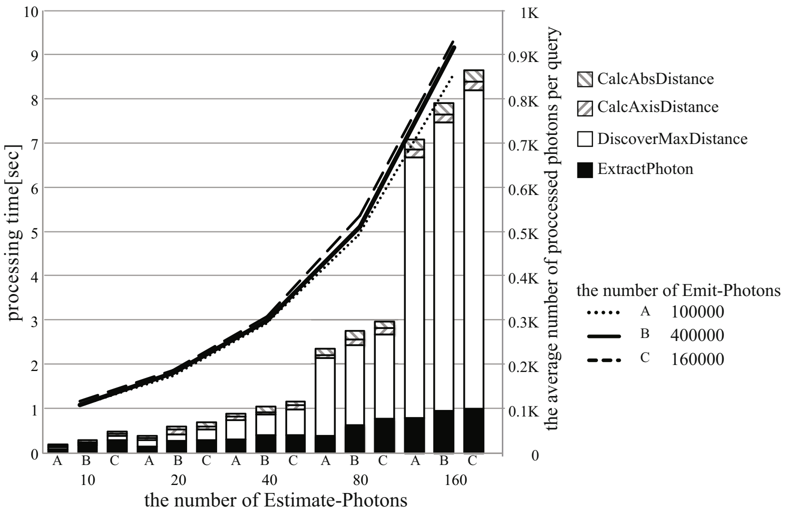


Fig. 3. Processing results versus # of Estimate-Photons and breakdown of processing time of k-NN

3 Profiling of K-NN Search Using Kd-tree

We investigate relationship between processing times and some parameters for k-NN using kd-tree, and breakdown of processing time of k-NN. Three parameters, the number of queries, Emit-photons, and Estimate-photons were selected. The program of kd-tree was written in C, and compiled with option -O3. Table 1 shows the profiling setup. Fig. 2 shows the processing results versus the number of queries. In Fig. 2, the number of Emit-Photons is fixed to be 0.1M, and the number of Estimate-Photons are changed in three patterns. In Fig. 2, the bar chart shows processing time while the line chart shows the average number of processed photons per query. The bar chart shows that the processing time is directly proportional to the number of queries in each pattern. The line chart shows that the average number of processed photons almost keeps constant with some variant. Fig. 3 shows the processing results versus the number of Estimate-Photons and breakdown of processing time of k-NN. In Fig. 3, the number of queries is fixed to be 0.1M, and the number of Emit-Photons are changed in three patterns. In Fig. 3, the bar chart shows processing time while the line chart shows the average number of processed photons per query. Moreover, the stacked column chart shows breakdown of processing time of k-NN. In Fig. 3, k-NN is classified into the four processes as following.

CalcAbsDistance
Calculate the distance between query and photon.
CalcAxisDistance
Calculate the distance between query and photon on axis.
DiscoverMaxDistance
Discovery max distance in the distance list.
ExtractPhoton
Extract the photon from the memory.

The bar chart and line chart show that the processing results are greatly changed when the number of Estimate-Photons increases. On the other hand, these chart also show that the processing results are not so changed, while the number of Emit-Photons increases. Furthermore, the stacked column chart shows that most of the processing time is occupied by the discovery process of max distance.

From these profiling results, it is summarized that the parameter which gives the largest effect to the processing time is the number of Estimate-Photons. In k-NN with kd-tree, after finding the required number of Estimate-Photons, the number of processing photons can be reduced by shortening the search radius. That is, if the number of Estimate-Photons is enough small, most of kd-tree is pruned even with a large number of Emit-Photons, and processed time is not largely increased. On the other hand, if the number of Estimate-Photons is large, the search radius can not be easily shortened resulting in a large processing time. Moreover, the processing time of discovery process of max distance will be increased by increasing the number of Estimate-Photons. In addition, it is

Table 1. Profiling setup

CPU	Core i7 2600K (3.4GHz)
Memory	8GB
OS	Ubuntu 11.04 x86_64
CPU code Compiler	gcc 4.4.5
Language	C++

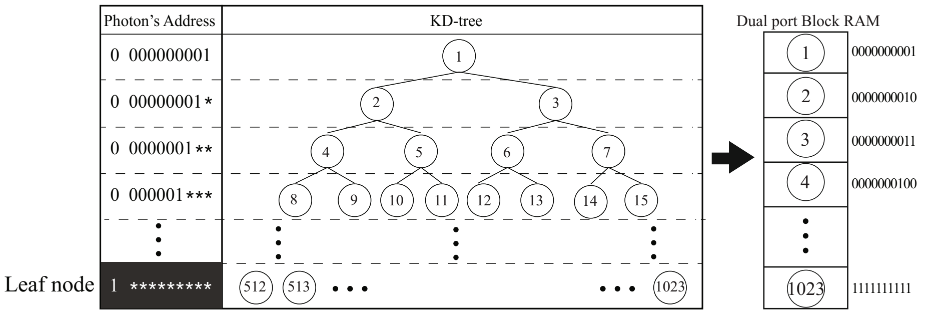


Fig. 4. Kd-tree on an FPGA

summarized that the most of processing time of k-NN is discovery process of max distance. In discovery process of max distance, time complexity increases by $O(N)$ where let the number of Estimate-Photons be N. On the other hand, time complexity of the others processes is $O(1)$ regardless of the parameters.

4 k-NN with kd-tree on an FPGA

4.1 Policies of Implementation

Our target is implementing kd-tree with fundamental k-NN functions on an FPGA. Although our direct target application is the photon mapping, it can be used in other applications. Since the FPGA is used as an off-loading engine, the proposed hardware only manage the k-NN function, and other functions are computed in a host computer connected with the FPGA.

In k-NN with kd-tree, since it is necessary to obtain data from the memory frequently, throughput will be reduced if hardware is implemented to execute memory access and search operations sequentially. Here, the throughput is measured by the number of processed photons per unit time. In order to improve the throughput, maximizing memory access bandwidth is needed. For doing it, the search operations are pipelined, and multiple queries are processed in parallel. In addition, for the discovery process of maximum distance which is easy to increase time complexity, we propose a sophisticated method for storing data in the memory to reduce the time complexity.

4.2 Implementation of kd-tree on an FPGA

Fig. 4 shows an implementation of kd-tree on an FPGA. The photon data is stored into embedded memory (Dualport BlockRAM) on the FPGA. For storing RAM, the right child of the photon, stored in the address i is stored in the address $i \times 2$, and the left child of it is stored in the address $i \times 2 + 1$. The bit width to access kd-tree is fixed at 10 corresponding to the height of the kd-tree as shown in Fig. 4. By this way of storing, we can judge whether the obtained photon is a leaf node by checking the MSB of the address accessed.

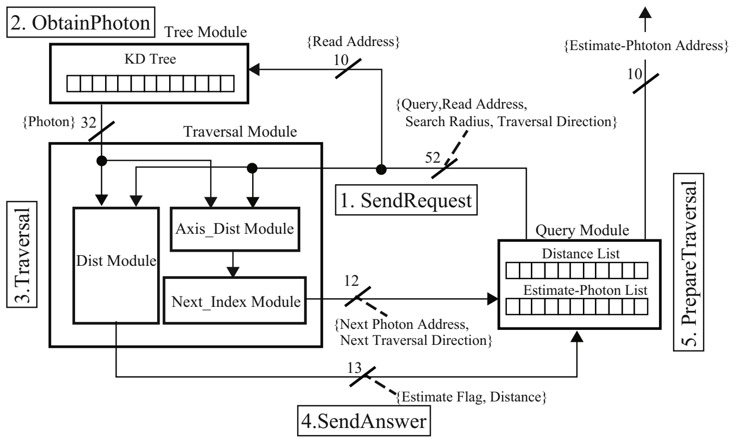


Fig. 5. Fundamental structure of k-NN and steps of traversal

Here, the space of 1000 × 1000 × 1000 is used as the space emitted photons, and so the coordinate information is setted 30 bits. Added with two bits for axis information, 32 bits data is attached to a query. In this experimental implementation, unit size of kd-tree is a 32 × 1024 BlockRAM.

4.3 Fundamental Structure

Fig. 5 shows a fundamental structure minimum functions of k-NN on an FPGA. This circuit is mainly consisting of Query Module, Tree Module, and Traversal Module. The following shows roles of these modules.

Query Module
Query Module is a controller module that manages the process of one query. It has a query data, a Estimate-Photon list, a distance list and traversal parameters. Estimate-Photon list has the address of Estimate-Photons, and the distance list has the distance between photon and query. Each has the depth corresponding to the number of Estimate-Photons.

Tree Module
Tree Module is a module with BlockRAM stored kd-tree. As an initial operation, it receives photon data and builds kd-tree on BlockRAM.

Traversal Module
Traversal Module is a module that traversals kd-tree based on photon, query, and traversal parameters. The traversal is executed through the internal module, Axis_Dist Module, Next_Index Module, and Dist Module. Axis_Dist Module calculates distance between photon and query on axis. Next_Index Module calculates the next traversal parameters including the next photon address from the current parameters and a result of Axis_Dist Module. Dist Module calculates the distance between absolute distance and judges

whether the processed photon is Estimate-Photon by comparing the absolute distance with the search radius.

The following shows how traversal is done in the structure.

1. SendRequest: Query Module sends the read address and the read request (Read Address) of photon to Tree Module on a fixed clock cycle. At the same time, it sends the traversal parameters (Query, Read Address, Search Radius, Traversal Direction) to Traversal Module.
2. ObtainPhoton: When Tree Module receives the request from Query Module, it sends the photon data to Traversal Module on the next clock cycle.
3. Traversal: Traversal Module starts the traversal using received parameters.
4. SendAnswer: The traversal results (Next Photon Address, Next Traversal Direction, Estimate Flag, Distance) are sent to Query Module.
5. PrepareTraversal: Estimate-Photon list and distance list are updated and the next traversal parameters are set in Query Module. If the processed photon is Estimate-Photon, the photon address and the distance will be stored in the list of each. If the list is full, the maximum value of distance list is overwritten by received result. Then, the maximum value of the distance list is calculated, and set for using the next search radius. At the same time, the index of maximum distance is also calculated and saved to use overwriting maximum distance by the next traversal result. After all these processes are finished, the next traversal is started from step 1.

By repeating these steps, traversal of kd-tree for a query is done. It sends Estimate-Photon Addresses from the list after all of search about one query, and starts the search for a new query.

4.4 Acceleration of Discovery Process of Maximum Distance

In the discovery process of max in Query Module, it is needed to compare distances for the number of Estimate-Photons. In photon mapping, time complexity of this process easily increases, since the number of Estimate-Photons easily increases. For instance, for generating high-quality images, more than 50 Estimate-Photons are required. On the other hand, the bit width of distance is not radically increased. For example, if the space size becomes from $1000 \times 1000 \times 1000$ to $10000 \times 10000 \times 10000$, bit width of distance will increase 12 bits to 15 bits. Thus, we try to discover maximum distance by filtering bit sequence of the same digit of each distance instead of comparing each distances. By this method, the maximum distance and its index can be obtained in the number of the steps same as the bit width of the distance. Fig. 6 shows this method. Through the following steps, the maximum distance and its index are obtained.

1. RotateMemory: memory is rotated by 90 degrees, and each bit of distance is stored in each stage of memory. In this way, input distance operation requires clock cycles corresponding to the number of bit width of the distance, however, output bit sequence operation can be done on one clock cycle.

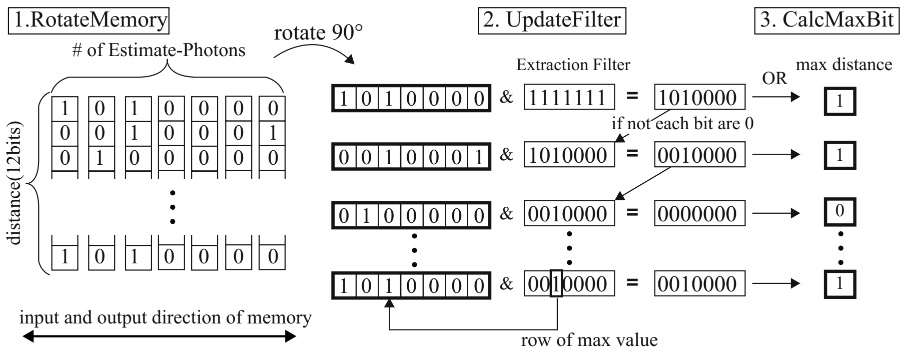


Fig. 6. Storing memory method, and calculation maximum distance and its index. By three steps, maximum distance and its index is obtained.

2. UpdateFilter: The bitwise AND of the Extraction Filter and a bit sequence obtained from N stage of the memory is done. In first operation, N is the top of memory. Extraction Filter is initialized with 1 for each bit. If all bits of a result is not 0, Extraction Filter is updated with the result. Otherwise, it is not updated.
3. CalcMaxBit: The bitwise OR of the result of step 2 is calculated and the result is set to a N digit of maximum distance.
4. Increment N, and from step 2 to 3 are repeated until bottom of memory is calculated.

As a result of these process, the digit of Extraction Filter, the value of which is one, is the position which stores maximum distance in the memory. In Fig. 6, the maximum value is stored in the third from the left in the memory. Furthermore, by step 3, we can obtain the maximum distance during filter update. It means that discovery maximum distance can be done regardless the number of Estimate-Photons. When DualPort BlockRAM is used, these processes are completed in 13 clock cycles.

4.5 Maximizing Memory Access by Multiple Connections Query Module

In the hardware shown in Fig. 5, memory storing kd-tree is not frequently accessed. For example, in the waiting time of Query Module sending results, it is not accessed at all. Thus, we connect multiple Query Modules to Tree Module so that the photon can be read out continuously. Requests to Tree Module, access with Traversal Module and output search result are assigned to each Query Module per each clock cycle via a multiplexer. By making the number of Query Modules same as the clock cycles of request interval of Query Module, the blank of memory access can be filled. As a result of implementation, thirteen clock cycles are needed for discovery maximum distance, five cycles for Traversal Module,

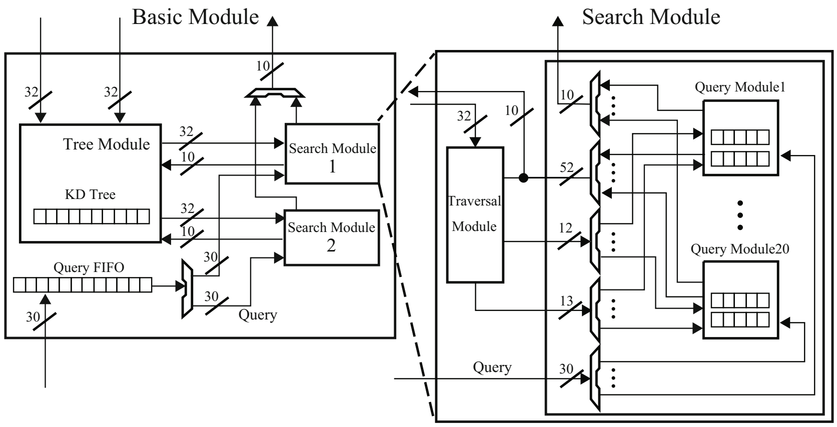


Fig. 7. Maximize memory access by connecting multiple QueryModules to TreeModule. Two Search Modules, which is consist of 20 Query Modules and one Tracing Module, are connected to one port of kd-tree (DualPort BlockRAM).

and a cycle for multiplexor assignment, that is twenty cycles are needed in total. The memory (DualPort BlockRAM) can manage two input/output in a clock cycle. Thus, we connect a module which is consisting of twenty Query Modules and one Traversal Module to each port of the memory. Fig. 7 shows this structure. By using this structure, two photons can be obtained and computed per a clock cycle. This module is used as a fundamental structure of k-NN.

5 Evaluation

Here, evaluation results of the implementation are shown. Although the implemented module is designed for the photon mapping, it can be applied to the other applications by fitting the size or width of the tree. For using various applications, we tried to various family of FPGAs with different size. Three different size

Table 2. Resource usage of basic module in FPGA series

	slice Registers		slice LUTs		BlockRAM		Op. Freq.
	Used	Available	Used	Available	Used	Available	[MHz]
Virtex7	8719(1%)	866400	13013(3%)	433200	42(2%)	1470	292.654
Virtex6	8713(1%)	595200	11913(4%)	297600	42(3%)	1064	284.131
Spartan6	8823(4%)	184304	12386(13%)	92152	82(30%)	268	155.885

Table 3. Resource usage of 26 basic modules in Virtex7

	slice Registers		slice LUTs		BlockRAM		Op. Freq.
	Used	Available	Used	Available	Used	Available	[MHz]
Virtex7(×26)	217820(26%)	866400	375897(86%)	433200	1050(71%)	1470	260.865

FPGAs, Virtex7(XC7VX550T), Virtex6(xc6vsx475t), and Spartan6(xc6vslx150) are selected, and resource usage and maximum operating frequency are measured. FPGAs with a large size of memory are selected in each FPGA series. Table 2 shows the results. All modules are implemented in verilog-HDL, and simulated with NC-verilog 8.10. Designs are synthesized and place&routed by using Xilinx ISE 13.4.

In Table 2, the usage of slice LUTs and BlockRAM is high in all devices. In particular, since Spartan6 has only a small amount of BlockRAM, its usage becomes high. Since only tree modules can be implemented on Spartan6, and clock frequency is not so high (155MHz), high degree of acceleration is not expected. On the other hand, since Virtex7 and Virtex6 have a lot of resources, problems with a large size node can be implemented. Twenty six basic modules can be implemented on Virtex7. This system allows to obtain and calculate 52 photons per a clock cycle. Table 3 shows the synthesis results. The throughput, which is the number of photons that can be calculated per clock, is 13.564G[OPS].

We compared this result with the throughput of a software implementation of k-NN with kd-tree. The number of Emit-Photons is 1023, and the number of Estimate-Photons is 10, and the time required process to the 1M queries are evaluated. The experimental setup is shown in Table 1. The program is described in C language and complied with the option -O3. As a result, the processing time is 1.63 seconds, the number of processed photons is 78.568M, and the throughput is 48M[OPS]. It appears that the proposed module provides the throughput of approximately 282 times as the software execution at maximum.

6 Conclusion and Future Work

An acceleration method of k-NN with kd-tree in photon mapping on an FPGA is proposed. Prior to the design on an FPGA, the computation time is profiled and influence of parameters is analyzed As a result, in k-NN with kd-tree in photon mapping, it appears that the number of Estimate-Photons gives the largest impact to the processing time, and a large part of processing time is occupied by the discovery process of max distance.

In the proposed design, we maximized the effective throughput of the block RAM by connecting multiple Query Modules to both ports of RAM. Furthermore, an implementation of the discovery process of the max distance which is not depending on the number of Estimate-Photons is proposed. Through the implementation on Spartan6, Virtex6 and Virtex7, it appears that 26 fundamental modules can be implemented on Virtex7. As a result, the proposed module achieved the throughput of approximately 282 times as that of software execution at maximum.

In this research, as experimental implementation, we configure the node size to 1024. However, many nodes are needed in the actual photon mapping. Therefore, in future work, we should investigate the circuit performance of kd-tree with more large size. At that time, it will be considered that the memory resources will be wasted by the concentrating access to the part of the large kd-tree. Thus, we will consider that multiplexing nodes which access it are concentrated, or placed in split trees.

References

1. Jensen, H.W.: Global illumination using photon maps. In: Proc. of the Eurographics Workshop on Rendering Techniques, pp. 21–30. Springer (1996)
2. Larsen, B.D.: Real-time global illumination by simulating photon mapping (September 2004)
3. Purcell, T., Donner, C., Cammarano, M., Jensen, H.W., Hanrahan, P.: Photon mapping on programmable graphics hardware. In: Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware 2003, pp. 41–50 (2003)
4. Singh, S., Pan, S.H., Ercegovac, M.: Accelerating the photon mapping algorithm and its hardware implementation. In: 22nd IEEE Intl. Conf. on Application-Specific Systems, Architectures and Processors, pp. 149–157. IEEE (September 2011)
5. Huaiqing, H., Tianbao, W., Qing, X.: Photon Mapping Parallel Based On Shared Memory System. Imaging and Graphics (2009)
6. Jensen, H.W.: Realistic Image Synthesis Using Photon Mapping. A. K. Peters, Ltd., Natick (2009)
7. Foley, T., Sugerman, J.: KD-tree acceleration structures for a GPU raytracer. In: Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics Hardware, p. 15. ACM Press, New York (2005)
8. Havran, V.: Heuristic ray shooting algorithms (2000)
9. Saegusa, T.: An FPGA implementation of k-means clustering for color images based on Kd-tree. In: Intl. Conf. on Field Programmable Logic and Applications, FPL 2006, pp. 1–6 (2006)
10. Saegusa, T., Maruyama, T.: An FPGA implementation of real-time K-means clustering for color images. Journal of Real-Time Image Processing 2(4), 309–318 (2007)
11. Hughey, M.K., Berry, M.W.: Improved query matching using kd-trees: A latent semantic indexing enhancement. Information Retrieval 2, 287–302 (2000)
12. Wess, S., Althoff, K., Derwand, G.: Using k-d Trees to Improve the Retrieval Step in Case-Based Reasoning. In: Wess, S., Richter, M., Althoff, K.-D. (eds.) EWCBR 1993. LNCS, vol. 837, pp. 167–181. Springer, Heidelberg (1994), http://dx.doi.org/10.1007/3-540-58330-0_85

SEU Resilience of DES, AES and Twofish in SRAM-Based FPGA

Uli Kretzschmar, Armando Astarloa, and Jesús Lázaro

Department of Electronics and Telecommunications
University of the Basque Country UPV/EHU
Bilbao, Spain
{uli.kretzschmar,armando.astarloa,jesus.lazaro}@ehu.es

Abstract. Cryptographic algorithms play an important role in a broad range of applications. Block cipher algorithms represent a popular choice in many products and applications, where data needs to be handled in a secure way. The wide application of the Data Encryption Standard (DES) or its successor the Advanced Encryption Standard (AES) show evidence for the adequacy of these ciphers, which is based on their security combined with a high data throughput. There are many studies analysing and comparing different attributes of block cipher algorithms, like implementation efficiency or security against attacks. The main contribution of this work is the evaluation of the SEU resilience of different algorithms by applying a SEU injection flow on FPGA implementations of three popular block ciphers.

1 Introduction

Since the introduction of DES in 1977, data encryption was beginning to play an increasing role in many different electronic applications. Whenever critical data, like e.g. the readout of an electronic heat cost allocator or identification data on an RFID card, are to be transmitted on an insecure channel encryption needs to be considered. But also other applications like disc encryption or even protected firmware updates require symmetrical data encryption.

Reconfigurable devices such as of FPGAs represent a very suitable choice for the implementation of solutions in small- to medium-volume productions. It provides access to the benefits of modern nano scale production processes without creating high non-recurring engineering costs. In addition to that, FPGAs provide a short time-to-marked and the possibility of in the field reconfiguration. The reconfigurable nature of these devices makes it possible to react on protocol changes or to correct errors in the initial design after product shipping.

Together with these advantages, SRAM based FPGAs have the disadvantage of an increased SEU susceptibility. A SEU can be caused by high energy particles entering the silicon substrate. Different sources of these high energy particles are known [1]. One possible source of particles is cosmic radiation. But also packaging material impurities like uranium and thorium can emit alpha particles as they decay. A third source can be low-energy particles interacting with insulator materials on the substrate.

If a SEU occurs in the configuration memory of the FPGA, an unexpected change in the configurable logic or routing can be induced. This can change the behaviour of the application implemented on the device. When implementing cryptographic algorithms on FPGA the effects of SEUs need to be considered. One way of doing this is identifying the Failure In Time (FIT) rate or the directly correlated Mean Time Between Failure (MTBF), which is facilitated for Xilinx FPGA using the data from the Rosetta Experiment [2]. Within the Rosetta Experiment groups of the same FPGA are tested on the occurrence of SEU in order to provide the expected FIT rate per megabyte of configuration memory. For getting the devices FIT rate this value from the Rosetta Experiment is typically multiplied by the configuration memory size, or for a more exact evaluation the number of critical configuration bits. The number of critical configuration bits can be found by multiplying the device size with the percentage of critical bits of the given application, which was determined e.g. by an SEU injection scheme.

The work is structured as follows: Section 2 introduces the three block ciphers on which the SEU resilience evaluation is executed. In the following Section 3 the method of SEU injection is summarized. The different implementations of the block ciphers and their attributes are presented in Section 4 followed by the SEU resilience results in Section 5 and a final conclusion.

2 Tested Block Ciphers

2.1 DES – Data Encryption Standard

The Data Encryption Standard was developed in response to a call for proposals issued by the National Bureau of Standards (NBS), which is the former name of the National Institute of Standards and Technology (NIST). This call for proposals was based on the need of encryption for the US governments sensitive information. Winner of this call for proposals was an enhanced version of the IBM cipher *Lucifer* developed by Horst Feistel, after whom the whole subclass of *Feistel ciphers* are named today. In the year 1977 DES was published as FIPS PUB 46-2, with the latest version being FIPS PUB 46-3 [3].

DES consists of 16 rounds of permutations and substitutions. The blocksize and the keysize are 64 bits, whereas the effective keysize is reduced to 56 bits because in each byte of the key a bit is used as parity. Each of the 16 cipher rounds consists of the following steps: expansion, key mixing, substitution and permutation. The substitution step uses eight s-boxes to produce a non-linear function, which is the basis for the security of DES.

Because of its keysize of effectively only 56 bit, DES could be broken in 1997 by a brute force attack using multiple computers [4], later on also by special architectures like the FPGA-based *COPACOBANA* [5] which is able to break DES within days.

2.2 AES – Advanced Encryption Standard

Due to the short key length of DES together with the continuous advances in computation performance a need emerged to replace DES by a new algorithm providing a bigger key length and higher security. A call for proposals was issued in 1997 by the National Institute of Standards and Technology in order of finding the AES, which was the name selected for the successor of DES. Out of fifteen received proposals a subset of the Rijndael algorithm won the competition. This decision was based on the fact that it fulfilled best the given requirements of NIST, such as security, cost of implementation in terms of silicon area for hardware implementations and calculation time for software implementations.

AES has a single blocksize of 128 bit and a keysize of either 128, 192 or 256 bit, whereas Rijndael allows both the key- and the blocksize to be a multiple of 32bit which lays between 128 and 256 bit. The algorithm consists of 10-14 rounds depending on the keysize. A normal round consists of four steps, namely SubBytes, ShiftRows, MixColumns and AddRoundKey. All these actions operate on the so-called state, a 4 by 4 matrix of 8 bit values, which represents the block data. The SubBytes step uses a s-box as a lookup table for substituting the state bytes providing the non-linearity of AES. The complete algorithm is described in FIPS PUB 197 [6]

AES is used in a wide range of applications such as wireless LAN, voice over IP, disc encryption and file compression.

2.3 Twofish

The algorithm Twofish also participated in the Advanced Encryption Standard contest and it reached the second round of the best five proposals. Twofish has a blocksize of 128 bit and a keysize of 128, 192 or 256 bit. In contrast to Rijndael it uses the feistel structure, which is also used by DES. The algorithm uses four s-boxes and executes 16 rounds of encryption. The complete algorithm is described in [7].

Twofish is used in a variety of disc encryption programs as well as in the GNU Privacy Guard.

3 SEU Test Flow

The desired result of a SEU resilience study based on SEU injection is an estimation on the percentage of critical bits, where a critical bit is one bit of the FPGAs configuration memory, which when flipped alters the behavior of the design implemented on the FPGA. This percentage of critical bits can be converted to the λ parameter giving the number of expected errors in one billion hours of operation or to the Mean Time Between Failure (MTBF) using the following equations:

$$\lambda = deviceSize * ratio_{critical} * RTSER \tag{1}$$

$$MTBF = \frac{10^9}{\lambda * 365.24 * 24} \tag{2}$$

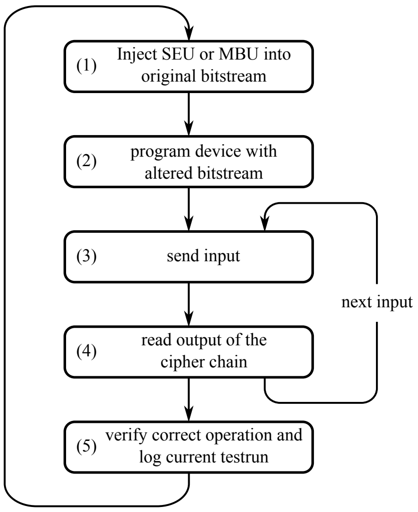

(1) Inject SEU or MBU into original bitstream
(2) program device with altered bitstream
(3) send input
(4) read output of the cipher chain
(5) verify correct operation and log current testrun
next input

Fig. 1. SEU injection and test-flow

The factor RTSER stands for Real Time Soft Error Rate as described in [2].

Due to the big device sizes and the size of the bitstreams it is not feasible to test all bits of the configuration memory. Instead a subset of randomly chosen bits is tested and the percentage of critical bits within this subset is assumed to be equal to the percentage of critical bits on the whole device.

For SEU resilience evaluation the test flow from [8] is used together with the adoptions made in [9]. Figure 1 summarizes this flow.

The flow consists of five sub-steps and the execution of the five steps is called testrun. A testrun returns the result for one bit if it is critical or not.

3.1 Error Injection and Device Programming

The first two steps of the flow are used for determining the bit position, which is to be tested and for setting up the FPGA for the behavioural test.

For injection of an SEU or a MBU into FPGA configuration memory for emulation of SEU resilience it is necessary to analyse the bitstream of the application of interest to a certain degree. Within this package oriented file consisting of headers and payload only bits of the payload designated to the FDRI register [10] may be altered, because this payload represents the data written to the configuration SRAM. It also needs to be ensured that CRC checking is disabled when generating the initial bitstream to avoid CRC errors when programming the device with a modified bitstream.

After injection of a single- or multiple errors into the initial design(-bitstream), the resulting modified configuration file gets programmed using the *IMPACT* [11] programming tool. This tool features a command line interface allowing an efficient automation of this step in the flow.

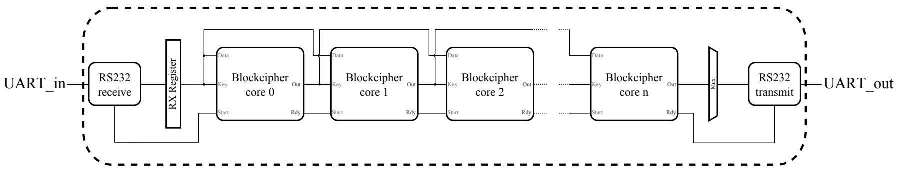


Fig. 2. Chaining of blockcipher modules

3.2 Behaviour Validation

Succeeding the programming of the SEU or MBU affected bitstream the behaviour of the implemented design needs to be validated in order to categorize the injected error bit positions as critical or uncritical in step (3) and (4) of Figure 1.

Within this work a UART connection for communication with the design, namely the sending of a plaintext for the cipher modules and the receiving of the encrypted results is used. Although UART does not allow for very high communication speeds it is a simple communication interface requiring only a low amount of logic for its implementation. These small resource requirements are highly beneficial for SEU resilience testing, because like this it is unlikely that injected errors will affect the UART part of the design.

For each plaintext sent to the block cipher modules the correct output can either be determined by using a software implementation of the same algorithm, or by executing a so-called golden run, where the device gets programmed with an unmodified bitstream. The data gained for the golden run serves as a reference for determining correct device behaviour.

The validation routine consists of eight complete calculations of the cipher chain. Eight different input vectors are supplied via UART and the eight corresponding calculation results get checked against the golden run.

Sending a single input vector with subsequent testing of the chains results, would not be a sufficient test, because like this the coverage of the s-boxes would not be sufficient. For example an 8 by 8 s-box contains 256 different values, but it gets accessed only 16 times in one encryption for the cases of the Twofish implementation. Using multiple input vectors for testing as done in this work tests the design to a higher degree. This higher test coverage causes a better detection of critical bits.

3.3 Flow Control

Gaining significant SEU resilience data requires testing of a large number of bit positions, which feeds the need of a completely automatic test flow control. In this work a custom set of PHP scripts was developed for the flow control, the error injection, the control of the device programming and the result logging. The UART communication was implemented in C++ as well as the checking against the golden run.

Table 1. Implementation results on Xilinx XC5VFX70T

	DES	AES	Twofish
Blocksize/Keysize [bit]	64/64	128/128	128/128
Chain size n [nr. of cipher modules]	91	22	30
Virtex-5 Slices	11008 (98%)	10976 (98%)	11187 (99%)
Virtex-5 LUTs	38546 (86%)	40658 (90%)	40297 (89%)
Virtex-5 Registers	18172 (40%)	17490 (39%)	21275 (47%)
cycles per encryption	16	16	73
f_{max} for single* [MHz]	256	318	91
Data throughput per module* [Mbit/s]	1024	2544	160
Accumulated data throughput** [Gbit/s]	93.2	56.0	4.8

4 Implementation Results

This work uses the Virtex-5 FPGA XC5VFX70T mounted on the Xilinx Evaluation Platform ML507. Since each of the block cipher modules evaluated by this work occupies just a fraction of the actual capacity available on the FPGA for a single instance, multiple instances are chained in order to fill the FPGA to a high degree. A high device utilization reduces the number of configuration bits controlling unused resources, which are likely to be uncritical. The concept of this chaining is summarized in Figure 2.

Providing data to the chain and receiving the chains output is realized using RS232. Initially a 64 bit (for DES) or 128 bit register (for AES and Twofish) receives 8 or 16 bytes of data.

The data of this register is connected to the data- and key-inputs of the first encryption unit in the chain. For all following modules of the chain the encryption key is fed by the output of the previous module and the data input is fed with the output-data of the module two blocks ahead in the chain. Using different inputs for key- and encryption-data inputs instead of using the output of the previous module for the two inputs of the following module represents a more realistic case and results in better data of the SEU testing. This improvement is based on the prevention of unwanted design optimization by synthesis, which otherwise could be possible due to the equivalent values on key- and data-inputs. The ready signal of each module is registered and triggers the encryption of the subsequent module in the following cycle.

After the last block cipher module has finished its calculations the sending of the 64 or 128 bit of encryption result is executed. The 8 or 16 bytes of result are send by a RS232 module to the connected host PC.

In the following a summary of the block cipher implementations used in this work is given.

4.1 DES

As an example for the DES cipher the implementation given by the company Tetraedre [12] was selected. The style of this implementation requires the 64 bit data output of each module to be registered before feeding it to the next module in the chain. This implementation needs 16 cycles for one encryption and achieves a speed higher than 250 MHz on a Virtex-5 FPGA. The small size of the implementation allowed the construction of a chain of 91 DES encryption modules on the test board.

4.2 AES

The AES implementation of Hemanth Satyanarayana, available on OpenCores [13], was implemented. It calculates one AES round in one cycle, but due to a 64 bit interface for key- and data-input it requires 16 cycles performing a complete 10 round AES encryption. The achievable operating frequency is 318 MHz and 22 encryption modules could be fitted onto the XC5VFX70T device.

4.3 Twofish

The Twofish implementation of this work was also taken from OpenCores [7]. For having a similar interface as the AES implementation, a slight statemachine changes were implemented in order to get a single start signal per module instead of a *load key* and a *load data* signal as originally implemented. An operation frequency of 91 MHz and a chain size of 28 modules where achieved on the ML507 board.

Table 1 compares the gained results for the implementations of the different block ciphers. The values gained for f_{max} are marked with an asterisk to emphasize that this value is the estimated value by the Xilinx XST tool in standard settings (optimization on speed, normal effort) implementing not the whole chain, but a single module only. The row *Accumulated cipher data throughput*, marked with two asterisks, represents a hypothetical value assuming that all cipher modules are working in parallel, but the chain as used in this work operates sequentially. Nevertheless this theoretical value represents a great estimation on the throughput-area trade-off.

5 SEU Injection Results

For each of the three block cipher module chains the SEU test flow has been executed flipping a single bit, two- and three bits of the configuration bitstream, which emulates the occurrence of SEUs, 2-bit MBUs and 3-bit MBUs. For each of the three error types and each of the three block cipher chains 20000 testruns were executed giving an overall of 180000 error injections. Table 2 summarizes the test results.

The AES implementation exhibits the lowest number of critical bits of all three block ciphers against all three tested types of injected errors. Only slightly

Table 2. Injection Results

	DES	AES	Twofish
Errors in 20000 1-bit SEU injections	2796 (13.98%)	2675 (13.38%)	3398 (16.99%)
Errors in 20000 2-bit MBU injections	5233 (26.17%)	4977 (24.89%)	6471 (32.36%)
Errors in 20000 3-bit MBU injections	7393 (36.97%)	7021 (35.11%)	8832 (44.16%)

elevated values compared to these of the AES were found for the DES implementation, whereas the Twofish chain showed a significant higher susceptibility to SEU, 2-bit MBU and 3-bit MBU.

The obtained critical bit values of more than 13% are quite elevated. A Xilinx study [14] states, that only a very low percentage of all applications will exhibit a critical bit rate higher than 10%. There are two main reasons for the high critical bit ratio of the cipher chains: firstly encryption algorithms have the nature of permuting data to a high degree which enables a high degree of logic to be used at the same time. This behaviour differs from other applications, like e.g. processor cores where whole parts of logic (e.g. specific instructions) are implemented, but in some cases never used. Secondly this work uses the given FPGA to a very high degree. A big part of the device resources are used leaving a small amount of unused configuration bits.

Using the Equation (2) allows the finding Mean Time Between Failure for the corresponding percentage of critical configuration bits. Figure 3 shows the resulting values for the different ciphers in the green bars. The MTBF for the Twofish chain is 150 years, compared to 182 years for DES and 190 years for the AES implementation.

Even though a time between two errors of hundreds of years seems to be a very uncritical value (especially taking into consideration, that not all systems run permanently), there are applications, where this value can already lead to issues. If for example a hypothetical system consists of 1000 boards implementing the same application and each board has a MTBF of 150 years, the time between faults in any of the boards is only 0.15 years or 55 days.

Further conclusions can be drawn when comparing the results of the fault injection campaign to data gained by the bitstream generation tool. Within this tool there is an option for static evaluation, which generates a file (.ebd) containing the information on critical configuration bits. Details on this static evaluation can be found in [8]. Figure 3 also contains this estimation based MTBF in the red bars. The estimation based robustness data is more pessimistic than the actual value obtained by fault injection. It can also be observed, that the robustness estimation found the AES and TwoFish chains to be equally robust against SEU. This point underlines the advantage of robustness evaluation by fault injection, because like this a significant difference between these two ciphers could be detected.

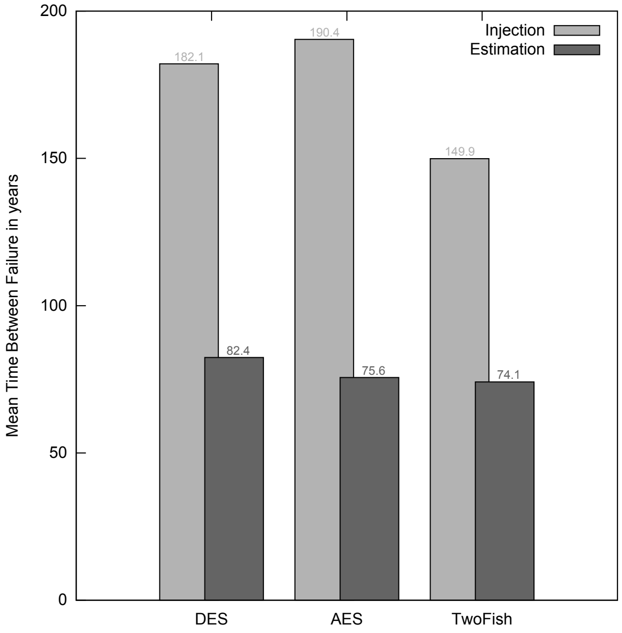


Fig. 3. MTBF for the test application and SEU type errors

The reason for the significantly inferior SEU resilience of Twofish is likely to be caused by an efficient use of the FPGAs resources by synthesis. For example focusing on the FPGA slice resources, when a high percentage of the available resources is used, the corresponding configuration bits are likely to be critical. An example of the contrary for e.g. Look Up Tables (LUT) would be, if synthesis was only able to use the four or five inputs of the 6 input LUTs of Virtex-5.

No special influence of the fact that a cipher belonging to the group of Feistel ciphers could be observed. AES is no Feistel cipher, but has a similar SEU resilience as the Feistel cipher DES. In contrary the two Feistel ciphers DES and Twofish have significantly differing SEU resiliences.

An interesting observation is, that the results for the AES implementation are worse compared the results in [13], which uses the same AES implementation, the same FPGA, the same idea of chaining the IP cores and the same number of IP cores. The reason for this difference lies in the structure of the encryption chain. In [13] the AES modules are chained in a way that both key- and data-input of the *n-th* AES module are fed by the ciphertext-output of the *(n-1)-th* module. The higher complexity of interconnection in the chain used by this work results in a more complex configuration of the interconnect resulting in a worse SEU resilience.

6 Conclusion

This work uses three different block ciphers, each represented by an open source VHDL design. For each block cipher implementation a chain of multiple cores was build using a high amount of FPGA resources. On the resulting cipher chains an error injection testflow for one-, two- and three-bit errors was executed determining the SEU resilience of the corresponding block cipher. All block ciphers were found to be highly susceptible to SEU or MBU giving an indication, that the consequences of SEU are not negligible especially for applications using an high amount of cipher operations.

Acknowledgements. This work has been supported by the Ministerio de Ciencia e Innovación of Spain within the project TEC2011-28250-C02-01 and it has been carried out in the Research and Education Unit UFI11/16 of the UPV/EHU and supported by the Department of Education, Universities and Research of the Basque Government within the fund for research groups of the Basque university system IT394-10.

References

1. Xilinx Corp., Considerations Surrounding Single Event Effects in FPGAs, and Processors, Xilinx Doc. (March 2012), `http://www.xilinx.com`
2. Xilinx Corp., Device reliability report, fourth quarter 2010, Xilinx Doc. (February 2011), `http://www.xilinx.com`
3. NIST, Data Encryption Standard (DES) (FIPS PUB 46-3), National Institute of Standards and Technology (October 1999)
4. Curtin, M., Dolske, J.: A brute force search of DES keyspace (May 1998), `http://www.interhack.net/pubs/des-key-crack/`
5. Kumar, S., Paar, C., Pelzl, J., Pfeiffer, G., Schimmler, M.: Copacobana a cost-optimized special-purpose hardware for code-breaking. In: Proc. of the 14th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM) (April 2006)
6. NIST, Advanced Encryption Standard (AES) (FIPS PUB 197), National Institute of Standards and Technology (November 2001)
7. Opencores.org, Twofish - Project: Twofish Core, OPENCORES (February 2002), `http://opencores.org`
8. Kretzschmar, U., Astarloa, A., Lazaro, J., Jimenez, J., Zuloaga, A.: An Automatic Experimental Set-up for Robustness Analysis of Designs Implemented on SRAM FPGAs. In: International Symposium on System on Chip (SoC) (November 2011)
9. Kretzschmar, U., Astarloa, A., Lazaro, J., Bidarte, U., Jimenez, J.: Robustness Analysis of Different AES Implementations on SRAM Based FPGAs. In: Intl. Conf. on Reconfigurable Computing and FPGAs (ReConFig) (December 2011)
10. Xilinx Corp., Xilinx-5 FPGA Configuration User Guide, Xilinx Documentation (August 2010), `http://www.xilinx.com`
11. Xilinx Corp., iMPACT User Guide, Xilinx Docu. (January 2002), `http://www.xilinx.com`
12. Tetraedre Sarl, Auvernier, TCDG - DES cryptographic module (September 2010), `http://www.tetraedre.com/advanced/index.php`
13. Satyanarayana, H.: AES128 - Project: aes_crypto_core, OPENCORES (December 2004), `http://opencores.org`
14. Chapman, K.: Virtex-5 SEU Critical Bit Information Extending the capability of the Virtex-5 SEU Controller, Xilinx Documentation SEU lounge (February 2010), `http://www.xilinx.com`

A Fast Poisson Solver for Hybrid Reconfigurable System

Vitor Gomes[1], Haroldo Campos Velho[1], and Andrea Charão[2]

[1] Brazilian Institute for Space Research
Laboratory of Computing and Applied Mathematics 12227-010,
São José dos Campos/SP - BRA
{vitor.gomes,haroldo}@lac.inpe.br

[2] Federal University of Santa Maria, Computer Systems Laboratory
97105-900, Santa Maria/RS - BRA
andrea@inf.ufsm.br

Abstract. This paper presents the design and implementation of a fast Poisson solver on a reconfigurable hybrid system. Our hybrid solver integrates a FPGA-based FFT coprocessor to collaborate in the solution of a numerical meteorological model involving one-dimensional shallow water equations. The Poisson equation is solved using a singular value decomposition associated with the Moore-Penrose inverse. The hybrid fast Poisson solver is evaluated under different amount of data entry and shows performance gains compared to the reference application.

Keywords: Fast Poisson Solver, Hybrid Reconfigurable Systems, FPGA.

1 Introduction

The Poisson equation is an elliptic partial differential equation that may arise in evolution models, as for example weather forecast models. The ability to efficiently solve such equation allows complex systems to be processed in time for accurate forecasts. When solving the Poisson equation, an efficient technique is using a Fast Fourier Transform (FFT). Besides the development of more efficient algorithms, one can exploit different architectures for high performance computing. One of the promising architectures are hybrid systems, where heterogeneous processing units cooperate to speed up computationally-intensive applications.

In this paper, we describe the design and implementation of a Fast Poisson Solver in a hybrid reconfigurable system. Our target architecture incorporates FPGAs as specialized coprocessors to accelerate applications. Our solver is applied to a numerical meteorological model involving one-dimensional shallow water equations.

The rest of this paper is organized as follows. Section 2 provides some background on Fast Poisson Solver and Fast Fourier Transform. It also presents the target computing system for this work and discusses some related work focused on solving PDEs on hybrid reconfigurable systems. Section 3 presents our hybrid design and explain our FFT coprocessor architecture. It also presents some implementation details. Section 4 describes our experimental analysis and discusses the results, while Section 5 presents some final considerations.

2 Background and Related Work

2.1 Fast Poisson Solver

Algorithms that reduce the computational cost of $O(N^2)$ the solution of the Poisson equation are called Fast Poisson Solver. One way to reduce the complexity processing of this algorithm is the use of FFT, which provides a solution with complexity $O(NlogN)$. In this work, we use a singular value decomposition (SVD) associated with the Moore-Penrose inverse.

Moore-Penrose Inverse. Let A be an $m \times n$. Then a $X_{m\times n}$ matrix that satisfies any or all of the following properties (Penrose conditions) is called a generalized inverse

$$(1) AXA = A; \qquad (2) XAX = X; \tag{1}$$

$$(3) (AX)^* = AX; \qquad (4) (XA)^* = XA. \tag{2}$$

where $(*)$ is the conjugate transpose. A matrix satisfying all of the properties above is called a *Moore-Penrose Inverse* (or M-P inverse) of A. Every matrix A has an unique M-P inverse [1], which will be denoted by A^+.

For computational purposes, the above definition is not practical. However, there are many algorithms to find the M-P inverse [2]. In this work, we use a matrix decomposition that is of great utility for the manipulation of rectangular (or square) matrices. This is the singular value decomposition (SVD) of a scalar matrix, as described below.

Let A be an $m \times n$ matrix with complex elements and of rank r. Then there exists unitary matrices U, V of orders m and n, respectively, such that

$$A = UDV^*; \qquad D = \begin{bmatrix} D_1 & 0 \\ 0 & 0 \end{bmatrix}; \tag{3}$$

where $(*)$ is the conjugate transpose, D is $m \times n$ and $D_1 = diag[d_1, d_2, \ldots, d_r]$ is a nonsingular diagonal matrix of order r.

By using the SVD, it can be found a convenient formula for the M-P inverse (A^+). That is, if $A = U^*DV^*$, where U and V are unitary matrices, then

$$A^+ = VD^+U; \qquad D^+ = \begin{bmatrix} D_1^{-1} & 0 \\ 0 & 0 \end{bmatrix}. \tag{4}$$

For the matrix equation $Ax = b$, the solution: $x = A^+b$ is understood as the least squares solution to the system [2].

Circulant Matrices. When A is a circulant matrix, the M-P inverse can be calculated in an easier way. A circulant matrix C of order n , or simply circulant, is a matrix of the form

$$C = circ(c_1, c_2, \dots, c_n) = \begin{bmatrix} c_1 & c_2 & \dots & c_n \\ c_n & c_1 & \dots & c_{n-1} \\ \vdots & \vdots & & \vdots \\ c_2 & c_3 & \dots & c_1 \end{bmatrix}; \tag{5}$$

and a circulant can be diagonalized using Fourier matrices F_n^* e F_n [3]

$$C = F_n^* \Lambda F_n; \tag{6}$$

where Λ is the diagonal matrix

$$\Lambda = \sum_{k=0}^{n-1} c_{k+1} (\Omega_n)^n;$$
$$\Omega = diag[1, \omega, \omega^2, \dots, \omega^{n-1}]; \qquad \omega = e^{\frac{2\pi i}{n}}. \tag{7}$$

Thus, if C is circulant its M-P inverse is the circulant [3, 2]

$$C^+ = F_n^* \Lambda^+ F_n; \tag{8}$$

where

$$\Lambda^+ = diag[\lambda_1^+, \lambda_2^+, \dots, \lambda_n^+]; \qquad \lambda_k^+ = \begin{cases} \frac{1}{\lambda_k} & \text{if } \lambda_k \neq 0 \\ 0 & \text{if } \lambda_k = 0 \end{cases}. \tag{9}$$

2.2 Fast Fourier Transform

Fourier transforms are linear transformations used in several scientific and engineering applications. In their discrete formulation, these transforms are usually the core of computational applications, from image processing to atmospheric simulation. The Discrete Fourier Transform (DFT) of a sequence of N numbers can be computed as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}; \qquad k = 0, 1, ..., N-1; \tag{10}$$

where $W_N = e^{-2\pi\sqrt{-1}/N}$ is a trigonometric coefficient known as the twiddle factor. The Fast Fourier Transform algorithm (FFT) [4] computes DFT reducing the complexity from $O(N^2)$ to $O(NlogN)$.

There are many ways to structure the FFT algorithm. One variant is the radix-2 algorithm: it takes a divide-and-conquer approach, which operates on an N-point data set, where N is a power of 2. Its basic operation is known as "butterfly" and consists of two complex additions and a complex multiplication. The radix-2 algorithm yields the smallest butterfly unit, which allows for greater flexibility in studying the design space [5, 6].

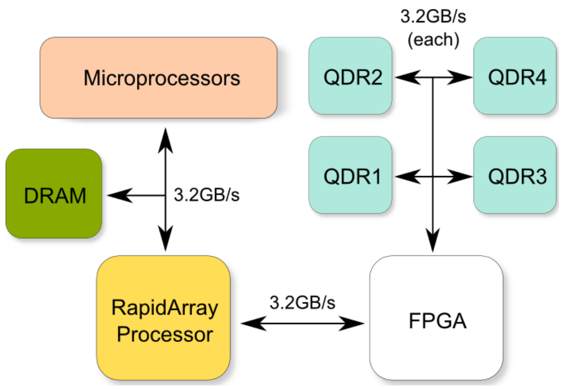


Fig. 1. Cray XD1 blade architecture

2.3 Cray XD1

Our target architecture is a Cray XD1, which is one of the first commercial hybrid systems combining CPUs and FPGAs. Our Cray XD1 system is made up of six interconnected nodes (blades), each one containing two 2.4GHz AMD Opteron general-purpose processors and one Xilinx Virtex II Pro FPGA. Figure 1 shows the architecture of an XD1 node (blade). The reconfigurable device has direct access to four banks of QDR II SRAM. Through a RapidArray processor, the FPGA can also access the DRAM of the processors [7]. While developing hybrid programs for the XD1, two key issues are moving data between the FPGA and the processors and efficiently using the memory hierarchy available to FPGA designs.

2.4 Solving PDEs on Hybrid Reconfigurable Systems

As a computationally intensive application, solving PDEs is suitable for hardware acceleration and FPGA-based implementation. However, few studies have evaluated the use of FPGA in collaboration with the CPU for this type of application. Studies on solving PDEs in these architectures are mainly focused on GPUs and the Cell architecture [8]. Hybrid reconfigurable systems are usually associated with image processing, molecular dynamics and mixed precision.

A recent work that deals with the solution of PDEs using reconfigurable computing is [9]. This work discusses the implementation of Finite Difference and Finite Element methods in software and hardware. These implementations vary in the numerical representation used. Some use fixed point, while others use floating-point, depending on the complexity of the solution and the available hardware resources. The emphasis of this work is on solutions that can perform well on low cost platforms. These platforms are characterized by low bandwidth for communications between FPGA boards and the host. These solutions are advantageous for embedded solutions, but provide limitations to their use in

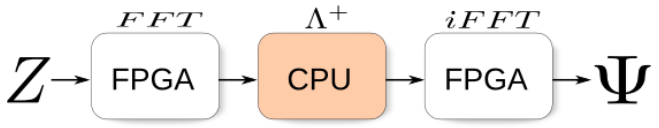


Fig. 2. Task mapping and data flow between FPGA and CPU

high performance hybrid reconfigurable systems. This is specially true for applications that constantly use the FPGA to perform critical kernels of intensive processing, requiring continuous transfer of data between the devices.

To our knowledge, there is not yet a thorough investigation of hybrid designs that leverages the power of using both the FPGA and the CPU to compute Fast Poisson Solver. This approach is rather recent, due to the new reconfigurable computing systems brought to the market over the past few years. Some related work on hybrid designs address different computational kernels and are limited to linear algebra operations and optimization problems [10, 11]

3 Hybrid Fast Poisson Solver

The design of the hybrid FFT Poisson solver aims to harness the computing power of both the CPU and the FPGA, using the solution presented in section 2.1. After discretization of the elliptic equation (the case treated here is a Poisson equation with periodic boundary conditions), the steps of the solution are: (a) factorization of coefficient matrix, see Eq. 6; (b) calculate the FFT of the input Z; (c) multiply the result of (b) by the diagonal matrix Λ^{+}; and (d) calculate the inverse FFT of the result of (c), reaching the solution of the Poisson equation. Step (c) has computational complexity $O(N)$, because only the elements on the main diagonal of the matrix are nonzero.

In this solution of the Poisson equation, the Fourier transform is the critical step and therefore was chosen to be processed in FPGA. The task mapping and data flow between FPGA and CPU are illustrated in Figure 2. Because the FFT is the most complex task in this work, subsection below shows the design details of the FFT coprocessor.

3.1 FFT Coprocessor

Our architecture aims to maximize the use of reconfigurable hardware resources, dealing with the costs of data transfer between CPU and FPGA and the various memory levels. To do so, it aggressively explore multiple levels of parallelism, latency hiding and computing pipeline opportunities in the hybrid system. To deal with each of these issues, our architecture is structured into functional units as shown in the block diagram in Figure 3.

The Communication Unit is an interface between the RapidArray processor, that links the FPGA to the processors. This entity responsible for transferring

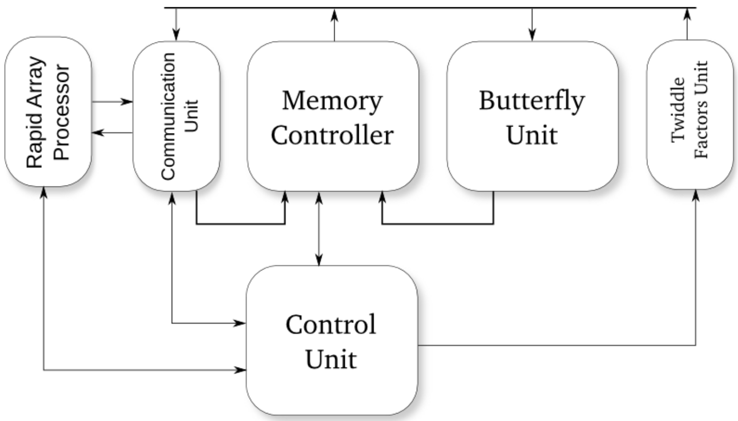


Fig. 3. Block diagram of the FFT coprocessor

the data set from blade memory to FPGA QDR banks. To perform this, the Communication Unit has two parallel processes that communicate directly with the RapidArray processor. The first process is responsible for requesting the data set. The requests are sequentially made at each clock cycle. The second process is responsible for handling responses of the RapidArray processor that can come out of order. The data organization is done using the internal memory of the FPGA before being sent to the module responsible for memory management.

The Butterfly Unit is the computational core of our architecture. It is responsible for processing the butterfly operations. Because the butterfly is the only FFT computational operation, the Butterfly Unit can perform the complex multiplication and complex additions through a pipeline without the need for advanced controls. Figure 4 shows the diagram of Butterfly Unit. In this Figure, a and b represents the inputs, tf is the Twiddle factor and vd is a signal indicating a valid input data.

Arithmetic operations which compose the complex operations are performed in parallel to reduce the total number of cycles required. Butterfly Unit comprises six floating-point adders and four multipliers. These modules allow the construction of a butterfly core that requires 33 cycles to perform this computation.

With this architecture, the Butterfly Unit can deliver a result every clock cycle after all stages of the pipeline are filled. However, this requires the data to be available at this frequency. This is a challenge, since each operation requires three 64 bits operands (two complex points and a complex twiddle factor) and generates two new complex points that should be stored on this same rate.

To address these issues we designed two units – Twiddle Factor Unit and Memory Controller – which also appear in the diagram of Figure 3.

The Twiddle Factor Unit is responsible for providing the trigonometric constants used in the butterfly computing. This unit produces these constants at runtime using $N - 1$ basic constants stored directly in hardware. The banks

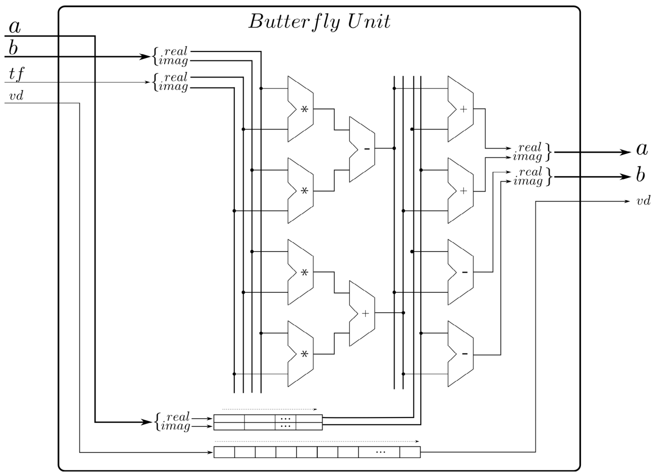


Fig. 4. Block diagram of the Butterfly Unit

of QDR memory are not used in this unit because they are necessary for the Memory Controller presented following.

To ensure the delivery of a new twiddle factor each clock cycle, a buffer is used at the output of this unit and the generation process is started during data transfer, before processing.

The other unit responsible for providing data every clock cycle for Butterfly Unity is the Memory Controller. This module provides access to four banks of QDR memory connected to the FPGA. These memory banks allow reads and writes completely independent. Thus there are several ways to organize reading and writing in the QDR, which required the greatest effort in this project.

The biggest challenge is to store the data distributed in four banks of QDR so they can be read and written two points per cycle. Each memory bank stores 64 bits per position which allows us to store a complex number (two 32 bits floating-point elements). For reading and writing a pair of data points per cycle we need, at least, two QDR banks. One technique that allows parallel reading is data replication. Thus it is possible to read two complex numbers per clock cycle. Nevertheless, the data recording is done sequentially, requiring two cycles, which prevents its use in our case.

Another alternative that has been evaluated is the sequential distribution of the blocks as used in the RAID-0 scheme. However, the sequence memory access of FFT requires, in some cases, to read data from the same memory bank, thus preventing the reading of parallel data.

As an alternative that avoids replication of data and distribute the data among four banks of QDR, we developed an addressing scheme for FFT, where reading and writing is always done in pairs. Subsection 3.1 describe the functioning of this system in more detail.

Table 1. Access sequence 16-point FFT

Step 1		Step 2		Step 3		Step 4	
0	8	0	4	0	2	0	1
4	12	2	6	1	3	2	3
2	10	1	5	8	10	4	5
6	14	3	7	9	11	6	7
1	9	8	12	4	6	8	9
5	13	10	14	5	7	10	11
3	11	9	13	12	14	12	13
7	15	11	15	13	15	14	15

Generation of trigonometric constants and reading of memory data must be synchronized for correct computation of the butterfly. The Control Unit is responsible for this task, to control the stages of computation, enabling the sending and receiving data by the Communication Unit. In addition, this unit manages state registers that store hardware metrics and provide the state of the hardware to software application.

Memory Controller Details. Addressing the memory for the FFT computation with high performance is a challenge. As mentioned above, the need for reading and writing of a complex data pair in each cycle requires an assessment of the pattern access memory used for computing the FFT. Table 1 shows the sequence of addresses accessed in each step of a 16-point FFT.

In the first step, the access sequence is {0; 8}, {4; 12}, {2; 10}, {6; 14} to the pair {7; 15}. In the second step, access is {0; 4}, {2; 6}, and so on. Observing these sequences, we found that there is a pattern between two pairs of sequential accesses and next accesses step. This pattern is exemplified by the different shades of gray in the Table 1. The sequence that this form of access changes between steps keeping a pattern.

Evaluating these patterns of memory access, we designed an addressing scheme that keeps data in memory in order to be accessed, facilitating the process of reading and writing. The addressing scheme used for data distribution between the blocks during storing is shown in Figure 5.

In this Figure, the blocks are grouped in two pairs {QDR1; QDR3} and {QDR2; QDR4}. Both pairs share $a_w(k-1,1)$ bits of address signal (k is the address size) and the signal $a_w(0)$ is used to enable storing. The difference between the block pairs is that the enable signal en_w is inverted. The input data to be written (d_w) is crossed between the pairs. In this case, QDR1 is connected with QDR2 and QDR3 is connected with QDR4.

The addressing scheme for reading is shown in Figure 6. We can see that this scheme is similar to the storing scheme. Pairs {QDR1, QDR3} and {QDR2, QDR4} share an address signal generated by the concatenation of $a_r(k-1)$ and $a_r(k-3,1)$. The $a_r(k-2)$ bit is connected to the multiplexer selection signal.

For both schemes, the address counters a_w and a_r are incremented sequentially every clock cycle, addressing the data in the order required for correct FFT

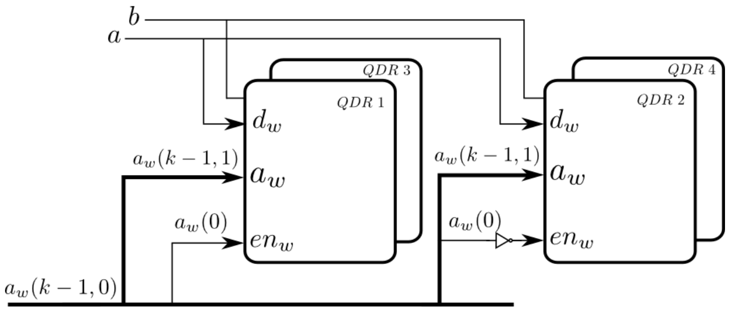


Fig. 5. Addressing scheme for storing

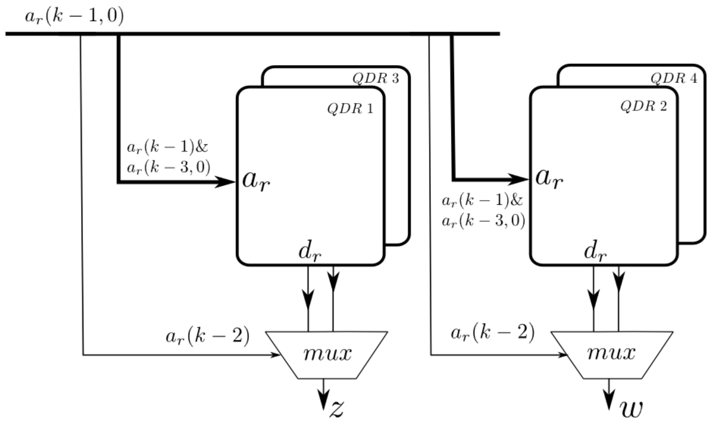


Fig. 6. Addressing scheme for reading

processing. Although there is sharing of address and data buses between the blocks, there is no data replication.

3.2 Implementation

The units of the design used in the FFT coprocessor described in section 3.1 were implemented using VHDL. The hardware description uses 64 bits to represent each data item, comprised of two 32 bits floating-point elements representing the real and imaginary parts of a complex number. For the floating-point operations, we used the Xilinx Floating-Point Core providing a 32-bit IEEE754 compliant implementation of adders and multiplies.

Our implementation of FFT in FPGA accepts a maximum of 2^{14} points. This limitation is due to the amount of internal memory available to the FPGA on CRAY XD1 blade (Virtex2P XC2VP50-7) to store twiddle factors.

To synthesize, place and route the hardware description we used Xilinx ISE 10.1. Our hybrid design runs at 190MHz on the FPGA.

4 Experimental Analysis: DYNAMO Model

DYNAMO [12] is a one-dimensional model that simulates various meteorological phenomena in atmospheric dynamics. It is deduced from shallow-water equations, involving the evolution of the primitive variables (horizontal components, wind and geopotential) or the evolution of the variables for the prognosis (vorticity, divergence and geopotential).

The most intensive computation in DYNAMO is calculating the current function Ψ and the velocity potential χ, respectively related to the vorticity ζ and the divergence δ. These variables are obtained through the solution of two Poisson equations

$$\nabla^2\Psi = \zeta; \qquad\qquad \nabla^2\chi = \delta; \tag{11}$$

employed in the calculation of the zonal and meridional velocities u and v [12]

$$u = \frac{\partial\chi}{\partial x}; \qquad\qquad v = \frac{\partial\Psi}{\partial x}. \tag{12}$$

By applying the classical form of the centered finite differences in (11), with periodic boundary conditions, the equation becomes

$$A_\Delta\Psi = Z \tag{13}$$

where $\Psi = [\psi_1, \psi_2, \ldots, \psi_{N_x}]^T$, $Z = [\zeta_1, \zeta_2, \ldots, \zeta_{N_x}]^T$ and

$$A_\Delta = \begin{bmatrix} -2 & 1 & & & 1 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 1 & & & 1 & -2 \end{bmatrix}.$$

The original DYNAMO model is written in Fortran. Our work focused on a subroutine named POIS1D, which solves the Poisson equation. To couple our hybrid fast Poisson solver in this model, we developed two versions of this subroutine: a reference version implemented in C, which runs only on CPU, and a hybrid version which performs simultaneous computations on CPU and FPGA. We coupled our C code with the original Fortran application using the Intel Fortran Compiler.

We compared the numerical results of DYNAMO using both versions of the POIS1D subroutine. As expected, they produced the same results, since our FPGA-based architecture performs floating-point operations using the same IEEE-754 standard as used by the CPU. After these tests to make sure the solution integrated with the DYNAMO provided correct values, the function POIS1D was isolated from the DYNAMO model for a series of performance tests. We performed tests for different sizes of input vectors: 64, 128, 256, 512, 1024, 2048, 4096, 8192, and 16284 points. For each input vector, we performed 10 executions

Table 2. Execution times for software and hybrid POIS1D

Points	Time (μs)		Std. Dev.		Speedup
	Software	Hybrid	Software	Hybrid	
64	58.00	23.70	1.52	0.64	2.45
128	104.30	32.60	1.60	1.50	3.20
256	183.80	54.90	1.89	0.70	3.34
512	365.80	105.10	1.92	2.77	3.48
1024	728.00	201.30	2.26	4.08	3.62
2048	1482.00	384.80	1.50	3.82	3.85
4096	3059.20	783.60	2.45	6.90	3.90
8192	6391.10	1595.70	3.47	13.83	4.01
16384	15040.10	3241.50	4.18	22.92	4.64

and calculated the average execution time. In Table 2 we present the results of our tests. The first column lists the size of the input vector, while the second and third columns shows the execution times for our reference POIS1D implementation in software and our hybrid version which runs in CPU and FPGA. The fourth and fifth columns present the standard deviation from the 10 executions we performed for each case. The last column present the speedup obtained with our hybrid solver. The speedup is calculated as the ratio between the reference implementation and the hybrid one.

In these experimental results, we observe that hybrid version is faster for all data sizes. The smallest gain was obtained for the execution with 64 points, while the highest gain was obtained for 16384-points experiment. In the most cases the standard deviation was lower for the reference application. This behavior may be related to data transfer between CPU and FPGA, which depends on several factors of the system (OS, DMA, etc.).

5 Conclusions and Future Work

In this paper, we have proposed an hybrid architecture to perform a fast Poisson solver in a hybrid reconfigurable computing system. In our design, both the CPU and the FPGA cooperate to compute the fast Poisson solver coping with data transfer costs. Fast Fourier transform was implemented as an specialized coprocessor to accelerate the critical kernel of fast Poisson solution. Our key design issue were to address efficiently four memory banks of FPGA to perform parallel read and write operations every clock cycle. The proposed schemes are efficient because they avoid data replication and do not require many resources of the FPGA to be implemented.

Our results show that the hybrid approach achieves speedups for all tested input data sizes and is able to harness the overall computing power of the hybrid system. Furthermore, the performance gains compared to the reference application grows with the number of input points, indicating that potentially may provide better results for computing meshes with many points.

As future work, we plan to evaluate ways to reduce the use of the internal memory of FPGA by Communication and Twiddle Factor Units to enable the implementation of FFT on higher input set. In addition, we plan to harness the computational power of the CPU while the FPGA performs the FFT computation. Therefore, it is necessary to evaluate the workload partitioning between the devices and the data processing dependency.

References

1. Campbell, S., Meyer, C.: Generalized Inverses of Linear Transformations, ser. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics (2009)
2. de Velho, H.F.C., Claeyssen, J.C.R.: Singular value decomposition in the numerical integration of an atmospheric model. In: XIII Iberian Latin American Congress on Computational Methods in Engineering, CILAMCE 1992, Porto Alegre, RS, BR, pp. 344–353 (1992)
3. Davids, P.J.: Circulant Matrices. John Wiley & Sons, New York (1979)
4. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation 19(90), 297–301 (1965), `http://dx.doi.org/10.2307/2003354`
5. Takahashi, D., Kanada, Y.: High-Performance Radix-2, 3 and 5 Parallel 1-D Complex FFT Algorithms for Distributed-Memory Parallel Computers. J. Supercomput. 15(2), 207–228 (2000)
6. Hemmert, K.S., Underwood, K.D.: An Analysis of the Double-Precision Floating-Point FFT on FPGAs. In: Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 171–180 (2005)
7. Cray Inc., Design of Cray XD1 QDR II SRAM Core, Mendota, MN, USA (2005)
8. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. Sci. Program. 18, 1–33 (2010)
9. Hu, J.: Solution of partial differential equations using reconfigurable computing. Ph.D. dissertation, University of Birmingham (December 2011), `http://etheses.bham.ac.uk/1655/`
10. Zhuo, L., Prasanna, V.K.: High Performance Linear Algebra Operations on Reconfigurable Systems. In: SC 2005: Proc. of the 2005 ACM/IEEE Conference on Supercomputing. IEEE Computer Society, Washington, DC (2005)
11. Bondhugula, U., Devulapalli, A., Dinan, J., Fernando, J., Wyckoff, P., Stahlberg, E., Sadayappan, P.: Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths. In: FCCM 2006: Proc. of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 152–164. IEEE Computer Society, Washington, DC (2006)
12. Lynch, P.: DYNAMO—A one dimensional primitive equation model. Dublin, Irlanda, Tech. Rep. (1984)

An Architecture for IPv6 Lookup Using Parallel Index Generation Units

Hiroki Nakahara[1], Tsutomu Sasao[2], and Munehiro Matsuura[2]

[1] Kagoshima University, Japan
[2] Kyushu Institute of Technology, Japan

Abstract. This paper shows an area-efficient and high-speed architecture for IPv6 lookup using a parallel index generation unit (IGU). To reduce the size of memory in the IGU, we use a liner transformation and a row-shift decomposition. Also, this paper shows a design method for the parallel IGU. A single memory realization requires $O(2^n)$ memory size, where n denotes the length of prefix, while the IGU requires $O(nk)$ memory size, where k denotes the number of prefixes. In IPv6 prefix lookup, since n is at most 64 and k is about 340 K, the IGU drastically reduces the memory size. Since the parallel IGU has a simple architecture compared with existing ones, it performs lookup by using complete pipelines. We loaded more than 340 K IPv6 pseudo prefixes on the Xilinx Virtex 6 FPGA. Its lookup speed is higher than one giga lookups per second (GLPS). As for the normalized area and lookup speed, our implementation outperforms existing FPGA implementations.

1 Introduction

1.1 Demands for Lookup Architecture in IPv6 era

The core routers forward packets by IP-lookup using **longest prefix matching (LPM)**. With the rapid growth of the Internet, LPM has become the bottleneck in the network traffic management. The following conditions must be satisfied to solve the problems:

High Speed Lookup: When a core router works at more than 40 Gbps link throughput (OC-768), it requires more than 125 million lookups per second (MLPS) for a minimum packet size (40 bytes). Now, a 100 Gbps link requires more than 320 MLPS, and the next generation router requires 400 Gbps link.

Low-Power Consumption: R. Tucker predicted that, with the rapid increase of traffic, core routers would dissipate the major part the total network power dissipation [14]. Thus, we cannot use power-hungry ternary content addressable memories (TCAMs). Le *et al.* proposed the memory-based IP lookup architecture on the FPGA, which dissipate lower power than the TCAM [5]. This paper also considers a method that uses a memory-based architecture.

Reconfigurability: On Feb. 3, 2011, IPv4 addresses maintained by Internet Assigned Numbers Authority (IANA) are depleted. Since transition from IPv4 addresses to IPv6

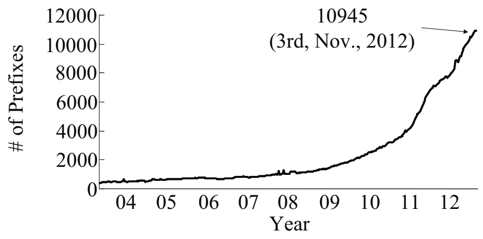


Fig. 1. Numbers of IPv6 prefixes in the routing table for border gateway protocol (BGP)

addresses are encouraged, IPv6 addresses are widely used in core routers. However, since it is a transition period, specifications for IPv6 address are changed frequently[1]. Thus, reconfigurable architecture is necessary to accomodate the changes of specifications.

Large-Capacity: As shown in Fig.1, on Nov. 3, 2012, the number of raw IPv6 address in the border gateway protocol (BGP) was about 10 K. The number of IPv4 addresses increased by 25-50 K prefixes per year [2]. Also, the number of IPv6 addresses increases with the rapid growth. Thus, large-capacity routers are necessary for the future IPv6.

1.2 Proposed Architecture and Contributions of the Paper

This paper proposes a memory-based architecture satisfying the four conditions. When IPv6 prefixes with length n are loaded in a single memory, the amount of memory would be $O(2^n)$, which is too large to implement. In this paper, we use a parallel index generation unit (IGU) that reduces the total amount of memory to $O(kn)$, where k denotes the number of prefixes [9]. Also, since the parallel IGU has a simpler architecture than existing ones, it performs a fast lookup by using pipelines. Our contributions are as follows:

1. We loaded more than 340 K pseudo IPv6 prefixes on the parallel IGU implemented on a single FPGA. Its performance is more than 1 GLPS (Giga lookups per second) lookup. As far as we know, this is the first implementation of 1 GLPS engine on a single FPGA.
2. We reduced the total amount of memory for IGUs by using both a linear transformation and a row-shift decomposition. This paper reports of the first implementation of LPM architecture using the parallel IGU.
3. We compared the parallel IGU with existing implementations on FPGA, and showed that the parallel IGU outperforms others.

The rest of the paper is organized as follows: Chapter 2 introduces an architecture for LPM; Chapter 3 shows the IGU and its memory reduction method; Chapter 4 shows the design method for the parallel IGU; Chapter 5 shows the experimental results; and Chapter 6 concludes the paper.

[1] IPv4-compatible IPv6 addresses are abolished. Also, site-local addresses would be abolished.

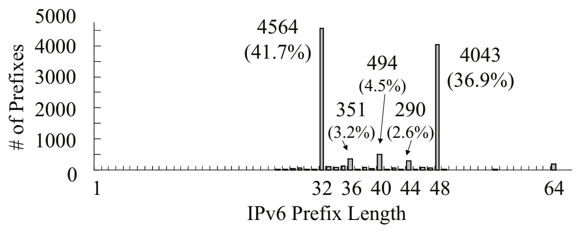


Fig. 2. Distribution of raw IPv6 prefixes (Nov. 3, 2012) [6]

2 Architecture for IPv6 Prefix Lookup

2.1 IPv6 Prefix

The IPv6 address (128 bits) is an extension of the IPv4 address (32 bits). This extension accommodates much larger number of addresses than IPv4. An IPv6 address consists of 64 bits **network prefix (prefix)** and 64 bits interface ID. Since only prefixes are used by the core routers to make forwarding decisions, this paper considers an architecture for the IPv6 prefix lookup. Similar to IPv4 prefix, an IPv6 prefix follows the variable-length subnet masking (VLSM) rule. It consists of n bits network ID and $(64 - n)$ bits sub-network ID. The prefix consists of the following information with bit length:

- FP (Fixed Prefix): 3 bits represented by "001". It means a global unicast accepting route aggregation.
- TLA (Top-Level Aggregation) ID: 13 bits
- sub-TLA: 13 bits
- RES (Reserved for future use): 6 bits
- NLA (Next-Level Aggregation) ID: 13 bits
- SLA (Site-Level Aggregation) ID: 16 bits

Fig. 2 shows the distribution of raw IPv6 prefixes (Nov. 3, 2012) [6]. We observe that variance of the numbers of prefixes with different lengths are quite large. In this paper, we utilize this property to reduce the amount of hardware.

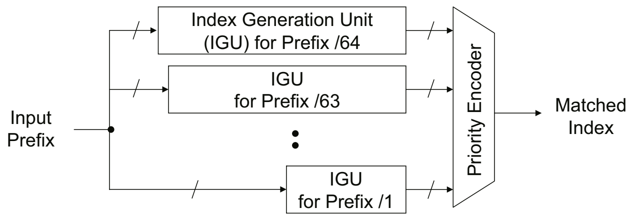


Fig. 3. Architecture for an LPM function

2.2 Longest Prefix Matching (LPM) Function [12]

Definition 2.1. *The* **LPM table** *stores ternary vectors of the form* $VEC_1 \cdot VEC_2$*, where* VEC_1 *consists of* $0's$ *and* $1's$*, and* VEC_2 *consists of* $*'s$ *(don't cares). The*

length *of prefix is the number of bits in* VEC_1*. To assure that the longest prefix address is produced, entries are stored in descending prefix length. The* **LPM function** *is the logic function* $\boldsymbol{f} : B^n \rightarrow B^m$*, where* $\boldsymbol{f}(x)$ *is a minimum address whose* VEC_1 *corresponding to* $\boldsymbol{x}$*. Otherwise,* $\boldsymbol{f}(\boldsymbol{x}) = 0^m$*.*

Let P_l be the subset of the prefixes with length l, and $\mathcal{P} = \{P_1, P_2, \ldots P_s\}$ be a set of subsets of the prefixes. Each P_l is represented by an **index generation function** [10].

Definition 2.2. *[10] A mapping* $F(\boldsymbol{X}) : B^n \rightarrow \{0, 1, \ldots, k\}$*, is an* **index generation function with weight** $\boldsymbol{k}$*, where* $F(\boldsymbol{a}_i) = i$ $(i = 1, 2, \ldots, k)$ *for* k *different* **registered vectors***, and* $F = 0$ *for other* $(2^n - k)$ *non-registered vectors, and* $\boldsymbol{a}_i \in B^n$ $(i = 1, 2, \ldots, k)$*. In other words, an index generation function produces unique indices ranging from 1 to* k *for* k *different registered vectors, and produces 0 for other vectors.*

Example 2.1. *Table 1 shows an index generation function with weight seven.* ■

An LPM function can be decomposed into a set of index generation functions. Thus, this paper focuses on a compact realization of an index generation function.

2.3 Architecture for an LPM Function

Fig. 3 shows an architecture for an LPM function realized by index functions with weight k for P_l and a priority encoder, where k equals to the number of prefixes in P_l. When we realized an index generation function for P_l by a single memory, the memory size becomes $O(2^l)$, which is too large for large l. This paper uses an **index generation unit (IGU)** with $O(k)$ memory size.

Table 1. Example of an index generation function f

x_1	x_2	x_3	x_4	x_5	x_6	f
0	0	0	0	1	0	1
0	1	0	0	1	0	2
0	0	1	0	1	0	3
0	0	1	1	1	0	4
0	0	0	0	0	1	5
1	1	1	0	1	1	6
0	1	0	1	1	1	7

Table 2. Decomposition chart for $f(X_1, X_2)$

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_5
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_4
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_3
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_2
00	0	0	0	0	0	0	0	0	1	2	3	0	0	0	4	0	
01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
10	5	0	0	0	0	0	0	0	0	0	0	0	0	7	0	0	
11	0	0	0	0	0	0	0	0	0	0	6	0	0	0	0	0	
x_6, x_1																	

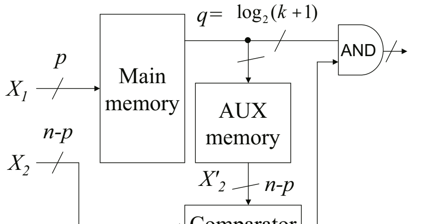
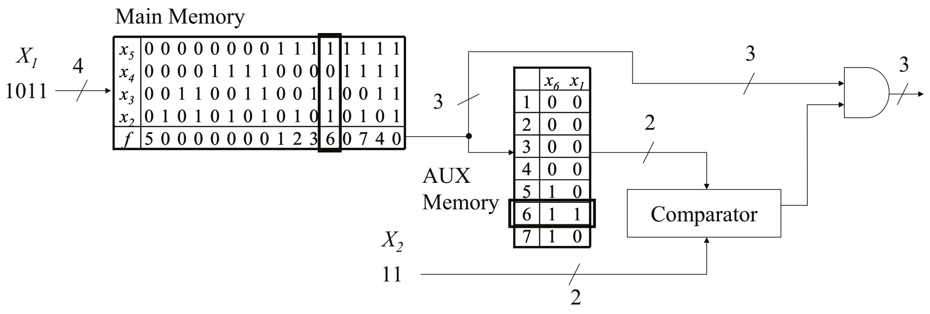


Fig. 4. Index Generation Unit (IGU)

3 Index Generation Unit (IGU)

Table 2 is a **decomposition chart** for the index generation function f shown in Table 1. The columns labeled by $X_1 = (x_2, x_3, x_4, x_5)$ denotes the **bound variables**, and rows labeled by $X_2 = (x_1, x_6)$ denotes the **free variables**. The entry denotes the function value. We can represent the non-zero elements of f by the **main memory** $\hat{f}$ whose input is X_1. The main memory realizes a mapping from a set of 2^p elements to a set of $k+1$ elements, where $p = |X_1|$. The output for the main memory does not always represent f, since $\hat{f}$ ignores X_2. Thus, we must check whether $\hat{f}$ is equal to f or not by using an **auxiliary (AUX) memory**. To do this, we compare the input X_2 with the output for the AUX memory by a **comparator**. The AUX memory stores the values of X_2 when the value of $\hat{f}(X_1, X_2)$ is non-zero. Fig. 4 shows the index generation unit (IGU). First, the main memory finds the possible index corresponding to X_1. Second, the AUX memory produces the corresponding inputs X_2' ($n - p$ bits). Third, the comparator checks whether X_2' is equal to X_2 or not. Finally, the AND gates produce the correct value of f.



Main Memory
X_1
4
1011
x_5 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
x_4 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
x_3 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
x_2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
f 5 0 0 0 0 0 0 0 0 1 2 3 6 0 7 4 0
3
AUX Memory
x_6 x_1
1 0 0
2 0 0
3 0 0
4 0 0
5 1 0
6 1 1
7 1 0
2
Comparator
3
3
X_2
11
2

Fig. 5. IGU for Table 1

Example 3.2. *Fig. 5 shows an example of the IGU realizing the index generation function shown in Table 1. When the input vector is* $X(x_1, x_2, x_3, x_4, x_5, x_6) = (1, 1, 1, 0, 1, 1)$*, the corresponding index is "6". First, the main memory produces the index. Second, the AUX memory produces the corresponding value of* X_2'*. Third, the comparator checks whether* X_2 *and* X_2' *are equal. Since the corresponding input* X_2 *is equal to* X_2'*, the AND gates produces the index. In this case,* $n = 6$*,* $p = 4$*, and* $q = 3$*.* ∎

Example 3.3. *To realize the index generation function* f *shown in Table 1, a single memory realization requires* $2^6 \times 3 = 192$ *bits. On the other hand, in the IGU shown in Fig. 5, the main memory requires* $2^4 \times 3 = 48$ *bits, and the AUX memory requires* $2^3 \times 2 = 16$ *bits. Thus, the IGU requires* 64 *bits in total. In this way, we can reduce the total amount of memory by using the IGU.* ∎

Example 3.2. is an ideal case. Actually, a column may have two or more than non zero-elements. In such a case, the column has a **collision**. When a collision occurs, a main memory cannot realize a function.

Example 3.4. *Table 4 shows a decomposition chart for an index function* f' *shown in Table 3. In Table 4, the first column has a collision for elements "5" and "6".* ∎

Table 3. An index generation function f' causing a collision

x_1	x_2	x_3	x_4	x_5	x_6	f'
0	0	0	0	0	1	1
0	0	0	0	1	0	2
0	0	0	1	0	0	3
0	0	1	0	0	0	4
0	1	0	0	0	0	5
1	0	0	0	0	0	6

Table 4. Decomposition chart for $f'(X_1, X_2)$

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_3
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_4
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_5
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_6
00		1	2		3				4								
01	5																
10	6																
11																	
x_1, x_2																	

Table 5. Decomposition chart for $\hat{f}'(Y_1, X_2)$

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	$y_1 = x_3 \oplus x_1$
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	$y_2 = x_4 \oplus x_1$
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	$y_3 = x_5$
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	$y_4 = x_6$
00		1	2		3				4								
01	5																
10													6				
11																	
x_1, x_2																	

3.1 Linear Transformation [8]

Let $\hat{f}(Y_1, X_2)$ be the function whose variables $X_1 = (x_1, x_2, \ldots, x_p)$ are replaced by $Y_1 = (y_1, y_2, \ldots, y_p)$, where $y_i = x_i \oplus x_j$, $x_i \in \{X_1\}$, $x_j \in \{X_2\}$, and $p \geq \lceil \log_2(k+1) \rceil$. This replacement is called a **linear transformation**, which can avoid a collision.

Example 3.5. *Let f' be an index generation function shown in Table 3. Table 5 shows the decomposition chart for $\hat{f}'(Y_1, X_2)$, where $Y_1 = (x_3 \oplus x_1, x_4 \oplus x_1, x_5, x_6)$, and the column labels denote Y_1, and the row labels denote X_2. In Table 5, since no collision occurs, it can be realized by the IGU shown in Fig. 4.* ■

The linear transformation for p variables is realized by p copies of two-input EXORs. In an FPGA, since these can be realized by p LUTs, their amount of hardware is negligible small.

As shown in Example 3.5., index generation functions often can be represented with fewer variables than original functions. By increasing the number of inputs p for the main memory, we can store virtually all vectors.

Conjecture 3.1. *[9] Consider a set of uniformly distributed index generation functions with weight k (≥ 7). If $p \geq \lceil log_2(k+1) \rceil - 3$, then, more than 95% of the functions can be represented by an IGU with the main memory having p inputs.*

Thus, for the IPv6 prefix lookup problem, a linear transformation of p variables can reduce the amount of memory $O(2^n)$ into $O(2^p)$.

3.2 Row-Shift Decomposition [7]

In this part, we introduce a **row-shift decomposition** to further reduce of memory size for the IGU. Table 6 shows a decomposition chart for the index generation function

Table 6. Decomposition chart for $\hat{f}'(Y_1)$

	0	0	0	0	1	1	1	1	y_1
	0	0	1	1	0	0	1	1	y_2
	0	1	0	1	0	1	0	1	y_3
0	5	2	3		4		6		
1	1								
y_4									

Table 7. Decomposition chart for $\hat{f}'$ after row-shift

	0	0	0	0	1	1	1	1	y_1
	0	0	1	1	0	0	1	1	y_2
	0	1	0	1	0	1	0	1	y_3
0	5	2	3		4		6		
1	$\rightarrow$	$\rightarrow$	$\rightarrow$	1					
y_4									

$\hat{f}'(Y_1, Y_2)$, where $Y_1 = (x_3 \oplus x_1, x_4 \oplus x_1, x_5)$ and $Y_2 = (x_6)$. In Table 6, the first column has a collision for the entries "1" and "5". Consider the decomposition chart shown in Table 7 that is obtained from Table 6 by shifting the rows for $y_4 = 1$ by three bit to the right. Table 7 has at most one non-zero element in each column. Thus, the modified function can be realized by a main memory with inputs Y_1.

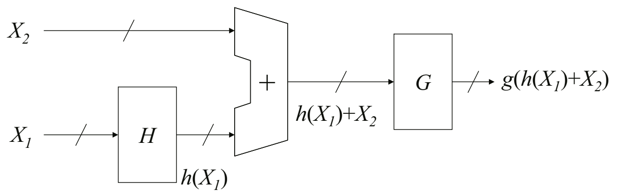
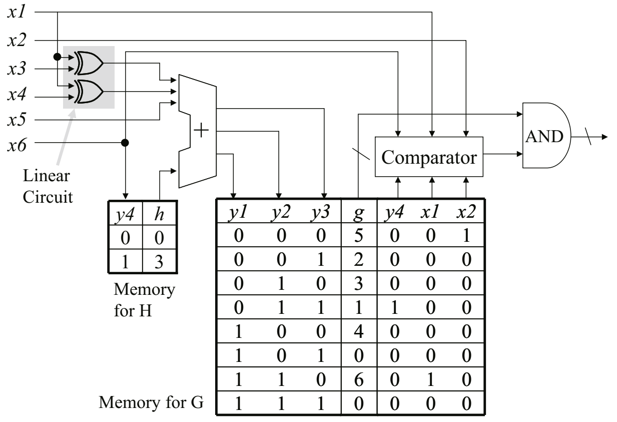


Fig. 6. Row-shift decomposition

Let X_1 be the row variables, and X_2 be the column variables. In Fig. 6, assume that the memory for H stores the number of bits to shift ($h(X_1)$) for each row specified by X_1, while the memory for G stores the non-zero element of the column after the shift operation: $h(X_1) + X_2$, where "+" denotes an unsigned integer addition. We call this a **row-shift decomposition**.



y4	h
0	0
1	3

y1	y2	y3	g	y4	x1	x2
0	0	0	5	0	0	1
0	0	1	2	0	0	0
0	1	0	3	0	0	0
0	1	1	1	1	0	0
1	0	0	4	0	0	0
1	0	1	0	0	0	0
1	1	0	6	0	1	0
1	1	1	0	0	0	0

Fig. 7. IGU using a linear transformation and a row-shift decomposition

Example 3.6. *Fig. 7 shows the IGU using a linear transformation and a row-shift decomposition realizing f' shown in Table 3. Let $\mathcal{Y} = (Y_1, Y_2)$, where $Y_1 = (x_3 \oplus x_1, x_4 \oplus x_1, x_5)$, and $Y_2 = (x_6)$. First, EXOR gates generates $\mathcal{Y}$. Second, the first memory for*

h produces the shift value $h(Y_1)$. Third, the adder produces $h(Y_1) + Y_2$. In this implementation, since we realize both the main memory and the AUX memory by a single memory, the second memory g produces the index and the corresponding (y_4, x_1, x_2) simultaneously. Next, the comparator checks if they are equal or not. Finally, the AND gates produces the correcting index. ■

Example 3.7. *To realize f' shown in Table 3, a single memory realization requires $2^6 \times 3 = 192$ bits. On the other hand, in the IGU shown in Fig. 7, the first memory for H requires $2^1 \times 3 = 6$ bits, and the second memory for G requires $2^3 \times (3 + 3) = 48$ bits. Thus, the IGU requires 54 bits in total. In this way, we can reduce the total amount of memory by using a linear transformation and a row-shift decomposition.* ■

4 Design of Parallel IGU

4.1 Method to Find Linear Transformation

From here, we present a method to find a linear transformation. We assume that the prefix lookup architecture updates its prefix patterns. In this case, it is impractical to find an optimum solution by spending much computation time. To find a reasonably good setting of the EXOR gates, we use the following heuristic algorithm [10], which is simple and efficient.

Algorithm 4.1. *Let $f(X_1, X_2)$ be the index generation function of n variables with weight k, and let $p = \lceil log_2((k + 1)/3)\rceil + 1$ be the number of the bound variables in the decomposition chart.*

1. *Let $\{X_1\} = (x_1, x_2, \ldots, x_p)$ be the bound variables, and $X_2 = (x_{p+1}, x_{p+2}, \ldots, x_n)$ be the free variables.*
2. *While $|X_1| \leq p$, find variables $x_i \in \{X_2\}$ that makes the following value minimum.*

$$|(\text{\# of vectors with} x_i = 0) - (\text{\# of vectors with} x_i = 1)|.$$

   Let $X_1 \leftarrow X_1 \cup \{x_i\}$.
3. *For each pair of variables (x_i, x_j), where x_i is a bound variables, and x_j is a free variables, if the exchange of x_i with x_j decreases the number of collision, then do it, otherwise discard it.*
4. *For each pair of variables (x_i, x_j), if the replacement of x_i with $y_i = x_i \oplus x_j$ decreases the number of collisions, then do it, otherwise discard it.*
5. *Terminate.*

4.2 Design of IGU Using Row-Shift Decomposition [7]

For Table 7, we could represent the function without increasing the columns. However, in general, we must increase the columns to represent the function. Since each column has at most one non-zero element after the row-shift operations, at least k columns are necessary to represent a function with weight k. We use the **first-fit method** [13], which is simple and efficient.

Algorithm 4.2. *(Find row-shifts)*

1. *Sort the rows in decreasing order by the number of non-zero elements.*
2. *Compute the row-shift value for each row at a time, where the row displacement $r(i)$ for row i has the smallest value such that no non-zero element in row i is in the same position as any non-zero element in the previous rows.*
3. *Terminate.*

When the distribution of non-zero elements among the rows is uniform, Algorithm 4.2. reduces the memory size effectively. To reduce the total amount of memories, we use the following:

Algorithm 4.3. *(Row-shift decomposition)*

1. *Reduce the number of variables by the method [9]. If necessary, use a linear transformation [8] to further reduce the number of the variables. Let n be the number of variables after reduction.*
2. *Let $q_1 \leftarrow \lceil \frac{n}{2} \rceil$. From $t = -2$ to $t = 2$, perform Steps 3 through 6.*
3. *Partition the inputs X into (X_1, X_2),where $X_1 = (x_p, x_{p-1}, \ldots, x_1)$ denotes the rows, and $X_2 = (x_n, x_{n-1}, \ldots, x_{p+1})$ denotes the columns.*
4. *$p \leftarrow q_1 + t$.*
5. *Obtain the row-shift value by Algorithm 4.2..*
6. *Obtain the maximum of the shift value, and compute the total amount of memories.*
7. *Find t that minimizes the total amount of memories.*
8. *Terminate.*

x1	*x2*	*x3*	*x4*	*f*
0	0	0	1	1
0	0	1	*	2
0	0	*	*	3

Prefix Expansion $\Longrightarrow$

x1	*x2*	*x3*	*x4*	*f*
0	0	0	1	1
0	0	1	0	2
0	0	1	1	2
0	0	0	0	3

Fig. 8. Example of prefix expansion

4.3 Prefix Expansion

Let P_l be the set of the prefixes with length l, and let $\mathcal{P} = \{P_1, P_2, \ldots P_s\}$ be the set of the set of the prefixes. As shown in Fig. 3, the parallel IGU consists of s IGUs and a priority encoder whose size is proportional to s. Thus, the straightforward implementation requires large amount priority encoder and many IGUs.

To reduce the number of required IGUs, we merge multiple P_l into a group. By expanding the prefixes in P_l to ones with length $l + 1$, we can make a group including P_{l+1} and P_l. We call this **prefix expansion**. The next example shows it.

Example 4.8. *The left-hand side Table in Fig. 8 stores $\{P_2, P_3, P_4\}$, where $P_2 = \{00{*}{*}\}$, $P_3 = \{001{*}\}$, $P_4 = \{0001\}$. By performing prefix expansion to P_2, we have $P_3' = \{000{*}, 001{*}\}$. By the longest prefix matching (LPM) rule, the prefix $\{001{*}\}$ that is equal to $\{001{*}\}$ in P_3 is ignored. Also, by performing prefix expansion to P_3', we have P_4' shown in the right-hand side Table in Fig. 8.* ■

Table 8. The number BRAMs to realize IGUs with non-uniform grouping (BRAMs marked with "*" were realized by distributed RAMs in the actual implementation)

Group	#prefixes	Memory H		Memory G		# of 36Kb
	in a group	#In	#Out	#In	#Out	BRAMs
(15,16,17,18)	102	4	6	7	18	2 *
(19,20,21,22)	225	7	7	8	22	2 *
(23,24,25,26)	1,571	10	11	11	26	3 *
(27,28)	806	6	11	11	28	3 *
(29,30)	1,240	6	12	12	30	5 *
(31)	2,824	9	12	12	31	5 *
(32)	8,474	9	14	14	32	16
(33)	1,469	8	11	11	33	3 *
(34)	4,408	10	12	12	34	5 *
(35)	2,318	10	11	13	35	9
(36)	6,957	11	13	13	36	9
(37)	4,079	13	12	12	37	8
(38)	12,237	14	14	14	38	24
(39)	6,592	12	12	13	39	11
(40)	19,776	13	14	14	40	22
(41)	6,874	13	13	13	41	12
(42)	20,623	14	15	15	42	42
(43)	9,451	14	14	14	43	27
(44)	28,354	13	15	15	44	47
(45)	8,522	13	14	14	45	24
(46)	25,569	14	15	15	46	48
(47)	42,768	14	16	16	47	92
(48)	128,305	14	17	17	48	179
(49,50)	929	10	10	10	50	3*
(51,52)	1,048	11	11	11	52	4*
(53,54)	594	9	10	10	54	3*
(55,56)	421	8	9	9	9	2*
(57,58)	530	9	9	9	58	2*
(59,60,61,62)	289	7	8	9	62	2*
(63,64)	386	8	9	9	64	2*
Total	347,749					616

Table 9. Comparison of non-uniform grouping with uniform ones

Grouping	#prefixes	#groups	#BRAMs	#Slices
Non-uniform (Table 8)	347,749	30	616	2,299
without grouping (direct realization of $\mathcal{P}$)	348,877	50	655	3,979
Uniform for two subsets	382,132	25	785	1,899
Uniform for four subsets	1,250,695	13	2,512	939

Fig. 2 shows that the variance of the numbers of prefixes with different lengths P_l is quite large. When the prefix expansions to P_l consisting of a small number of prefixes is applied, they can be stored into a single BRAM [2]. On the other hand, when the prefix expansion to P_l consisting of a large number of prefixes is applied, in the worst case, the size would exceed that of the available BRAMs. Thus, we make a **non-uniform grouping** G_i $\mathcal{G} = \{G_1, G_2, \ldots, G_r\}$, where G_i is generated from the different number of P_l. To find an optimal grouping without increasing BRAMs, we use the following:

Algorithm 4.4. *(Non-uniform grouping) Let P_l be the set of the prefixes with length l, $\mathcal{P} = \{P_1, P_2, \ldots, P_s\}$ be the set of the prefixes, G_j consists of single or several P_ls, r be the number of groups, and $\mathcal{G} = \{G_1, G_2, \ldots, G_r\}$, where $r \leq s$.*

[2] For Xilinx Virtex 6 FPGA, the BRAM stores 36Kbits.

```
1.   Apply Algorithms 4.1. and 4.3. to $P_l$ $(1 \leq l \leq s)$ to generate the IGU. Let $B_l$ be
     the number of BRAMs to realize the IGU.
2.   $r \leftarrow 1$, and $i \leftarrow 1$.
3.   $G_r \leftarrow P_i$, and $B \leftarrow B_i$.
4.   $i \leftarrow i + 1$. If $i > s$, then go to Step.7.
5.   Perform a prefix expansion to $G_r \cup P_i$, then apply Algorithms 4.1. and 4.3. to them
     to generate the IGU. Let $B_{temp}$ be the number of BRAMs to realize the IGU.
6.1. If $B + B_i \geq B_{temp}$, then $G_r \leftarrow G_r \cup P_i$, $B \leftarrow B_{temp}$, and go to Step.4.
6.2. $r \leftarrow r + 1$, and go to Step.3.
7.   Terminate.
```

Table 10. Comparison with existing FPGA implementations

Architecture	#prefixes	#Slices	# of 36Kb BRAMs	Off-chip SRAM [Mb]	Normalized area		Throughput [MLPS]
					#Slices	#BRAMs	
Baboescu et al. (ISCA2005) [1]	80 K	1,405	530	—	17.5	6.6	125
Fadishei et al. (ANCS2005) [3]	80 K	14,274	254	—	178.4	3.1	263
Le et al. (FCCM2009) [5]	249 K	16,617	473	—	66.7	1.8	340
2-3-tree-IPv6 (IEEE Trans.2012) [4]	330 K	15,358	580	32.5	46.5	4.5	373
BST-IPv6 (IEEE Trans.2012) [4]	330 K	14,096	1,025	3.2	42.7	3.3	390
Parallel IGU	340 K	5,577	575	—	**16.4**	**1.7**	**1,002**

5 Experimental Results

5.1 Implementation of the Parallel IGU

We designed the parallel IGU using Xilinx's PlanAhead 14.2, and implemented it on the Roach2 board (FPGA: Xilinx Virtex-6 (XC6VSX475T), 74,400 Slices,1,064 BRAMs (36Kb)). Pseudo IPv6 prefixes were generated from the present raw 340 K IPv4 prefixes (Nov. 3, 2012) using a method [15]. Since the present IPv6 uses prefixes with length 15 or more, we generated such prefixes only.

Table 8 shows the number of BRAMs in IGUs to load 340 K pseudo IPv6 prefixes generated by Algorithm 4.4. Table 9 compares non-uniform grouping with uniform one. In Table 9, *#Slices* includes the number of slices for both IGUs and the priority encoder. Table 9 shows that, the number of BRAMs for the non-uniform grouping is smaller than that for the uniform grouping. Although the non-uniform grouping requires more slices than the uniform one, it consumes less than 10% of the FPGA available resources.

Since we implemented small memory part (marked with "*" in Table 8) by distributed RAMs instead of BRAMs, they consumed 3,288 slices. Thus, the parallel IGU used 5,577 slices and 575 BRAMs. Since we implemented a complete pipeline architecture, the maximum clock frequency was 501.4 MHz. By using a dual port BRAM, the lookup speed for the parallel IGU was 1,002 MLPS (mega lookups per second).

5.2 Comparison with Existing Implementations

Table 10 compares the parallel IGU with existing implementations. Since existing implementations store different numbers of prefixes to compare the efficiency, we used the normalized area, which shows the number of primitives (# of slices or BRAMs) per a prefix. As for the off-chip SRAMs, we converted them to the equivalent of 36Kb BRAM numbers. Table 10 shows that, as for the lookup speed, the parallel IGU is 2.56-8.01 times faster than existing implementations. As shown in Fig. 7, the parallel IGU has a simple architecture which is suitable for pipelined implementation to increase the throughput. Also, as for the normalized area, the parallel IGU has the smallest implementation. Therefore, the parallel IGU outperforms existing FPGA realizations.

6 Conclusion

This paper showed the parallel IGU for IPv6 Lookup. To reduce the memory size of the IGU, we used linear transformation and row-shift decompositions. Also, this paper showed a design method for the parallel IGU. We implemented the parallel IGU on the Xilinx Virtex 6 FPGA, it loaded more than 340 K IPv6 prefixes, and its lookup speed is 1,002 MLPS. Experimental results showed that our implementation outperforms existing FPGA realizations.

Acknowledgments. This research is supported in part by Strategic Information and Communications R&D Promotion Program (SCOPE), and the Grants in Aid for Scientistic Research of JSPS.

References

1. Baboescu, F., Tullsen, D.M., Rosu, G., Singh, S.: A tree based router search engine architecture with single port memories. In: ISCA 2005, p. 123 (2005)
2. Chao, H.J., Liu, B.: High performance switches and routers. JohnWiley& Sons, Inc., Hoboken (2007)
3. Fadishei, H., Zamani, M.S., Sabaei, M.: A novel reconfigurable hardware architecture for IP address lookup. In: ANCS 2005, pp. 81–90 (2005)
4. Le, H., Prasanna, V.K.: Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning. IEEE Trans. on Compt. 61(7), 1026–1039 (2012)
5. Le, H., Prasanna, V.K.: Scalable high throughput and power efficient IP-lookup on FPGA. In: FCCM 2009 (April 2009)
6. University of Oregon route views project, `http://www.routeviews.org/`
7. Sasao, T.: Row-shift decompositions for index generation functions. In: DATE 2012, pp. 1585–1590 (2012)
8. Sasao, T.: Linear decomposition of index generation functions. In: ASPDAC 2012, pp. 781–788 (2012)
9. Sasao, T.: Memory-based logic synthesis. Springer (2011)
10. Sasao, T., Matsuura, M., Nakahara, H.: A realization of index generation functions using modules of uniform sizes. In: IWLS 2010, June 18-20, pp. 201–208 (2010)
11. Sasao, T.: On the number of variables to represent sparse logic functions. In: ICCAD 2008, San Jose, California, USA, November 10-13, pp. 45–51 (2008)

12. Sasao, T., Butler, J.T.: Implementation of multiple-valued CAM functions by LUT cascades. In: ISMVL 2006, Singapore, May 17-20 (2006)
13. Tarjan, R.E., Yao, A.C.-C.: Storing a sparse table. Communications of the ACM 22(11), 606–611 (1979)
14. Tucker, R.: Optical packet-switched WDM networks: a cost and energy perspective. In: OFC/NFOEC (2008)
15. Wang, M., Deering, S., Hain, T., Dunn, L.: Non-random generator for IPv6 tables. In: HOTI 2004, pp. 35–40 (2004)

Hardware Index to Set Partition Converter

Jon T. Butler[1] and Tsutomu Sasao[2,⋆]

[1] Naval Postgraduate School, Monterey, CA, 93921-5121, USA
jon_butler@msn.com
[2] Kyushu Institute of Technology, Iizuka, Fukuoka, 820-8502, Japan
sasao@ieee.org

Abstract. We demonstrate, for the first time, high-speed circuits that generate partitions on a set S of n objects. We offer two versions. In the first, partitions are produced in lexicographical order in response to successive clock pulses. In the second, an index input determines the set partition produced. Such circuits are needed in the hardware implementation of the optimum distribution of tasks to processors. Our circuits are combinational. For large n, they can have large delay. However, one can easily pipeline them to produce one set partition per clock period. We show 1) analytical and 2) experimental time/complexity results that quantify the efficiency of our designs. Our results show that a hardware partition generator running on a 100 MHz FPGA produces partitions at a rate that is approximately 10 times the rate of a software implementation on a processor running at 2.26 GHz.

1 Introduction

A partition of a set S is the placement of elements of S into blocks. For example, there are 15 partitions of four distinct elements 0, 1, 2, and 3. These are $\{\{3,2,1,0\}\}$ (all elements in the same block), $\{\{3,2,1\},\{0\}\}$, $\{\{3,2,0\},\{1\}\}$, $\{\{3,2\},\{1,0\}\}$, $\{\{3,2\},\{1\},\{0\}\}$, $\{\{3,1,0\},\{2\}\}$, $\{\{3,1\},\{2,0\}\}$, $\{\{3,1\},\{2\},\{0\}\}$, $\{\{3,0\},\{2,1\}\}$, $\{\{3\},\{2,1,0\}\}$, $\{\{3\},\{2,1\},\{0\}\}$, $\{\{3,0\},\{2\},\{1\}\}$, $\{\{3\},\{2,0\},\{1\}\}$, $\{\{3\},\{2\},\{1,0\}\}$, and $\{\{3\},\{2\},\{1\},\{0\}\}$ (all elements in separate blocks). Neither the order of the blocks, nor the order of elements within each block matters. For example, partitions $\{\{3,1\},\{2,0\}\}$ and $\{\{0,2\},\{1,3\}\}$ are identical. The number of partitions increases rapidly as the number of elements increases, and are counted by the *Bell* numbers $B(n)$. For example, for sets of size $n = 2,\ 3,\ 4,\ 5,\ 6,\ 7,$ and 8, the number of set partitions is $B(n) = 2,\ 5,\ 15,\ 52,\ 203,\ 877,$ and 4140. Bell numbers have the property that, for large n, $B(n)$ is approximated by $(\frac{n}{\ln n})^n$ [8], p. 64.

Partitions are important combinatorial objects. For example, partitions on n elements enumerate the equivalence relations on n elements [17]. Each block represents all elements related by the equivalence relation.

⋆ This research is supported by a Grant-in-Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS).

One way to generate all partitions is to generate all binary numbers, one per clock, discarding those that are not partitions. However, only a few of these numbers are partitions. This approach produces partitions at a rate that is much slower than one partition per clock. Therefore, we seek a circuit that produces one partition per clock, where the input is an index to the partitions.

The ability to generate partitions has important practical applications. Hankin and West [7] show how partitions are used to solve optimization problems in bioinformatics, forensic science, and scheduling. For example, set partitions can be used to specify the ways tasks are allocated to processors, from which one seeks the partition that corresponds to the shortest computation time. This last application especially requires high-speed enumeration of partitions. Recent research in computational molecular biology has shown the importance of partitions in understanding the role of genes in determining global characteristics of species. For example, Chen, Liu, Liu, and Jiang [4] have identified the importance of solving the minimum common integer partition (MCIP) problem in ortholog assignment and DNA fingerprint assembly. This problem requires the enumeration of partitions at high speed, since so many partitions must be considered. In multi-state distribution *systems* (packet, water, gas, etc.) [11], the overall quality of service is dependent on attributes of the components, as measured by variables. There is a need to quickly enumerate partitions of the variables used in decision diagrams that model the system.

This paper can be viewed as a companion to [2], which describes the high-speed generation of combinations, as well as the generation of random combinations for use in reconfigurable computers. It can also be viewed as a companion to [3], which describes the high-speed generation of permutations, as well as the generation of random permutations. Together these three papers cover a subset of circuits that produce *combinatorial objects.* The advent of large programmable logic circuits has allowed computations to be performed in hardware that previously could only be done in software, but at a much higher rate. Much has been written about generating combinatorial objects in software (e.g. [6], [8] pp. 5-6). Indeed, there are many papers on programs and algorithms for enumerating partitions [6,9,10,12,14,16], including parallel algorithms [17]. However, as far as we know, there has not been a hardware enumeration of partitions. This paper addresses that deficiency.

2 Definitions

2.1 Introduction

Definition 1. *Given an n-set* $S = \{0, 1, \ldots, n-1\}$, $\{S_0, S_1, \ldots, S_{n-1}\}$ *is a* **partition** *of* S *iff 1)* $S_i \subseteq S$, *2)* $S_i \bigcap S_j = \emptyset$ *for* $i \neq j$, *and 3)* $\bigcup_{i=0}^{n-1} S_i = S$.

For example, $\{\{3, 1\}, \{2, 0\}\}$ is a partition on the 4-set $S = \{0, 1, 2, 3\}$. It is convenient to represent a partition in its restricted growth string form, as follows. Since a set partition is unchanged by a reordering of blocks, call the block in which $n-1$ is located block 0. Then, $n-2$ is either in the same block, block 0, or in a different block. If it is in a different block, call that block 1. Then, $n-3$ is either in block 0 or 1 or some other block. If it is in some other block, call that block 2. Continue in this way until all elements are assigned a block. For example, the partition $\{\{3,1\},\{2,0\}\}$ has the restricted growth string (0101). Formally,

Definition 2. *An n-element* **restricted growth string** *is a sequence* $(b_0 b_1 \ldots b_{n-1})$ *such that* $b_0 \leq b_i \leq \max_{0 \leq j < i} b_j + 1$*, where* $b_0 = 0$[1]*.*

The first element of a restricted growth string is always 0, signifying that element $n-1$ is always in block 0. A special characteristic of a restricted growth string is that each element is between 0 and 1 plus the maximum of all lower elements.

Lemma 1. *[13] There is a bijection between the set of partitions of an n-set and the set of n-element restricted growth strings.*

The one-to-one relation between partitions and restricted growth strings means that we can enumerate the latter with a guarantee that we enumerate the former. Especially, a circuit exists to convert restricted growth strings into partitions. Table 1 shows the set of all 15 partitions on $n = 4$ elements $\{3,2,1,0\}$. The first column shows the index i, where $0 \leq i \leq 14$. i indexes the set partitions according to the increasing lexicographical order of the restricted growth strings. The second column shows how the actual partition distributes the elements $\{3,2,1,0\}$ into blocks. Here, commas separate blocks and elements within the same block. The third column shows the restricted growth string. Each restricted growth string begins in 0, indicating that 3 is (always) in the first (0-th) block. The second element shows where element 2 is located (in the 0-th or 1-st block). The third element shows where element 1 is located (in the 0-th, 1-st, or 2-nd block). The fourth element shows where element 0 is located (in the 0-th, 1-st, 2-nd, or 3-rd block).

In order to deduce the circuit needed to produce a set partition from an index, we introduce the set partition tree. Specifically, the methodology to design a hardware index to set partition converter uses a tree structure to store all partitions on a set $\{n-1, n-2, \ldots, 1, 0\}$ of n elements.

Definition 3. *A* **set partition tree** *for n consists of three node-types*

1. *the single* **root** *node labeled* 0*,*
2. **internal** *nodes labeled i, for all* $i \in \{0, 1, \ldots, n-2\}$*,*
3. **terminal** *nodes labeled i, for all* $i \in \{0, 1, \ldots, n-1\}$*,*

and one edge-type

1. *an* **edge** *connects a node labeled i to a node labeled j iff along the path from the root node to j, there is no more nodes than n, and, for all node labels k,* $j \leq \max\{k\} + 1$.

[1] The term restricted growth *function* is also used to describe this.

Table 1. Partitions on a set of $n = 4$ Versus Their Index i

i	Partition	Restricted Growth String
0	$\{\{3,2,1,0\}\}$	(0 0 0 0)
1	$\{\{3,2,1\},\{0\}\}$	(0 0 0 1)
2	$\{\{3,2,0\},\{1\}\}$	(0 0 1 0)
3	$\{\{3,2\},\{1,0\}\}$	(0 0 1 1)
4	$\{\{3,2\},\{1\},\{0\}\}$	(0 0 1 2)
5	$\{\{3,1,0\},\{2\}\}$	(0 1 0 0)
6	$\{\{3,1\},\{2,0\}\}$	(0 1 0 1)
7	$\{\{3,1\},\{2\},\{0\}\}$	(0 1 0 2)
8	$\{\{3,0\},\{2,1\}\}$	(0 1 1 0)
9	$\{\{3\},\{2,1,0\}\}$	(0 1 1 1)
10	$\{\{3\},\{2,1\},\{0\}\}$	(0 1 1 2)
11	$\{\{3,0\},\{2\},\{1\}\}$	(0 1 2 0)
12	$\{\{3\},\{2,0\},\{1\}\}$	(0 1 2 1)
13	$\{\{3\},\{2\},\{1,0\}\}$	(0 1 2 2)
14	$\{\{3\},\{2\},\{1\},\{0\}\}$	(0 1 2 3)

A terminal node is simply the last node along a path from the root node. In a set partition tree, the restricted growth string of a partition is represented by the labels of edges along a path from the root node to a terminal node. Each node in a path specifies a block in which the corresponding element is located.

Example 1. *Fig. 1 shows the set partition tree for partitions with $n = 4$ elements. Following the leftmost path from the root node to a terminal node yields the node labels* (0000)*. This restricted growth string specifies that all elements,*

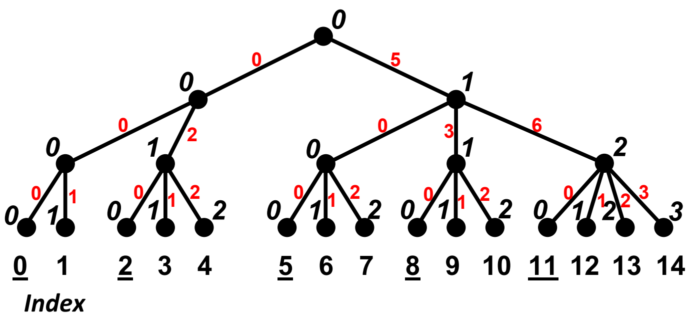


Fig. 1. Example of a Set Partition Tree for Set Partitions on Four Elements

3,2,1, and 0 belong to block 0. That is, this is the partition in which all elements are in a single block. Following the rightmost path yields the node labels (0123). *This restricted growth string specifies the partition in which all elements are in different blocks. Following the path with node labels* (0102) *yields a partition with 3 and 1 in the same block and 2 and 0 each in separate blocks with just one element.* *(End of Example)*

The set partition tree is similar to a decision tree. Each node has children nodes corresponding to all possible choices at that point. Each terminal node corresponds to a partition. Fig. 1 shows, as (additional) terminal labels the index of the partition. There are 15 partitions in this example, labeled 0, 1, ... , and 14.

Note that edges are labeled by the part that each contributes to the index. For example, the edge from the root node to the node labeled 1 has weight 5. This is because the indices on the right side of the tree corresponding to the latter node all have index 5 or greater. It follows that the index associated with each node can be obtained by summing the weights in edges associated with the path from the root node to the corresponding terminal node.

3 Circuit Implementations

3.1 Sequential Circuit Implementation

Fig. 2 shows a sequential circuit implementation of a set partition converter. A clock comes in at the right. At each clock pulse, this circuit produces the next set partition in increasing lexicographical order according to the restricted growth string. Specifically, it first generates $(b_0 \ldots b_{n-2}b_{n-1}) = (0 \ldots 000)$, then $(0 \ldots 001)$, etc.. At each stage, `Counter` counts up to a maximum value allowed in a restricted growth string representation. At this point, it cycles back to 0, just as is done in a conventional counter digit. The count finishes when b_{n-1} is $n-1$. At this point, `Done` is asserted. This could be used externally or it could stop the clock, preventing the circuit from receiving further clock pulses.

3.2 Single-Stage Combinational Circuit Implementation

Fig. 3 shows the single-stage index to set partition circuit for partitions of size $n = 4$. The index comes in on the left, and is tested by five comparators. These test the range of the index, and determine the first three elements of the restricted growth string. There are five possibilities, 000, 001, 010, 011, and 012. One of these five is applied to the one-hot MUX that drives the output. Also, the threshold is subtracted from the incoming index and the result applied to the output as the LSD or least significant digit. The threshold values in Fig. 3 are determined by the set partition tree shown in Fig. 1. They correspond to the indices associated with the 0 terminal nodes in Fig. 1. The corresponding indices are underlined in Fig. 1. There is only one stage in this implementation. As is discussed later, the number of comparators grows is approximated by $(\frac{n}{\ln(n)})^n$.

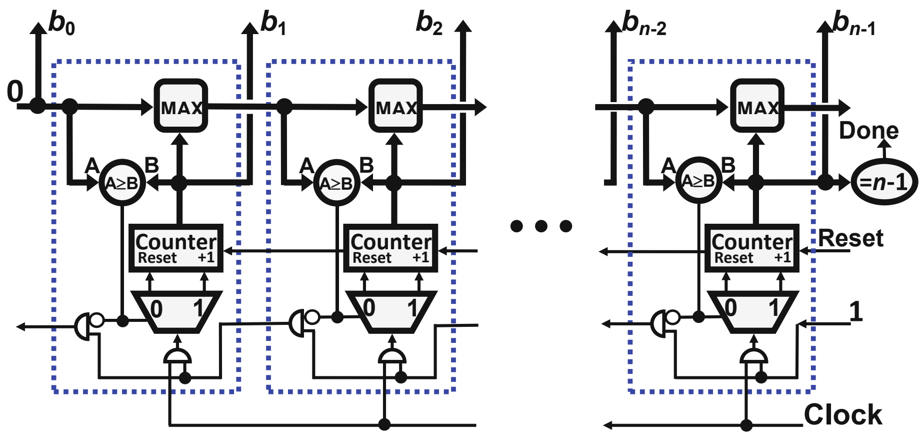


Fig. 2. Sequential Set Partition Generator

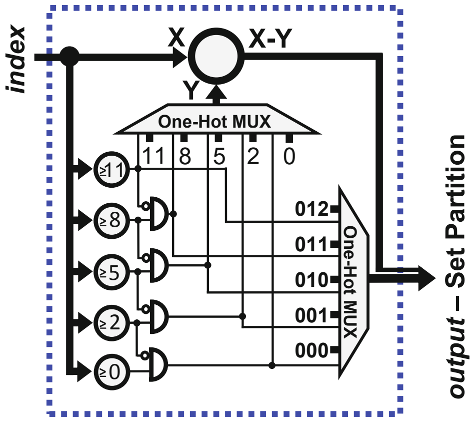


Fig. 3. Single-Stage Index to Set Partition Circuit for $n = 4$

3.3 Multi-stage Combinational Circuit Implementation

Fig. 4 shows the multi-stage index to set partition converter. Here, the index comes in on the left and is modified as it passes through the stages. At each stage, an element in the restricted growth string of the set partition is computed. For example, in the left stage, b_1 is determined. From Fig. 1, it can be seen that, if the index is 4 or less, b_1 is 0. Conversely, if the index is 5 or greater b_1 is 1. It follows that the threshold A in Fig. 4 is 5. Also, if the index is 5 or more, 5 is subtracted from the index and is passed to the next stage. Recall that the thresholds against which the index is compared vary according to the maximum value in the restricted growth string computed so far. In the leftmost stage, the output value of `MAX` is 0 or 1. This is passed to the next stage, which uses it to determine the two threshold values A and B.

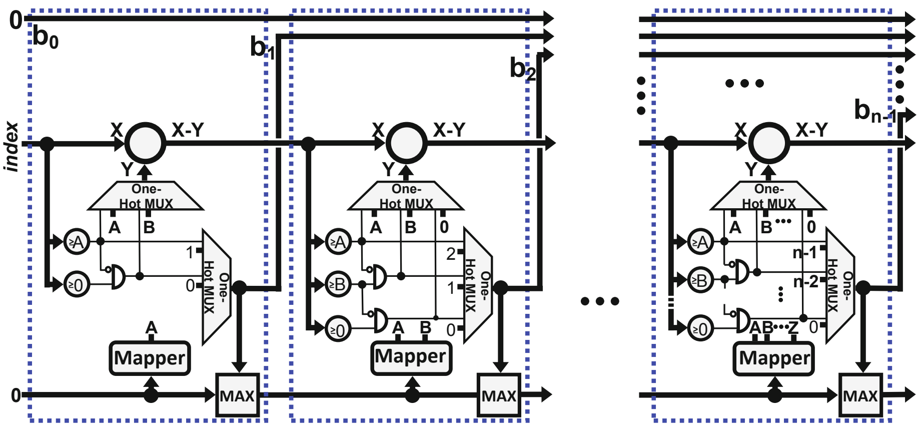


Fig. 4. Multi-Stage Index to Set Partition Circuit

Note that, in a multi-stage index to set partition converter for $n = 4$, there are nine comparators. The single-stage index to set partition converter has five. This raises the question of which circuit is the more compact for general n. This is addressed in the next section.

3.4 Circuit Complexity and Delay

Note that all three circuits use comparators. Further, the complexity of the other parts of the circuit is proportional to the number of comparators. For example, the number of AND gates is nearly the same as the number of comparators, and the one-hot MUX circuits have about as many inputs as the number of comparators. Therefore, it will be convenient to measure the circuit's complexity by the number of comparator it contains. Note that, in making this assumption, we assume that the delay and circuit complexity for comparators and multipliers remains constant as n varies.

Lemma 2. *The number of comparators C_i used in a set partition generator is*

1) sequential (Fig. 2): $C_1 = O(n)$,
2) single-stage (Fig. 3): $C_2 = O\left(\left(\frac{n}{\ln(n)}\right)^n\right)$, *and*
3) multi-stage (Fig. 4): $C_3 = O(n^2)$.

Proof
In the case of the sequential set partition generator, each stage has one comparator, and there are $n-1$ stages.

In the case of the set partition tree, the number of comparators is just the number of set partitions on $n-1$, which is $C_2 = B(n-1)$, where $B(n-1)$ is the $n-1$-th Bell number. From Berend and Tassa [1], we have $B(n-1) < \left(\frac{0.792(n-1)}{\ln(n)}\right)^{n-1}$. Thus,

$$C_2 = O\left(\left(\frac{n}{\ln(n)}\right)^n\right). \tag{1}$$

In the case of the set partition tree, the first (leftmost) block has 2 comparators. The next block has 3, the next 4, etc.. There are a total of $n-2$ blocks. Thus, $C_3 = \sum_{i=2}^{n-2} i = \frac{n(n+1)}{2} - 2n + 4$, and we can write

$$C_3 = O(n^2). \tag{2}$$

∎

It is clear from Lemma 2 that the multi-stage index to set partition converter has many fewer comparators than the single-stage converter, especially in the case of set partitions on many elements. Thus, the case for $n = 4$ discussed at the end of Section 3.3 is an aberration. We can also compare the circuits on the basis of their delay.

Lemma 3. *The delay D_i in a set partition generator is*

1) sequential (Fig. 2): $D_1 = O(n)$,
2) single-stage (Fig. 3): $D_2 = O(1)$, and
3) multi-stage (Fig. 4): $D_3 = O(n)$.

Proof
In the case of the sequential set partition generator, there are $n-1$ stages through which a signal must pass. In the case of the set partition tree, there is exactly one stage, and the delay is independent of n. Thus, this circuit has delay $O(1)$. In the case of the compact set partition tree, the index must propagate through $n-2$ stages. Thus, the delay is $O(n)$. ∎

Note that, in these calculations, we considered the multi-stage index to set partition converter to be *combinational.* When n is large, this circuit has large delay. In order to improve the throughput, we will create a pipelined circuit by inserting registers between stages. In the next section, we compare the experimental delay of a *pipelined* version of the multi-stage circuit with the combinational circuit of the single-stage circuit. As a result, the time comparisons will be (significantly) different from the derived delay.

3.5 Experimental Data

In the analysis above, we used the number of comparators as a measure for the complexity. In this section, we use actual FPGA resources. We synthesized the three circuits discussed above on the Altera Stratix IV EP4SE530F43C3NES FPGA. Table 2 shows the resource usage for the sequential version.

Table 2. Frequency/resources used to realize the sequential set partition generator on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	# Set Partitions	In # Bits	Out # Bits	Freq. (MHz)	Delay ns.	# Comb Fnc	# LUTs vs # inputs 7-	6-	5-	4-	3-	Est. # of Packed ALMs
5	52	6	15	236.2	4.234	42	1	1	12	8	20	22(0%)
6	203	8	18	172.6	5.793	60	2	5	19	13	21	34(0%)
7	877	10	21	156.4	6.393	58	3	1	15	9	30	31(0%)
8	4,140	13	24	130.8	7.643	63	3	3	13	11	33	35(0%)
16	1.05×10^{10}	34	64	122.1	8.190	245	5	28	87	61	64	135(0%)
32	1.28×10^{26}	88	160	53.0	18.863	741	3	231	221	94	192	459(0%)
64	1.72×10^{65}	217	384	25.9	38.584	1923	10	477	659	298	479	1146(0%)
128	1.12×10^{158}	526	896	11.0	91.278	3847	13	727	1666	427	1014	2134(1%)

From Table 2, for all values of $n \leq 16$, the achieved frequency exceeds 100 MHz. Thus, the sequential partition generator produces one partition per clock period for all $n \leq 16$, where the clock period is 10 ns.. To compare this rate to a software implementation of a sequential partition generator, we adapted Orlov's [13] program and ran it on an Intel®Core$^{\text{TM}}$2 Duo P8400 processor running at 2.26 GHz. For 8 and 16 element partitions, we achieve a rate of partitions of one per 94 ns. and 156 ns., respectively. This represents a 9.4 and 15.6 times speed-up realized by the hardware version over the software version.

The first column in Table 2 shows n. The second column shows the number of set partitions, the third column shows the number of input bits, and the fourth column shows the number of output bits. All remaining columns show circuit parameters provided by the synthesis tool, Synplify Pro. The fifth column shows the frequency specified by Synplify Pro. The corresponding delay is shown in the sixth column. The seventh column shows the number of combinational functions used in the realization. This is an overall measure of the logic resources used; it is generated in the first step of the synthesis, prior to the technology mapping process. The eighth through twelfth columns show the number of the various lookup tables (LUTs) that were used. The thirteenth column shows the number of packed ALMs used in the realization. The columns that represent *general* characteristics, including the number of combinational functions and the number of packed ALMs, show an approximate doubling of resources used with each doubling of n. This suggests a linear relationship between these resources and n. It correlates with the observed linear relationship between the number of comparators measure used in the previous section and n. Because of the large number of partitions, for moderate n (e.g., $n = 32$), it will be too time consuming to enumerate all partitions at typical FPGA clock frequencies (e.g., 100 MHz). However, such designs are useful in understanding the complexity/delay of these circuits. For index to set partition generators, however, even large n is useful, for example, if the index is random and the partitions are used in Monte Carlo simulations.

Table 3 shows the FPGA resources and frequency achieved on the Altera Stratix IV EP4SE530F43C3NES FPGA by the single-stage combinational logic index to set partition converter shown in Fig. 3. As discussed, this has short delay paths. This is indicated by the frequency, which has a relatively shallow decline as n, the number of elements, increases. Also, as discussed, this circuit has high complexity. This can be seen in Table 3 by the near 5-fold increase in the number of combinational logic circuits and by the nearly 5-fold increase in the number of ALMs as n increases by 1. For the single-stage circuit, it was possible to achieve an n of only 8, which is significantly smaller that the values of n achieved for the sequential and multi-stage circuits. In comparing the delay of the single-stage combinational logic index to the set partition converter with the multi-stage circuit below, it is important to recall that, unlike the multi-stage circuit, the single-stage circuit is not pipelined. Thus, the multi-stage circuit will achieve a higher clock speed. However, its latency will be larger.

Table 3. Frequency/resources used to realize the single stage index to set partition converter on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	# Set Partitions	In # Bits	Out # Bits	Freq. (MHz)	Delay ns.	# Comb Fnc	# LUTs vs # inputs 7-	6-	5-	4-	3-	Est. # of Packed ALMs
4	15	4	8	406.3	2.461	5	0	0	0	5	-	3(0%)
5	52	6	15	406.3	2.461	22	0	3	12	4	3	12(0%)
6	203	8	18	250.8	3.988	161	3	10	83	34	31	87(0%)
7	877	10	21	113.2	8.836	882	5	54	538	183	102	469(0%)
8	4,140	13	24	100.5	9.954	4100	32	1416	1223	652	777	2628(1%)

Table 4 shows the FPGA resources and frequency achieved on the Altera Stratix IV EP4SE530F43C3NES FPGA by the multi-stage index to set partition circuit shown in Fig. 4. This uses fewer resources than the single-stage circuit in Fig. 3, but its latency is greater. In the design of the multi-stage circuit, registers were placed between each stage. As a result, the delay figures shown are reduced, approximating the delay of one stage. The first index comes out of this circuit $n - 1$ clock periods.

Table 4. Frequency/resources used to realize the multi-stage index to set partition converter on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	# Set Par- titions	In # Bits	Out # Bits	Freq. (MHz)	Delay ns.	# Comb Fnc	# LUTs vs # inputs 7-	6-	5-	4-	3-	Est. # of Packed ALMs
5	52	6	15	403.5	2.478	57	0	4	19	20	14	35(0%)
6	203	8	18	275.0	3.636	100	1	7	36	38	18	63(0%)
7	877	10	21	227.8	4.389	203	0	8	82	71	42	121(0%)
8	4,140	13	24	203.0	4.926	326	3	20	124	107	72	196(0%)
16	1.05×10^{10}	34	64	101.4	9.859	3842	35	718	1524	1130	435	2339(0%)
32	1.28×10^{26}	88	160	55.6	17.973	38305	87	3671	19768	8016	6763	21206(9%)

The data shown comes from Verilog code that was written to implement each of the three circuit types. Synplify Pro was used to design each circuit. Further, ModelSim was used to simulate each circuit. In the case of the multi-stage circuit, a MATLAB program was written to produce a header file that was called from the Verilog code to provide threshold values for the comparators.

4 Concluding Remarks

To the best of our knowledge, our circuits are the first hardware implementations of set partition generators. The generation of set partitions by hardware has important practical applications. The challenge is to generate set partitions at one per clock period. We show two ways to accomplish this. The first is a sequential circuit that generates the partitions in lexicographical order according to their restricted growth string. This circuit is fast and can produce partitions of large sets. The second circuit is an index to set partition converter. In this circuit, an up counter on the index input produces set partitions in increasing lexicographical order, while a down counter produces set partitions in decreasing lexicographical order. Also, a random number generator at the index produces random set partitions. It is combinational, but can be pipelined to produce a set partition at one per clock. An analysis of the complexity of these two circuits show that the complexity of both grow polynomially with n, the number of elements in the partition, while the delay grows linearly with n. Also, for both circuits, we show experimental results that confirm these predictions. Specifically, small to large circuits were implemented on the Altera Stratix IV EP4SE530F43C3NES FPGA. Our experimental results show that an FPGA running at 100 MHz produces partitions at a rate that is about 10 times the rate of a software implemented partition generator on a processor that runs at 2.26 GHz.

Acknowledgements. We thank six referees for comments that improved this paper.

References

1. Berend, D., Tassa, T.: Improved bounds on Bell numbers and on moments of sums of random variables. Probability and Mathematical Statistics 30(2), 185–205
2. Butler, J.T., Sasao, T.: Index to Constant Weight Codeword Converter. In: Koch, A., Krishnamurthy, R., McAllister, J., Woods, R., El-Ghazawi, T. (eds.) ARC 2011. LNCS, vol. 6578, pp. 193–205. Springer, Heidelberg (2011)
3. Butler, J.T., Sasao, T.: Hardware index to permutation converter. In: 19th Reconfigurable Architectures Workshop (RAW 2012), Proc. of the 26th IEEE International Parallel and Distributed Processing Symposium, Shanghai, China, May 21-22, pp. 424–429 (2012)
4. Chen, X., Liu, L., Liu, Z., Jiang, T.: On the minimum common integer partition problem. ACM Trans. on Computational Logic V, 1–19 (2008)
5. Debnath, D., Sasao, T.: Fast Boolean matching under permutation by efficient computation of canonical form. IEICE Trans. Fundamentals (12), 3134–3140 (2004)
6. Beeler, M., Gosper, R.W., Schroeppel, R.: HAKMEM. MIT Artificial Intelligence Laboratory, Cambridge, MA, Memo AIM-239 (February 1972), `http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html#item175`
7. Hankin, R.K.S., West, L.J.: Set partitions in *R*. J. of Statistical Software 23, Code Snippet 2 (December 2007), `http://www.jstatsoft.org/`
8. Knuth, D.E.: Volume 4 Generating all combinations and permutations. In: The Art of Computer Programming, Fascicle 3. Addison-Wesley ISBN: 0-321-58050-8
9. Kawano, S., Nakano, S.: Constant time generation of set partitions. IEICE Trans. Fundamentals E88-A(4), 930–934 (2005)
10. McKay, J.K.S.: Algorithm 263, Partition Generator. Communications of the ACM 8(8), 493 (1965)
11. Nagayama, S., Sasao, T., Butler, J.T.: Analysis of multi-state systems with multi-state components using EVBDDs. In: Proc. 42nd International Symposium on Multiple-Valued Logic, Victoria, Canada, May 14-16, pp. 122–127 (2012)
12. Oommen, B.J., Ng, D.T.H.: On generating random partitions with arbitrary distributions. The Computer Journal 33(4), 368–374 (1990)
13. Orlov, M.: Efficient generation of set partitions (March 2002), `http://www.cs.bgu.ac.il/~orlovm/papers/partitions.pdf`
14. Reingold, E., Nivergelt, J., Deo, N.: Combinatorial Algorithms, Theory and Practice. Prentice-Hall (1977)
15. Sasao, T.: Memory Based Logic Synthesis, 1st edn. Springer (2011) ISBN: 978-1-4419-8103-5
16. Semba, I.: An efficient algorithm for generating all partitions of the set $\{1,2,..., n\}$. Journal of Information Processing 7(1) (1984)
17. Stojmenovič, I.: An optimal algorithm for generating equivalence relations on a linear array of processors. BIT 30(3), 424–436 (1990)

Teaching SoC Using Video Games to Improve Student Engagement

Christopher J. Martinez

University of New Haven, West Haven CT 06516, USA

Abstract. This paper introduces a project-based course for the emerging field of system-on-chip (SoC). SoC is allowing for a new perspective on embedded system education. Previous undergraduate embedded system courses have always based around the use of low-end 8-bit/16-bit microcontrollers (e.g. PIC, Freescale, AVR). The low-end microcontrollers are good at teaching input/output interface but do not fully explore the connection of designing the hardware and software interface. SoC allows for a student to design an embedded system that bridges the two areas of computer science and engineering. This paper will describe how a new course in SoC uses an engaging assignment of creating a video game system. A student will appreciate the hardware and software side of todays embedded computer systems after completing this course since the course will require the design of hardware and the design of a software system. The paper will show the layout of the current SoC course and offer suggestions on how the course can be modified to meet the academic rigor for different programs in computer engineering.

1 Introduction

Computer engineering programs strive to develop a curriculum that will allow their students to have a deep understanding of hardware and software. There are hardware focused courses such as circuits, digital logic, etc. that only expose the students to hardware and do not mention software. Similarly the computer science courses give a strong foundation in software but will not make a connection to hardware. The one area where students can see the connection between hardware and software is embedded systems.

Embedded systems is the hands-on course in a curriculum that can provide insight into the hardware and software interface. Embedded systems are normally taught as a sequence of courses in most computer engineering programs around the country. It is common that the embedded systems are two parts: one part that introduces the student to embedded systems with assembly language and a second part that has the student create designs using hardware and programming. The embedded systems courses have always been based around the use of low-end 8-bit or 16-bit microcontrollers, commonly used microcontrollers from companies such as Microchip, Freescale and Atmel. Embedded system education has begun to change in recent years with the emergence of system-on-chip (SoC) technology.

In the past, the University of New Haven had a traditional embedded system course sequence but has now been redesigned with SoC technology being included as a part of embedded systems. The first course is now a combination of the old two-sequence course that now looks at assembly language and industry standard buses. The SoC course aims to have a course that explores the hardware and software interface in depth. The SoC course is project-based to encourage student engagement. The course has the student design a 2D video game system. The project is a semester long project with hardware and software components.

2 Course Objectives

The course was set out to meet the following educational objectives: 1) Have an understanding of the EDA tools used in an FPGA SoC, 2) Have an understanding of IP cores used in SoC, 3) Have a more in-depth understanding of the hardware and software interface, 4) Be able to design custom IP cores using VHDL, 5) Be able to design a complex software application that requires hardware interaction, 6) Have an understanding on the importance of threads in an embedded system, and 7) Be able to troubleshoot a system with both hardware and software components.

The first offering of the course occurred in the Fall of 2010. The course was run with lectures and a number of independent projects that connected with topics discussed in class. The students would write software using prebuilt IP cores and not fully understand the connection being made at the hardware level. For the second offering of the course, the instructor changed the structure of the course to focus on a semester long project that was larger in scope and complexity. The previous offering in the Fall of 2010 had 6 projects spread across a 14 week term and the new offering had 8 projects in the same 14 week term. There were many examples of SoC courses that used real-world examples [1], [2], [3]. The courses discussed in [1], [2], [3] were successful and very engaging for the students. All the courses used robotics to explore the topics of SoC, the instructor decide to go in a different direction and focus on a video game. Video games have been explored be used in courses and instructors had great success in engaging the students [5], [6].

3 Course Project

The course is designed around a 2D video game project. The course has a scheduled three-hour lecture and uses the project assignments for assessment of the students progress. The project was done in pairs and used practical quizzes to assess each student individually. The course is a heavy workload with 8 projects so the working in pairs is essential in order to make the course manageable for the students.

The project requires a mixture of hardware and software components. The project requires the student to design the hardware for the joystick controller, VGA interface, and sprite logic in VHDL. The software requires the writing of a very simplistic 2D game engine. Figure 1 shows a block diagram of the 2D video game system.

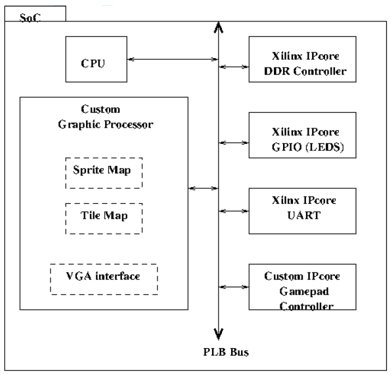


Fig. 1. SoC block diagram of 2D game system

Laboratory One: Introduction to EDA Tools – The first laboratory assignment is used to introduce the students to the Xilinx design tools. The students create a basic SoC that includes a number of Xilinx IP cores (GPIO IP, Timer IP, UART IP) using the XPS tool. The SDK is used for the students to develop a simple program that shows how to interface with IP cores and download the programs to the development board. The students write the same program using the Timer in two different ways, polling and interrupt in order to make LEDs blink on and off and have messages written to the UART interface.

Laboratory Two: Hardware for the NES Controller – This laboratory assignment is meant to be a refresher for the students in VHDL. The joystick to control the video game is from the 1st generation Nintendo (NES). The controller has 8 digital switches for Up, Down, Left, Right, Select, Start, A button and B button. The controller sends the information using a serial transmission. The goal of assignment is for the students to read the NES controller by writing the VHDL for serial communication.

Laboratory Three: Hardware for the Graphics Processor Part 1 – The laboratory assignment will focus on creating parts of the 2D graphic processor. The students learn about the VGA interface and how to design the digital circuit. The students implement a 640x480 resolution screen and must define the timing to generate the hsync and vsync signals. Creating the VGA circuit is only part of the assignment. Students also learn about memory limits in the FPGA chip and that it will not be possible to define each of the 307,200 pixels in memory. To reduce the memory footprint the video game system will be sprite-based. Sprite-based games represent video games using sprite-maps and tile-maps. The sprite-map is used to store all the elements that will appear on screen. The tile-map is a 2D array that stores an index to sprite to display on the screen. The sprites are defined to be 8 pixels by 8 pixels and a tile can hold one sprite.

A tile is made up of 64 pixels using a screen resolution of 320x240 a total of 1,200 tiles will be needed.

Laboratory Four: Hardware for the Graphics Processor Part 2 – The fourth laboratory assignment is for students to express their creative side by creating the graphics for their game and place in the sprite-map. The sprite-map is stored in the BRAM memory on the FPGA. The students will create enough BRAM memory space for 32 8x8 pixel sprites. The assignment also creates the background (walls, floor) and foreground (player and enemies) for the game. This requires two separate tile-maps, one for the background and one for the foreground. The blending of the two tile-maps is done by the introduction of an alpha bit for color. If the alpha bit on a foreground sprite is set the foreground is shown otherwise the foreground becomes transparent.

Laboratory Five: Generating an SoC IP Core – The hardware the students have created for the joystick and graphics are now combined into the SoC. The Xliinx XPS tool is used to create a custom IP Core that connects to the processor bus.

Laboratory Six: Software for the 2D Game Engine Building the Playing Field – The focus of the class shifts to developing the embedded software for the SoC. Building the video game requires the creation of a 2D game engine. The game engine controls what should be placed on the screen, controlling the enemies & player, game physics, etc. The first part of the game engine is to display the background for the game. The students create a low-level device driver to interface with the video IP core. The low-level driver allows the programmer to select if where data is written to the foreground tile-map, background tile-map, or sprite-map.

The game is made up of a number of screens that scroll as the player moves through the game. The games that will be produced are a platform side-scroller, e.g. Super Mario Brothers. The side-scroller is made up of worlds and levels. In this assignment the students will create a data structure that can implement the worlds, levels, and screens. The drawing of a screen is the basic element that must be developed first. A background screen is made up of elements that repeat a sprite multiple times. For example, when drawing the floor a programmer will want to draw a floor sprite 10 times in a horizontal direction. A data structure is developed to explain how to draw the screen. The screen is represented by an x and y coordinate that corresponds to a tile on the tile-map. The deliverable for this project is all the data structures and a demonstration showing the movement between worlds, levels, and screens.

Laboratory Seven: Software for the 2D Game Engine Building the Player – The next item that will be added to the 2D game engine is the moveable player. The side scrolling game will require that a player can traverse the screen in the horizontal and vertical direction. The horizontal direction will only require the player to run in both the left and right direction. The game engine will need to monitor the background and have the player interact with the environment. The player must stop moving when it encounters a wall, determines it has fallen into a pit that results in death, or encounters a moveable object in the environment.

The vertical direction requires the player to be able to jump and climb ladders. The addition of jumping requires a basic modeling of physics. The game engine requires a data structure to monitor the player.

In addition this project will require the students to develop a producer and consumer thread that is time multiplex with the frame rate of the video display. The producer is the game engine that determines the interaction on the screen. The producer manipulates the tile-maps. The consumer is executed on a timer interrupts that transfer data from the C program to the video buffer on the graphic hardware.

Laboratory Eight: Software for the 2D Game Engine Building the Enemies – The last laboratory assignment is adding enemies to the 2D game engine. The students will modify the data structure created for the player and adapt it to fit the needs of an enemy. The assignment required the creation of three types of enemies: ground enemy, flying enemy, and final boss. The game engine was also modified to allow for both the player and enemy to fire bullets at each other. Figure 2 shows a screen shot of the final game.

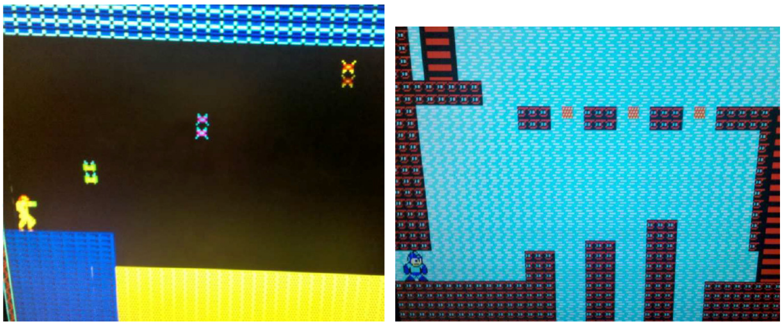


Fig. 2. Screenshot of final game from laboratory 8

4 Student Improvement

At the end of the course the instructor asks the students for their feedback using the common end of course student survey. The students rated their achievement of the seven course outcomes very highly with an average of 4.25 out of 5.0. The average improved by 0.14 from the student survey the previous year. The instructor was able to see that the video game did improve the student engagement in the course. In the Fall 2010 offering of the course (did not use the video game) the number of late submissions by the students was 45%. In contrast the introduction of the video game project in the Fall 2011 offering the number of late assignments were 15%, all late assignments came for a single team pair out of seven total team pairs. The students final grades had a large improvement due to the reduction in late submissions. The final grades improved by 6.86 points (half a letter grade).

5 Potential Modifications

The eight laboratory assignments described in this paper are guidelines as to how to create a 2D video game system. Modifications that could be included as other potential laboratory assignments include: 1) adding sound by creating an IP-core to access a DAC (music can be done using the karplus-strong algorithm is hardware or software), 2) a real-time embedded operating system can be used, and 3) a more advance graphic engine could be created (sprite transformation, eg. rotation, color change, etc., done in hardware).

6 Conclusion

An example of using a 2D video game system for a project based SoC senior level course has been presented. The eight laboratory assignments help to reinforce both hardware and software aspects of computer engineering that students have gained knowledge of in previous classes. The hardware and software required to develop is of moderate difficultly and the integration of the two is used to explain SoC technology and the hardware/software interface. Student comments show that the course was a positive experience and students were fully engaged in the course. The material covered in this paper is a foundation to a SoC course and will be improved in the years to come.

References

1. Barros, A., Lima, N., Xavier, J., Lima, M.E.: Teaching SoC design in a project-oriented course based on robotics. In: 2005 IEEE International Conference on Microelectronic Systems Education, pp. 25–26 (2005)
2. Bindal, A., Mann, S., Ahmed, B.N., Raimundo, L.A.: An undergraduate system-on-chip (SoC) course for computer engineering students. IEEE Transactions on Education, 279–289 (2005)
3. Sorensen, A.S., Falsig, S.: A System on Chip approach to enhanced learning in interdisciplinary robotics. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 4050–4056 (2010)
4. Kim, J.: An Ill-Structured PBL-Based Microprocessor Course Without Formal Laboratory. IEEE Transactions on Education, 145–153 (2012)
5. Sung, K., Hillyard, C., Angotti, R.L., Panitz, M.W., Goldstein, D.S., Norlinger, J.: Game-Themed Programming Assignment Modules: A pathway for Gradual Integration of Gaming Context into Existing Introductory Programming Courses. IEEE Transactions on Education, 416–427 (2011)
6. Butler-Purry, K., Srinivasan, V., Pedersen, S.: Video game for enhancing learning in Digital Systems Courses. In: Proceedings of 2009 American Society of Engineering Education Conference (June 2009)

Parameterized Design and Evaluation of Bandwidth Compressor for Floating-Point Data Streams in FPGA-Based Custom Computing

Tomohiro Ueno, Yoshiaki Kono, Kentaro Sano, and Satoru Yamamoto

Graduate School of Information Sciences, Tohoku University
6-6-01 Aramaki Aza Aoba, Aoba-ku, Sendai 980-8579, Japan
{ueno,kono,kentah,yamamoto}@caero.mech.tohoku.ac.jp

Abstract. We are applying bandwidth compression to enhance performance of FPGA-based custom computing. This paper presents and evaluates hardware design of a bandwidth compressor and decompressor for a floating-point data stream of various bit width. We show their structures parameterized for a bit width of an input word. Through FPGA-based prototype implementation, we evaluate the resource utilization, frequency, and compression ratio. The expermental results show that the compressor and decompressor for 32-bit and 64-bit floating-point numbers achieve bandwidth reduction at a ratio of 3.1 and 1.8 for 2D data of fluid dynamics computation, while they require only small area and operate at higher than 200MHz.

Keywords: bandwidth compression, floating-point data stream, custom computing, parameterized design.

1 Introduction

Stream computation is one of the useful and promising models to accelerate scientific computations with a custom computing machine. It can exploit a memory bandwidth with a successive and regular access pattern, achieving high-performance computation efficiently. Moreover, pipelining in stream computation allows higher performance to be obtained by using more hardware stages with a limited memory-bandwidth [9, 13]. However, memory bandwidth is still important to fundamentally increase achievable performance of stream computation. Since it is not easy to drastically increase the bandwidth of off-chip I/O pins, present and future semiconductor chips are perpetually suffering from the insufficiency of I/O bandwidth, while on-chip computing resources can be increased easily by technology scaling.

To handle the problem, we have proposed hardware for bandwidth compression of numerical data in stream computation [8, 14, 20]. The hardware performs high-throughput lossless compression of a floating-point data stream. With bandwidth compression, we can enhance an available memory bandwidth for computation by compressing data on the fly. The physical bandwidth of memory can efficiently be utilized with compressed data streams where redundancy of

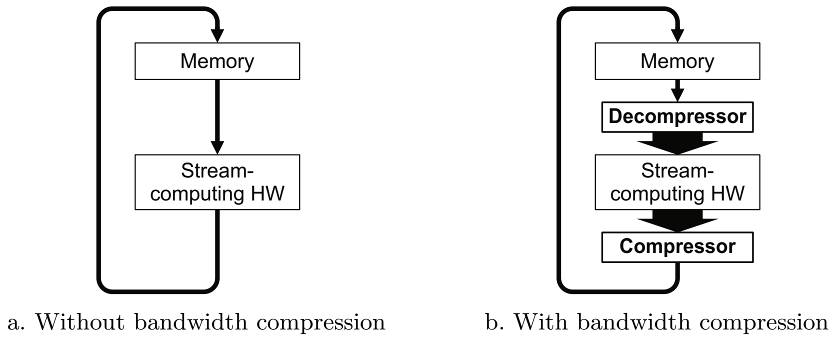


Fig. 1. Stream computation with/without bandwidth compression

bits is reduced. The lossless feature guarantees the complete reconstruction of original data, so that stream computation is not collapsed by bandwidth compression. Fig. 1 shows the stream computation with and without bandwidth compression. If the compressor reduces the data size to $1/r$ of the original size on average, we can substantially use the physical stream bandwidth as r times wider bandwidth. Since stream computation is inherently tolerant to the number of its pipeline stages, the delay cycles increased by the decompressor and compressor do not affect the overall performance of the stream computation as long as a throughput is maintained. Therefore, we require only high-throughput processing for compression and decompression.

So far, we have presented algorithms for lossless compression of a floating-point data stream, and designed high-throughput compressor and decompressor only for 32-bit single-precision floating-point numbers [8,20]. They demonstrated that the prediction-based algorithm achieves the compression ratio of about 3.5 for results of scientific computation, such as computatonal fluid dynamics (CFD). This means that the stream bandwidth can be enhanced to 3.5 times wider for CFD computation. Moreover, they also demonstrated that the compressor and decompressor are very small. Since the compressor and the decompressor are auxiliary hardware for computation itself, they should be implemented in small area, as long as high-throughput compression is available.

However, we also use different bit widths other than the 32-bit single-precision in actual computations. Most of scientific computations are based on operations of the 64-bit double-precision, and some of them may require higher precision such as the 128-bit quad-precision or more. Difference in the bit width can change the hardware area and the compression ratio. Moreover, each of the components in the compressor and decompressor has different complexity to the bit width, and therefore a critical-path can be found in a different stage depending on the bit width, while the hardware is designed to easily be pipelined. For practical usage of bandwidth compression, we should know trade-offs among a bit width, area, frequency and a compression ratio.

In this paper, we present and evaluate a parametrized design of the compressor and decompressor for various bit widths. We design components as parametrized hardware modules, and then put them together to build the compressor and decompressor. With an FPGA-based prototype system, we evaluate resource utilization, operating frequency, and a compression ratio of the compressor and decompressor for a different bit width.

This paper is organized as follows. Section 2 gives related work. Section 3 briefly describes the prediction-based algorithm for floating-point lossless data-compression, and the design of the compressor and decompressor. And then we show their parametrized structure for a given bit width. In Section 4, we verify and evaluate the compressor and decompressor implemented with a prototype system on FPGA. Finally, Section 5 gives conclusions and future work.

2 Related Work

Since bandwidth compression requires high-throughput, hardware implementation is necessary. So far, hardware-based bandwidth compression techniques have been researched for audio [10], video [5], and arbitrary data in a main memory [19] or in streaming transfer [4,15]. However, these techniques cannot directly compress floating-point data at a high compression ratio because they don't well exploit characteristics of floating-point numerical data.

Several lossless compression algorithms have been proposed to directly compress floating-point data. Lindstorm et al. [11] proposed a compression algorithm that combines prediction and entropy coding. They achieved high compression ratios for 2D and 3D data sets by using 2D or 3D prediction functions. Ratanaworabhan et al. [12] proposed compression for double-precision floating-point numbers using a hash table for context-based prediction. Their algorithm also achieved high compression ratio when using a sufficiently large table. These techniques achieve a better ratio than that of the other algorithms, however, they are designed for software implementation that provides insufficient throughput for bandwidth compression.

Tomari et al. [18] proposed an algorithm for hardware-based high-throughput decompression of double-precision floating-point data stream. They perform compression by software. Their FPGA-based implementation achieves high throughput decompression with a simple structure to look up a history table. However, the throughput of the software-based compression is limited. Their target is data transfer between a host node in a PC cluster and its accelerator, while we aim at bandwidth compression of data stream in a custom computing machine. In addition, their compression ratio is not high, which is at most 1.21, because only the sign and exponent bits are encoded while the significand bits are output as they are. Furthermore, they did not design and evaluate hardware for other precisions such as the single precision.

We design hardware to directly compress and decompress a floating-point data stream in stream computation. The hardware compressor and decompressor achieve a better compression ratio than that by general-purpose compressors such as bzip2. The design is parameterized for floating-point data with various precisions including 32-bit single, 64-bit double, 128-bit quad, and 256-bit octa-precision. We evaluate compression ratio, area, and frequency of FPGA-based prototype implementation for the parametrized design.

3 Algorithm and Design of Bandwidth Compressor

3.1 Requirements

To select an algorithm and design hardware, we considered the following requirements of bandwidth compression applied to stream computation with floating-point operations: 1) lossless compression, 2) as high compression ratio as possible, 3) single-pass compression, and 4) high-throughput, but small hardware cost. First, we have to guarantee the same data are reconstructed from compressed data because additional errors should not be introduced in scientific computation. Therefore, compression must be lossless. Second, compression ratio should be as high as possible to improve memory bandwidth. Third, compression has to be performed locally because entire data are not available in advance of stream computation. Single pass means that entire data are traversed only once.

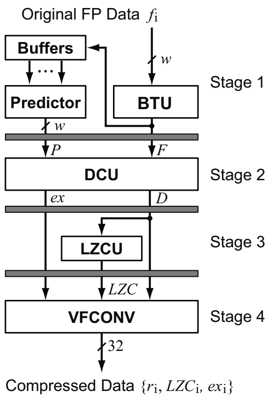
So far, several prediction-based lossless algorithms have been proposed [2,6,7, 11,12], to directly compress floating-point data. The algorithms achieve higher compression ratios than general-purpose compressors such as bzip2 [1]. They are single-pass algorithms because each datum is locally compressed based on data prediction. In these algorithms, a value of the next input is predicted by using previous input values, and it is encoded with a difference between the prediction and the actual value. Since they satisfy the requirements 1 to 3, we focus on the prediction-based compression algorithms.

The prediction-based algorithms are classified into two groups for the way of prediction: arithmetic-based one [6,7,11] and context-based one [2,12]. The arithmetic-based algorithm uses an arithmetic predictor to obtain the prediction by computation. The context-based algorithm uses a hash table to look up previous input phrases that is the same as the present phrase to predict the next input. For the fourth requirement, we select the arithmetic-based algorithm because it allows hardware to be faster and smaller than that for the context-based algorithm. The hash table requires a large on-chip memory and update of the memory for every prediction limits an operating frequency. Especially, we proposed to use 1D polynomial for arithmetic prediction [8,14] to reduce a buffer memory that is used to store the previous input data.

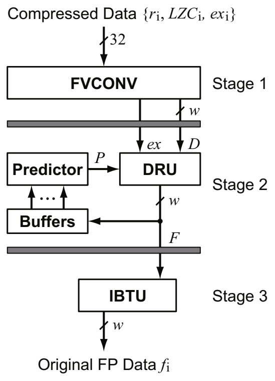
3.2 Prediction-Based Compression Algorithm

We compress a data stream $S = \{..., f_{i-1}, f_i, ...\}$ of IEEE754 floating-point numbers. Such a data stream is made by traversing a 2D or 3D grid. For f_i, the compressor computes its prediction p_i with some of the previous inputs stored in a buffer memory. When prediction is made with good accuracy, difference between f_i and p_i has many zeros from MSB because p_i has a closer bit pattern to that of f_i. By encoding these zeros with their length, we represent the difference with fewer bits than f_i.

Predictors. The computational results of scientific simulation such as CFD typically have some spatial continuity because they are solutions of the partial



a. compressor



b. decompressor

Fig. 2. Overview of compressor and decompressor

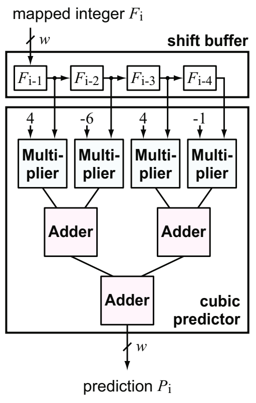


Fig. 3. Cubic predictor

differential equations governing the physical phenomena. For prediction, we approximate them locally with a polynomial function, which is called *a polynomial predictor*. This is similar to the idea given in [3].

Suppose that a numerical sequence $S = \{..., f_{i-1}, f_i\}$ is a set of regularly sampled values of a 1D function $f(x)$, so that $f_i = f(x_i) = f(i\Delta x)$ for integer i. Assuming that $f(x)$ is locally approximated by the $(n-1)$-th order polynomial function $f(x) = a_0 + a_1 x + a_2 + x^2 + ... + a_{n-1}x^{n-1}$, we can predict the next value $f_i = f(i\Delta x)$ by determining all a_k and computing $f(x)$ for $x = i\Delta x$. We can determine the n parameters, a_k, by evaluating the Lagrange polynomial [3] with the n previous values, $\{f_{i-n}, ..., f_{i-1}\}$. Considering $\Delta x = 1$ for simplicity, we obtain polynomial predictions, p_i, as follows:

$$p_i = 4f_{i-1} - 6f_{i-2} + 4f_{i-3} - f_{i-4} \quad \text{for } n = 4. \tag{1}$$

We refer to this predictor as a *1D cubic polynomial-predictor* or simply a *Cubic*.

Difference Encoders. We encode the difference between p_i and f_i with number of successive zeros from MSB, called *leading-zero count (LZC)*, and the remaining bits. We refer to the remaining bits as *a residual*, and denoted it with r. To obtain the difference, we adopt integer subtraction [11] instead of XOR difference [2,12] because subtraction generally gives better compression ratios [14]. For integer subtraction, we transform a floating-point number f to an unsigned integer F by flipping the sign bit of f for a positive number or all the bits for a negative number with magnitude scale maintained after the transformation [11]. As a result, positive and negative floating-point numbers are mapped continuously to the higher and lower space of unsigned integers, respectively, allowing closer numbers to always give smaller difference in bits. After p_i and f_i are transformed to their mapped integers, P and F, we compute $D = P - F$ for $P > F$, or $D = F - P$ for $P \leq F$. We also output whether $P > F$ or not, with an exchange bit, ex, where ex = 1 for $P > F$ and 0 for $P \leq F$.

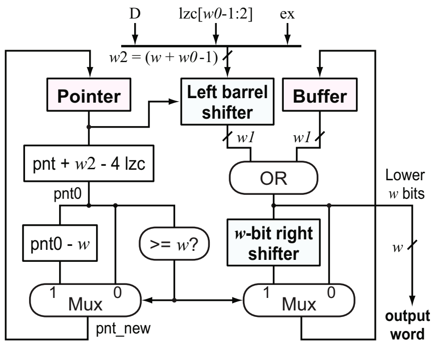


Fig. 4. Variable-to-fixed convetver

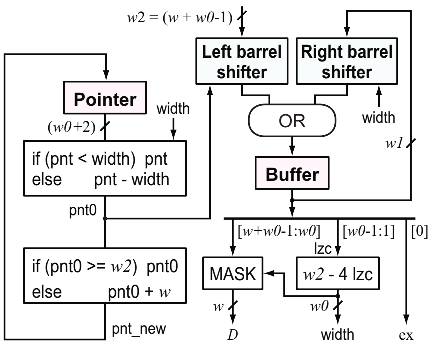


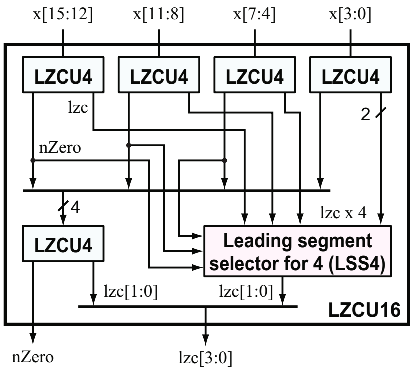
Fig. 5. Fixed-to-variable convetver

Then we obtain LZC of D. The raw LZC can be recorded directly, however we adopt *4-bit coding* [2,12,14] for easier handling of a variable length of the residual. Here suppose that a floating-point number has 32 bits for single precision. In this case, the raw LZC can be 0 to 31, which is represented in 5 bits. The residual has (32 − LZC) bits. On the other hand, the 4-bit coding represents LZC with a multiple of 4. For example, LZC=15 naively gives 17 residual-bits. In the 4-bit coding, the LZC is truncated to 12, and the residual becomes 20 bits being padded with 0s. The 4-bit coding is expected to have the advantage of 4-bit alignment that allows us to more simply output residuals with variable bit-length. Finally, we output a compressed datum that consists of the exchange bit ex, 3-bit $\lfloor \mathrm{LZC}/4 \rfloor$, and the variable-length residual r.

3.3 Overview of Hardware Compressor and Decompressor

Based on the compression algorithm mentioned above, we designed hardware compressor and decompressor for a single stream of single-precision floating-point numbers [8]. Figs.2a and b show the overviews of the compressor and the decompressor, which are pipelined with four and three stages. Datum from a floating-point data stream, $\{f_i\}$, is input to the compressor one by one every cycle, and then the compressor outputs a compressed bit-stream $\{(\mathrm{ex}, \mathrm{LZC}, r)_i\}$. The compressor is composed of *a binary transform unit (BTU)*, *a predictor with a buffer*, *a difference-computing unit (DCU)*, *an LZC unit (LZCU)* and *a variable-to-fixed length converter (VFCONV)*.

The input floating-point datum f is firstly transformed to an unsigned integer F by BTU as discribed in section 3.2. The obtained integers are stored in the buffer while it is also sent to DCU in the next pipeline stage. The buffer stores a necessary number of previous inputs. Then the predictor computes Eq.(1) to give prediction P for the current input F. Fig.3 shows the cubic predictor and the shift buffer, which requires the four inputs from the shift buffer in parallel. The integer transformation allows us to use integer operators for prediction instead of floating-point operators with high hardware costs and longer delays. Our preliminary experiments show that the prediction in mapped integers provides almost



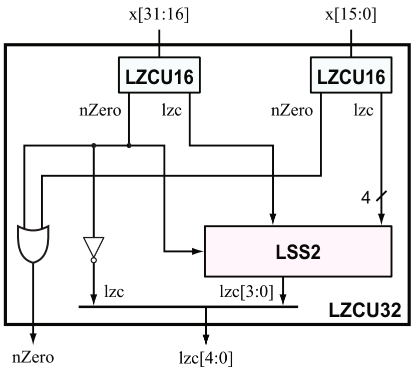


Fig. 6. Leading-zero count unit for 16 bits **Fig. 7.** Leading-zero count unit for 32 bits

the same compression performance as the floating-point prediction, especially for floating-point numbers whose exponents rarely change [8].

DCU computes the difference, D, between P and F by subtracting the smaller from the larger after swapping P and F if necessary. DCU also outputs the exchange signal, ex, which is asserted when P and F are swapped. LZCU computes (LZC) of D. Finally VFCONV outputs words of compressed data, which are of $\{r, \mathrm{LZC}, \mathrm{ex}\}$.

The decompressor consists of *a fixed-to-variable length converter (FVCONV)*, *the predictor with the buffer*, *a data-reconstruction unit (DRU)* and *an inverse binary transform unit (IBTU)*. The predictor and the buffer are the same as those of the compressor. The decompressor outputs the original floating-point data stream processing the compressed data stream. FVCONV generates D with ex, LZC and r. The predictor gives prediction P with the previously decompressed numbers stored in the buffer. DRU reconstructs mapped integers of the original data, F, with D, P and ex by inversed operation of the difference-computation. Finally IBTU transforms F to its original floating-point number f.

3.4 Parametrized Design

For the compressor and decompressor to process various bit width w, we parameterize their design. Here we describe structures of the major components: the predictor, VFCONV, FVCONV, and the LZCU. The other components are simple, and easy to be parameterized.

As shown in Fig. 3, the width of the buffer and operators in the predictor is parameterized with w. Fig. 4 shows a structure of the VFCONV. The VFCONV converts variable-length inputs to w-bit fixed-length outputs by using the pointer and the buffer. The pointer manages the MSB position of the data in the buffer. According to the pointer, the left barrel shifter adjusts the bit position where the input is inserted in the buffer. The VFCONV is composed of the two data-paths to update the pointer and the buffer, respectively. Here we define three parameters, w_0, w_1, and w_2, which are the length of the raw LZC, width of the buffer, and the

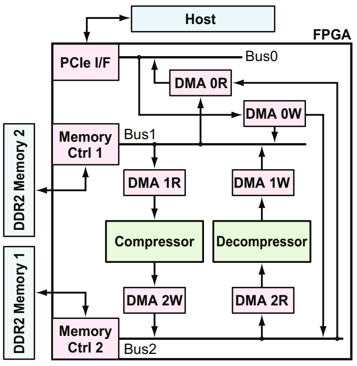


Fig. 8. Prototype system

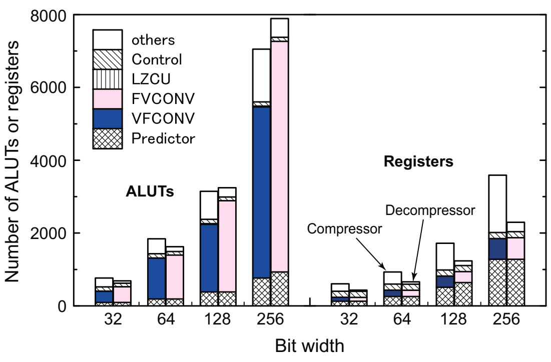


Fig. 9. Resource utilization

length of concatenated data of D, and LZC/4, and ex. They are obtained by the following equations: $w_0 = \log 2w$, $w_1 = 2w + w_0 - 1$, and $w_2 = w + w_0 - 1$.

Fig. 5 shows the FVCONV in the decompressor, which also has the pointer and buffer for conversion. The pointer is updated according to the update status of the buffer. The data in the buffer is shifted by the right barrel shifter when a datum is output from the buffer. The FVCONV is also parameterized with the parameters, w, w_0, w_1, and w_2.

Figs. 6 and 7 show the LZCUs for 16-bit and 32-bit inputs, respectively. We designed LZCUs for various bit width by using a 4-bit LZC unit, called LZCU4. The LZCU4 has a 4-bit input and outputs of 2-bit lzc and 1-bit non-zero (nZero) signal. We built an LZCU for a 16-bit input, called LZCU16, with five LZCU4s as shown in Fig. 6. In the LZCU16, we also have the leading-segment selector for 4 (LSS4) to select one from four lzc from the four LZCU4s. The 4-bit lzc of the LZCU16 is generated by concatenating the 2-bit lzc from the second-stage LZCU4 and the LSS4. Thus we can compose a four times wider LZCU by combining four LZCUs. Note that we can also compose two times wider LZCU by combining two LZCUs with LSS2 as shown in Fig. 7.

4 Implementation and Results

4.1 Implementation

To evaluate area, frequency, and compression ratio of the compressor and decompressor for various bit width, we implemented them in the FPGA-based prototype system as Fig. 8 shows. We used a Terasic DE4 board [16], which has ALTERA Stratix IV EP4SGX230 FPGA, two DDR2 memories, and PCI-Express Gen 2.0. The FPGA is from the high-end 40-nm FPGA series, which has 182400 ALUTs (adaptive lookup tables). The system is designed by using the ALTERA Qsys development tool. We implemented four systems with the compressor and decompressor for 32-bit, 64-bit, 128-bit, and 256-bit floating-point data streams, respectively. In addition to the compressor and decompressor shown in Fig. 2, we also implemented controllers for them to have Avalon-ST interface, which is

Table 1. Latency of pipeline stages in compressor and decompressor

Module	Bit width	Latency [nsec]			
		Stage 1	Stage 2	Stage 3	Stage 4
Compressor	32	3.286	3.377	3.296	**3.735**
	64	3.527	3.857	3.385	**4.077**
	128	**5.042**	4.223	4.097	4.684
	256	**7.041**	5.84	5.886	6.089
Decompressor	32	3.232	**4.103**	3.3	-
	64	3.361	**4.641**	3.926	-
	128	5.042	**5.044**	4.084	-
	256	**7.262**	7.036	5.898	-

ALTERA's stream bus standard in Qsys. We wrote a parametrized Verilog-HDL codes for the compressor and decompressor, and compiled them using Quartus II compiler ver.11.1 with "speed" option. The compressor and decompressor operating at 125MHz in the system are connected to the DDR2 memories via the DMA controllers, so that we can stream data into them from the memories.

4.2 Area

Fig. 9 shows resource utilization for the compressor and decompressor of different bit width. The "others" means the pipeline registers and some glue logics to connect the modules. For both of the compressor and decompressor, the number of ALUTs increases almost linearly. In the case of 32-bit width, the compressor and decompressor consume 764 and 689 ALUTs, respectively, which corresponds to only 0.42% and 0.38% of the total ALUTs on the FPGA. This means that they can be implemented in very small area. In the case of 64-bit width, the compressor and decompressor are still small, consuming only 1.01% and 0.89% of the total ALUTs, while the 256-bit designs require 3.87% and 4.33% ALUTs for the compressor and decompressor, respectively. The FVCONV and VFCONV occupy more than half area in the compressor and decompressor, respectively, because of their barrel shifters. The predictor is smaller than the FVCONV and VFCONV. This is because the constant multipliers are actually implemented with bit shift.

The number of registers also increases as bit width increases, however, their consumption is not so high. The predictor consumes more registers than those of the VFCONV and FVCONV for its shift buffer. The registers for the others of the compressor much more than those of the decompressor because the pipeline registers require more bits in the compressor. No DSP block and no block memory are utilized in the compressor and decompressor, so that these units can be used by major hardware modules for computation.

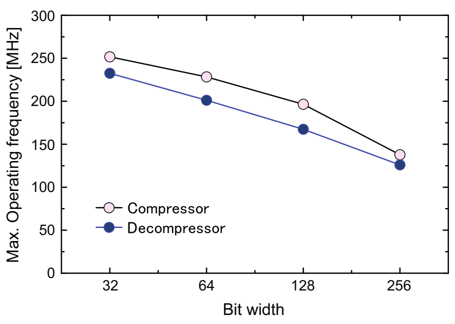


Fig. 10. Maximum operating frequency

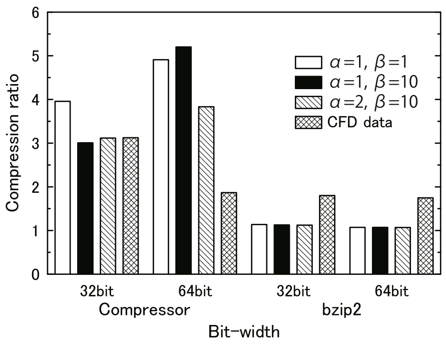


Fig. 11. Compression ratio

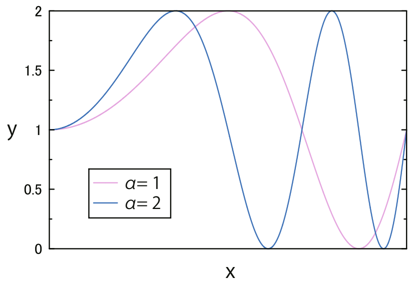


Fig. 12. Test data generated by sampling a function for $\beta = 1$

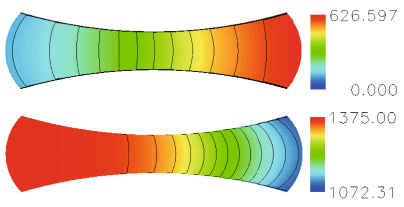


Fig. 13. CFD result of a 2D Laval nozzle. The color bar shows velocities and pressures

4.3 Frequency

Fig. 10 shows maximum operating frequency for different bit width. The compressor and decompressor can operate at higher than 232MHz and 201MHz for both 32-bit and 64-bit, respectively, while the frequency decreases as bit width increases. However, frequency higher than 120MHz is still available for both 256-bit compression and decompression. Table 1 shows latencies of pipeline stages. The compressor has a critical path in the stage 4 including VFCONV for 32-bit and 64-bit, while the stage 1 including the predictor becomes a critical path for 128-bit and 256-bit. This is because the latency of the adders in the predictor gets longer than that of the barrel shifter in VFCONV for wider bits.

On the other hand, the decompressor has a critical path in the stage 2 with the predictor for shorter bits, while the stage 1 including FVCONV gets to have a critical path for 256-bit. This is because the stage 2 has a long critical-path for the DRU, buffers, and the predictor. The path from the pointer to the buffer in the VFCONV gets long for large bit width due to the left barrel shifter and w-bit right shifter in the path.

4.4 Compression Ratio

We evaluated *a compression ratio*, which is defined with $\frac{\text{(Size of original data)}}{\text{(Size of compressed data)}}$, for different bit width. For evaluation, we used the test data and the CFD data shown in Figs. 12 and 13, respectively. The test data are obtained by sampling a function, $f_i = \sin\left(\frac{2\pi\alpha x^2}{32768^2}\right) + \beta$, at 32768 points.We used GNU MPFR (multiple-precision floating-point computations with correct rounding) library [17] to compute it for 32-bit and 64-bit floating-point numbers. The CFD data are obtained by computing fluids in the 2D Laval nozzle. The data are of a 101×301 orthogonal grid in generalized curvilinear coordinates, which contains velocity, pressure, temperature and specific heat value at constant pressure.

Fig. 11 shows the compression ratio. The compression ratios of the test data tend to be higher for smaller α and β, because such parameters make smaller prediction errors between adjacent samples in compressing the test data. As a result, the 32-bit and 64-bit compressions achieve the maximum compression ratio of 3.9 and 5.2, respectively. This means that they can reduce bandwidth at a degree of 3.9 and 5.2, respectively, for 32-bit and 64-bit compressions. On the other hand, the CFD data gave lower compression ratios than those of the test data, which are 3.1 and 1.8 for 32-bit and 64-bit compressions. This is due to the less continuity in the CFD data than the test data, however, the results show that bandwidth compression is also available for such computational data. In all cases, our compressor achieves equal or better ratios than those by the bzip2, a general-purpose software compressor.

Considering hardware utilization, compression performance per area is high for narrower bit width such as 32-bit and 64-bit. When we compress the computational data of Fig. 13, we can reduce bandwidth at a degree of 3.1 and 1.8 by consuming only 0.4% to 1.0% ALUTs. Since these bit widths are commonly used in scientific computation, our hardware-based compressor is especially useful for bandwidth compression in typical computation with single-precision or double-precision floating-point operations.

5 Conclusions

This paper presents the hardware designs and the evaluation of the bandwidth compressor and decompressor for a floating-point data stream in various bit width. We show the structure of these modules parametrized for a bit width of input word. By implementing the compressor and decompressor with FPGA, we evaluated the resource utilization, frequency, and compression ratio. The compressor allows to compress the 32-bit and 64-bit CFD data at a ratio of 3.1 and 1.8, respectively, operating at higher than 200MHz and consuming only small logic resources. These results mean that the bandwidth compressor for single and double-precision floating-point numbers is very useful for bandwidth compression at a low hardware cost. The hardware for wider bit width requires much more resources, and therefore it should be used when bandwidth compression is significantly required.

In our future work, we will evaluate the hardware for other numerical data types inclusing integer and fixed-point numbers. we will also develop a comrpessor and decompressor for multiple floating-point data streams, which are applied to our FPGA-based custom computing machine for fluid dynamics simulation.

Acknowledgments. This research was partially supported by Grant-in-Aid for Scientific Research (B) No.23300012 and Grant-in-Aid for Challenging Exploratory Research No.23650021 from the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

1. bzip2 (2010), `http://www.bzip.org/`
2. Burtscher, M., Ratanaworabhan, P.: FPC: a high-speed compressor for double-precision floating-point data. IEEE Trans. on Computers 58(1), 18–31 (2009)
3. Engelson, V., Fritzson, D., Fritzson, P.: Lossless compression of high-volume numerical data from simulations. In: Proceedings of Data Compression Conference (DCC), pp. 574–586 (September 2000)
4. Franaszek, P.A., Lastras-Montaño, L.A., Peng, S., Robinson, J.T.: Data compression with restricted parsings. In: Proc. of the Data Compression Conf. (2006)
5. Healy, D., Mitchell, O.: Digital video bandwidth compression using block truncation coding. IEEE Trans. on Communications COM-29(12), 1809–1817 (1981)
6. Ibarria, L., Lindstrom, P., Rossignac, J., Szymczak, A.: Out-of-core compression and decompression of large n-dimensional scalar fields. In: Proceedings of Eurographics, vol. 22(3), pp. 343–348 (September 2003)
7. Isenburg, M., Lindstrom, P., Snoeyink, J.: Lossless compression of predicted floating-point geometry. Computer-Aided Design 37(8), 869–877 (2005)
8. Katahira, K., Sano, K., Yamamoto, S.: FPGA-based lossless compressors of floating-point data streams to enhance memory bandwidth. In: Proceedings of the International Conference on Application-specific Systems, Architectures and Processors, pp. 246–253 (July 2010)
9. Kono, Y., Sano, K., Yamamoto, S.: Scalability analysis of tightly-coupled FPGA-cluster for lattice boltzmann computation. In: Proc. of the 22nd Intl. Conf. on Field-Programmable Logic and Applications, pp. 120–127 (August 2012)
10. Lim, J.S., Oppenheim, A.V.: Enhancement and bandwidth compression of noisy speech. In: Proceedings of the IEEE, vol. 67(12), pp. 1586–1604 (December 1979)
11. Lindstrom, P., Isenburg, M.: Fast and efficient compression of floating-point data. IEEE Transactions on Visual and Computer Graphics 12(5), 1245–1250 (2006)
12. Ratanaworabhan, P., Ke, J., Burtscher, M.: Fast lossless compression of scientific floating-point data. In: Proceedings of Data Compression Conference, pp. 133–142 (March 2006)
13. Sano, K., Hatsuda, Y., Yamamoto, S.: Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 234–241 (May 2011)
14. Sano, K., Katahira, K., Yamamoto, S.: Segment-parallel predictor for FPGA-based hardware compressor and decompressor of floating-point data streams to enhance memory i/o bandwidth. In: Proceedings of the Data Compression Conference, pp. 416–425 (March 2010)

15. Sukhwani, B., Abali, B., Brezzo, B., Asaad, S.: High-throughput, lossless data compression on FPGAs. In: 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 113–116 (May 2011)
16. Terasic Technologies, `http://www.terasic.com`
17. The GNU MPFR Library, `http://www.mpfr.org`
18. Tomari, H., Inaba, M., Hiraki, K.: Compressing floating-point number stream for numerical applications. In: 2010 First International Conference on Networking and Computing, pp. 112–119 (November 2010)
19. Tremaine, R.B., Franaszek, P.A., Robinson, J.T., Schulz, C.O., Smith, T.B., Wazlowski, M.E., Bland, P.M.: Ibm memory expansion technology (mxt). IBM Journal of Research and Development 45(2), 271–285 (2001)
20. Ueno, T., Kono, Y., Sano, K., Yamamoto, S.: FPGA-based implementation of compact compressor and decompressor of floating-point data-stream for bandwidth reduction. In: Proceedings of the 2012 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2012) (July 2012)

Hardware Acceleration of Matrix Multiplication over Small Prime Finite Fields

Shane T. Fleming and David B. Thomas

Imperial College London, London, United Kingdom
shane.fleming06@imperial.ac.uk

Abstract. Dense matrix-matrix multiplication over small finite fields is a common operation in many application domains, such as cryptography, random numbers, and error correcting codes. This paper shows that FPGAs have the potential to greatly accelerate this time consuming operation, and in particular that systolic array based approaches are both practical and efficient when using large modern devices. A number of finite-field specific architectural optimisations are introduced, allowing $n \times n$ matrices to be processed in O(n) cycles, for matrix sizes up to $n = 350$. Comparison with optimised software implementations on a single-core CPU shows that an FPGA accelerator can achieve between 80x and 700x speed-up over a Virtex-7 XC7V200T for $GF(2^k)$, but for $GF(3)$ and larger finite fields can provide practical speed-ups of 1000x or more.

Keywords: Galois Fields, Matrix Multiplication, Finite Fields, FPGA, Hardware Acceleration, Systolic Arrays.

1 Introduction

Matrix multiplication over finite fields (FF) is a common operation in many fields of science and engineering. One example is in cryptanalysis, where chained operations described as matrices must be multiplied together before being analysed for flaws [1]. Another is in the design of random-number generators, where exponentiation (i.e. repeated multiplication) of dense matrices is used to determine the period and quality of random number generators [2]. However, since matrix exponentiation is built on top of the matrix-matrix multiplication (MMM) kernel, performing fast MMM operations is the primary concern of this paper.

Current work published on the topic of MMM hardware acceleration has primarily been focused on the ring of real numbers ($\mathbb{R}$), represented as floating point and fixed point number representations such as in [3] and [4]. The research presented here hopes to expand upon this by investigating how this operation can be instead accelerated over FFs through use of an FPGA.

This paper explores how best to use FPGAs for MMM over finite fields, both in theoretical terms and practical terms. Our key contributions are:

– An analysis of the different available time-resource complexity trade-offs, showing that a large systolic array using $O(n)$ steps and $O(n^2)$ parallel computer resources is most appropriate for modern FPGAs.

- A flexible architecture for implementing MMM in current FPGA fabrics, which allows a single hardware description to be specialised for both matrix size and field size without any code changes.
- Evaluation of the hardware accelerator against existing software libraries, showing that while moderate speedup is achievable over $GF(2)$ and $GF(2^k)$, i.e. bits and strings of bits, it is possible to achieve speedups upwards of $1000\times$ for larger field sizes.

2 Background

Finite Fields: A finite field consists of a set of finite elements and two operations, addition and multiplication. They are often refered to as Galois Fields, $GF(p)$ where p is the number of elements within the field also known as the field size and is prime. For the purposes of this paper the number of bits b required to represent all p elements of the field is known as the field width. Certain properties also need to be satisfied in order to be classified as a finite field, as outlined by Shoup[5]. It is also possible to create finite fields for field sizes $GF(p^k)$ where k is a positive integer, these are sometimes referred to as extension fields. However, performing multiplication and addition is slightly more involved as each element is represented as unique polynomial expressions. The designs in this paper will deal with operations over $GF(p)$ and $GF(p^k)$ where p is a prime number, as it is possible to represent the fields as integers or polynomials and construct tables for multiplication and addition[6].

MMM Algorithms: The naive MMM operation computes the dot product of the corresponding row of the first operand matrix with the corresponding column of the second operand matrix for every element of the resultant. Therefore for square matrices of size n the asymptotic time complexity of the operation is $O(n^3)$.

More sophisticated algorithms improve this asmyptotic complexity, such as Strassens algorithm [7] or the Coppersmith-Winograd algorithm [8]. Strassens algorithm trades fewer multiplication operations for cheaper addition operations, reducing the complexity to $\approx O(n^{2.8074})$, while the Coppersmith-Winograd algorithm reduces the time complexity of the operation to $O(n^{2.3727})$.

The reduced complexity often comes at the expense of numerical stability which is a problem when using floating-point, but in the case of FFs all operations are exact[8]. However, these methods require large matrices that are outside the scope of what this paper aims to investigate.

3 Analysis

Both GPUs and FPGAs lend themselves readily to exploiting data parallelism. In this case FPGAs are preferred since the operation is being performed over finite fields and FPGAs are capable of performing functions using customised bit-widths, allowing the architecture to be tailored to the field size. In contrast, GPUs have fixed bit-widths meaning that some parts of the databus and functional units would be unused for most field sizes which typically only equire a few bits to represent them.

FPGA primitives are called Look-Up-Tables (LUTs) which are typically arrays of SRAM cells. L_i is used to specify the maximum number of LUT inputs available on a particular FPGA. Each combination of the L_i inputs has a corresponding SRAM cell, which means that these LUTs can be used to build complex computational functions.

It is possible to construct finite field multiplication and addition hardware as a table in LUTs. For small field sizes this will save a lot of resources compared to expensive modulo p, addition and multiplication hardware. However as the field size increases there will come a point where traditional hardware will start to be less resource intensive than using tables.

Due to the nature of this problem a direct tradeoff between the FPGA hardware usage and the time complexity exists. This section proposes four hardware algorithms that explore this tradeoff. Three metrics are used to analyse each algorithm, the resource complexity which is the rate the hardware resources usage increases as the problem size increases; time complexity which is the number of cycles that the operation takes to complete as the problem size increases and space complexity which is the rate that the amount of working memory (RAM) required to store the matrices increases as the problem size increases.

On top of these core metrics other factors also need to be considered for each design. The locality and complexity of routing between logical units is also important, as the longest connection, known as the critical path, will determine the maximum clock rate that the device can operate at. For all the algorithms it is assumed that the required matrices are all stored in block RAM, within the FPGA device, as the bandwidth to external memory is typically an order of magnitude lower than the bandwidth to block RAM. We assume that the number of MMMs executed on the FPGA will be large, with movement to and from main memory relatively infrequent. This is true in the case of MMM exponentiation, as well as blocked MMM operations which is where matrices are partitioned into submatrices, such as in Strassens algorithm.

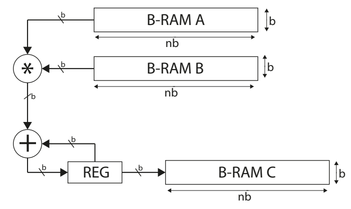
Algorithms: Figure 1 shows diagrams for the *sequential*, *vector parallel*, *systolic array* and *fully parallel* algorithms. For all the algorithms analysed, the space complexity was never a limiting factor so will not be considered when comparing the various algorithms since FPGAs have a lot of BRAMs.

The sequential algorithm in Figure 1(a) has one multiplier, one adder and an accumulation register. This gives it a low (constant) resource complexity however it has a high time complexity of $O(n^3)$ for a $n \times n$ MMM operation.

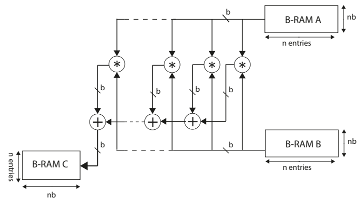
The vector parallel algorithm in Figure 1(b) has n multipliers and $n-1$ adders. This architecture has a poorer resource complexity compared to the sequential implementation however since this computes a dot product in parallel, the time complexity is reduced to $O(n^2)$.

A systolic array (SA), seen in Figure 1(c), is a mesh topology of processing elements (PEs) that are all locally connected to their immediate neighbours [9]. Data is fed into the PEs at the edges of the array and each PE passes data along to its immediate neighbours on the right and beneath. Each PE works in parallel to compute the dot product required for each element of the resultant. To compute each dot product requires n cycles so the overall algorithm has a time complexity of $O(n)$. However each PE requires an adder and multiplier, and since there are n^2 elements of the resultant the resource complexity is $O(n^2)$.

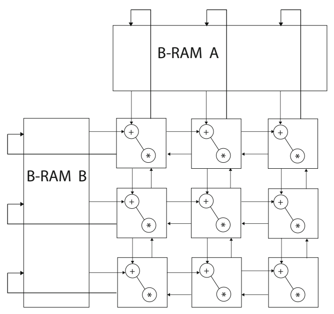
Finally Figure 1(d) shows a fully parallel implementation where every operation of every dot product is computed in parallel. This operation requires a large amount of resources with a complexity of $O(n^3)$, but it has constant time complexity.



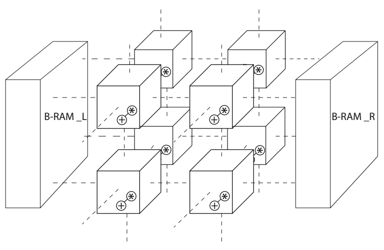
(a) Sequential MMM algorithm



(b) Vector Parallel MMM algorithm



(c) Systolic Array MMM algorithm



(d) Fully Parallel MMM algorithm

Fig. 1. Diagrams of MMM algorithms

For each of the algorithms presented in this section the amount of FPGA resources used in terms of LUTs and the number of cycles taken to perform each operation were estimated and graphed shown in Figure 2. For all estimates it is assumed that the operation is being performed over $GF(2)$ and that the data required is already present on the FPGA device, i.e. zero data transfer time, for measured results in real devices see Section 5. These graphs demonstrate the tradeoff between hardware usage and the number of cycles required to complete the operation.

Figure 2 demonstrates that the largest matrices that could be processed on an XC7V2000T for the fully parallel algorithm operating over $GF(2)$ is approximately 70×70, and it would take considerable synthesis effort to reach this already quite small size. This algorithms is also 3D and would be difficult to place and route since FPGAs are physically 2D structures. To supply inputs to this algorithm also requires an unrealistically high bandwidth of data and for these reasons this design was not considered. In the vector parallel architecture the adder chain grows as the problem grows. This produces a long critical path and routing complications for large n. The sequential algorithm has a very poor time complexity and will definitely not be able to outperform standard software techniques.

This leaves the systolic array (SA) as the chosen design. This algorithms provides a middle ground between the fully parallel implementation and vector parallel. Figure 2 shows that on a XC7V2000T matrices of up to $\approx 400 \times 400$ for $GF(2)$ should be supported. However, these are not the only motivating factors behind choosing the SA design. It has the same data bandwidth as the vector parallel architecture yet a much higher performance and every connection between the PEs are local and highly structured which makes the critical paths small and device placement easier.

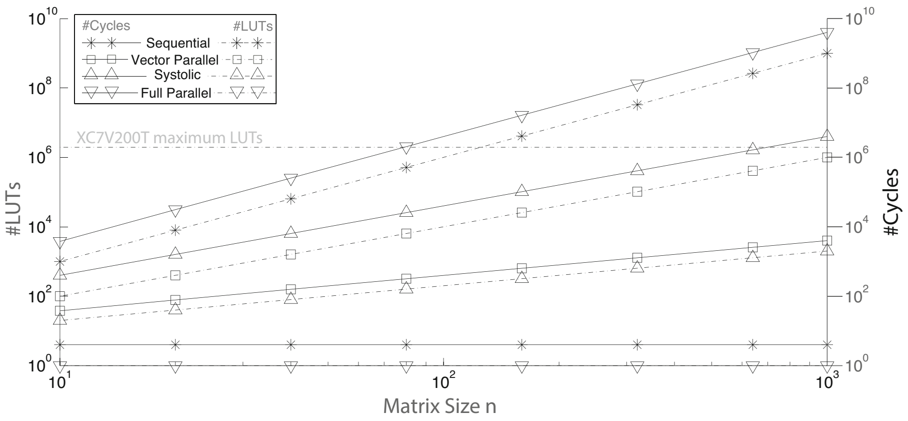


Fig. 2. Estimated hardware usage (LUTs) and time complexities (cycles) for each architecture between $n = 10$ and $n = 1000$

4 Proposed Solution

Systolic Array: When designing a SA (systolic array) two important things need to be considered: how does data flow from PE (processing element) to PE, and how are the hundreds of PEs controlled in order to ensure that data is sent and received at the correct point in time.

Figure 3 shows how data moves through a SA. Each processing element multiplies and accumulates data present at both its inputs. At the same time input data is passed across to the PE's direct neighbour on the right and passed down to the direct neighbour below.

Each PE calculates an element of the resultant and in order to do this each must perform a dot product operation. This dot product occurs sequentially using single cycles multiply-add-accumulates and a total of n cycles is required. To ensure that the correct elements of the operands rows and columns appear at each PE at the appropriate time the inputs need to be skewed in the fashion demonstrated in Figure 3. This has two effects: first, since the SA needs to fill and drain it increases the number of execution cycles by a constant factor; and second, since the inputs need to be skewed it either increases the amount of BRAM used to store the inputs, or increases the logic resources used to form delay logic to skew the input.

Figure 3 shows how the two input matrices pass through the SA as two wavefronts that combine to form a wavefront of computation. Once the computation wavefront has fully passed through a PE, the value of the corresponding element of the resultant is contained in the PEs internal accumulation register. This is shown as the PE being filled with a pattern in Figure 3.

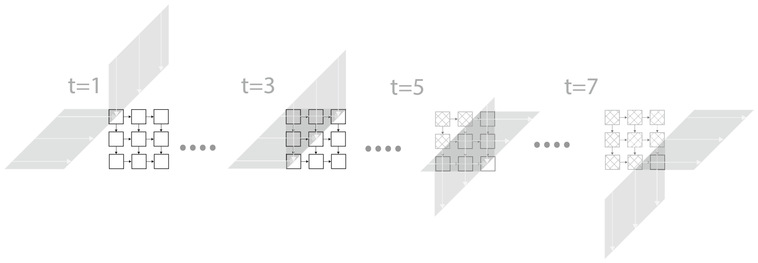
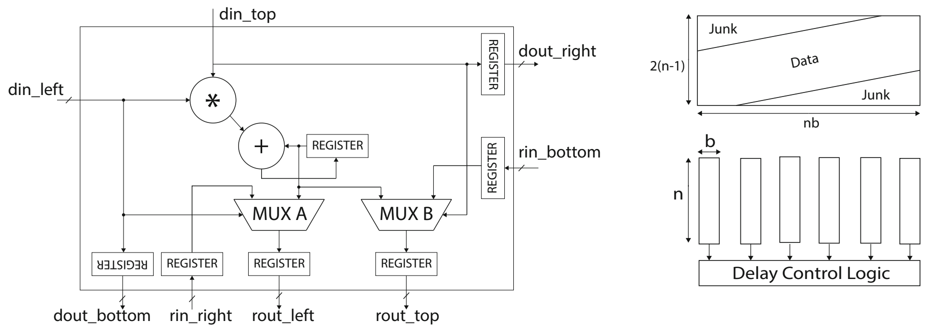


Fig. 3. Diagram to show how data flows through the processing elements of a 2×2 systolic Array

The Processing Element: The PE is the most fundamental part of the SA. Any small reduction in the resource requirements of the PE design will result in a large reduction for the overall SA, as they are unrolled n^2 times. This section explains how the size of the PE was kept as small as possible through optimising the required control signals and the arithmetic hardware.

Figure 4(a) shows the internals of a processing element, where data is fed into the left and the top inputs and multiplied. The result of this multiplication is then accumulated and stored inside a central register. Multiplexers A and B control the result outputs, which are either assigned the value of the result inputs or the value stored in the accumulation register, as discussed further later.

Since this design is for small finite fields, the multiplication and the addition are performed by generating a look-up table for all the possible inputs of each operation across the entire finite field which is then stored in LUTs. This makes multiplication and addition faster and smaller but limits the design to small prime fields. If we define M_L as the number of LUTs required to build a multiplication circuit, p as a prime number of field elements (field size) and b



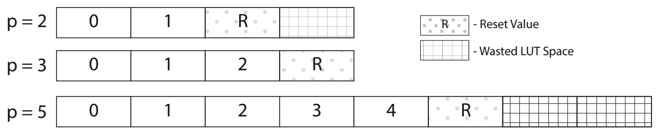
(a) (b)

Fig. 4. Diagram showing (a) The internal components of a processing element, and (b) possibilities of how memory fed into the systolic array could be organised

+	0	1	2	3
0	0	1	2	0
1	1	2	0	0
2	2	0	1	0
3	0	0	0	0

x	0	1	2	3
0	0	0	0	3
1	0	1	2	3
2	0	2	1	3
3	3	3	3	3

(a)



(b)

Fig. 5. Diagrams to show (a) Multiplication and Addition look up tables with added reset value over $GF(3)$, and (b) how all combinations of b bits are used for a particular p

as the number of bits required to represent all p elements (field width), where $b = \lceil log_2(p) \rceil$ then this method only pays off when the condition $b\lceil \frac{b}{L_i} \rceil < M_L$ is satisfied.

Before an MMM operation occurs the accumulation register of every PE needs to be reset. Each PE also needs to be aware of when it has completed a full dot-product operation. A symbol can be propagated through the inputs of the PE to perform both of these tasks. Since this architecture is operating over a prime finite field, p, all the numerical operations occur on numbers in the range of 0 to $p - 1$. With the exception of a special case where $p = 2$ the size of all the prime fields will be odd numbers. This means that not all the combinations of the b bits will be used. In particular the numerical value of p can be represented by b bits but does not represent a valid field element. The value p can be used as the control signal since it is unused and does not require any extra bits to represented it.

The multiplication and addition results will not entirely fill the LUTs except for the case when operating over $GF(2^k)$. This extra space can be used to add reset functionality. Figure 5(b) shows how the reset signal takes up space in the LUT. It can be seen that for $GF(2^k)$ this method is identical to having an extra reset wire between the PEs. However for the case when operating over $GF(p > 2)$ this is an improvement as no extra wires are required. So this method is at worst the same as having an extra reset wire, and in the best (and most common) case reset functionality is achieved for free.

In order to realise this the LUTs need to be adapted in the following manner: when a value p appears at either input of the multiplication LUT the output is the value p thus passing the reset signal onto the addition LUT. By modifying the addition LUT so that any number added to p returns 0, it is possible set the accumulation register to 0 regardless of that register's current state. An example of how the multiplication and addition tables are extended for GF(3) can be seen in Figure 5(a).

Extracting Data from PEs: Every element of the resultant is calculated and stored in the corresponding PEs accumulation register, so there needs to be a method for extracting these results once the multiplication is complete. The value p is fed into each PE once it has completed to reset the accumulation register, so this value can also be used to indicate that the PE should stream out its result. Figure 4(a) has four wires, *rin_bottom*, *rin_right*, *rout_top* and *rout_left* which are used to stream out the result in the opposite direction of the input data. The following simple rules demonstrate how data can be extracted from the SA in the same skewed format as it's input. Figure 6 demonstrates these rules for one row or column of a 3×3 SA. If a p (reset signal) is detected at the input of a PE then the *rout* ports are assigned the value of the accumulation register, otherwise they are assigned the values present at the *rin* ports. This method preserves the same staggered data format as the input and allows it to be fed straight back into the SA even if the previous result has not yet completed, giving the system an effective throughput of n cycles. Figure 6 also demonstrates that if p is fed into the *rin* ports of the PEs along the bottom and right edges of the SA then there is a reset signal present at the start and end of the output data.

5 Results

The previous sections present an argument for using a systolic array based architecture for MMM over finite fields, and provide a parametrisable concrete architecture designed for modern FPGAs. In this section we measure the achievable performance of the architecture in terms of resource utilisation, execution time, and speedup over software for a range of matrix and field sizes. The target use-case for the MMM architecture is as a building-block in applications such as exponentiation and blocked matrix multiplication where the IO to compute rate is low. Therefore due to the large number of operations performed on the device the main focus is on the performance of the MMM kernel, with less engineering attention given to IO. The SA architecture was placed and routed onto a XC7V200T device for various field and matrix sizes. In each case the execution time for matrix exponentiation was calculated based on the clock rate and

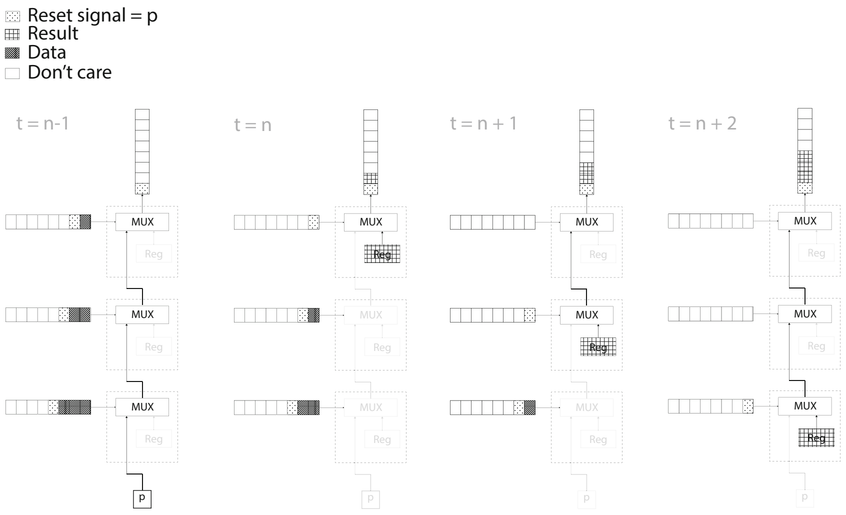


Fig. 6. Diagram showing how results are extracted from the SA

the known throughput of the device. Transfer time was measured on an actual device using an Alpha-Data ADM-XRC-5T2 acceleration card. This was then compared to single core execution using LinBox [10] the fastest known library for MMM over $GF(p > 2)$ and M4RI [11] the fastest known library over $GF(2)$. These software results were also comparable to results produced by SAGE which is a mathematical operations package that under the hood tries to automatically select optimal algorithms and in this case it makes use of the LinBox and NTL libraries. The fastest results were produced by SAGE on a single Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz and were ultimately used for the execution time comparison.

Figure 7 shows the place and route (PaR) results for SAs with a variety of bit widths b and matrix sizes n on a large Virtex7 XC7V200T device. The result for $b = 2$ is for an SA that can operate over both $GF(2)$ and $GF(3)$ since each requires two bits to represent the field, with the only actual difference being the values stored in the generated multiplication and addition LUTs which are calculated at synthesis time.

For $GF(2)$ and $GF(3)$ the largest SA that can be mapped and PaR in a few hours is approximately 360×360, any larger than this and the required time increases by an order of magnitude. This is very close to the estimated resource usage of 400×400, but it is slightly lower since it is very difficult to fully utilise all the logical resources on an FPGA device due to the time required to optimally pack slices.

As the number of bits required to represent the field increases, the LUT usage increases at a faster rate the matrix size increases. This makes sense as the size of each of the n^2 PEs increases since more LUTs are required for addition and multiplication. Unfortunately this means that the SA is only effective when operating in the region of finite fields that can be represented by $2 \leq b \leq 5$.

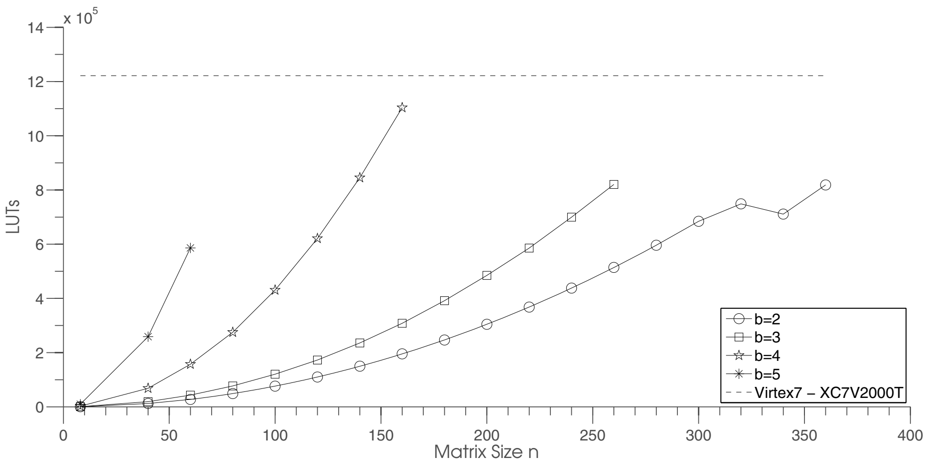


Fig. 7. Post place and route LUT usage of an SA for various bit widths on a Virtex 7 XC7V2000T device[1]

Implementing the SA was a direct implementation of the architecture described in Section 3. The main engineering hurdle was synthesising for $n > 120$ because the array had to be partitioned into multiple black boxes that were connected together. A Virtex FPGA connected via PCIe using a ADM-XRC-5T2 alpha-data card to a host computer was used to obtain accurate data transfer times for sending the matrix data to the device, which is slow and could be greatly improved upon.

In order to perform A^q where A is an $n \times n$ matrix, the SA requires n cycles to fill for the first MMM, n cycles to perform each subsequent operation and n cycles to drain the array after the last operation. Using a naive "square and multiply" for exponentiation this gives an execution time of $T_{SA} = 2T_{IO}(n) + (qn + n)\frac{1}{f}$ where T_{SA} is the execution time for the entire kernel operation, $T_{IO}(n)$ is the time to upload the matrix to the device and f is the clock rate of the device.

To calculate the speedup results a 3-point geometric moving average of the clock rate was taken to show the overall trend. Virtex-7 FPGA devices have a very high number of resources, which is due to a manufacturing processes known as Stacked Silicon Interconnect (SSI) technology. This technology is where multiple Super Logic Regions (SLR) are connected using a Silicon Interposer. Connections between theses SLR regions are made over what is called a Super Long Line (SLL) routes that are located within the Silicon Interposer [12]. It is these connections that seem to limit the clock rate as the SA grows in size, however the clock rate never drops below 85MHz. This also means that floorplanning was not attempted to further improve performance as improving clock rate would only lead to a two or three times improvement in performance.

[1] The reason for the lower LUT count at $n = 340$ is currently unknown. Reports produced by the tools provided no insight and more investigation is required.

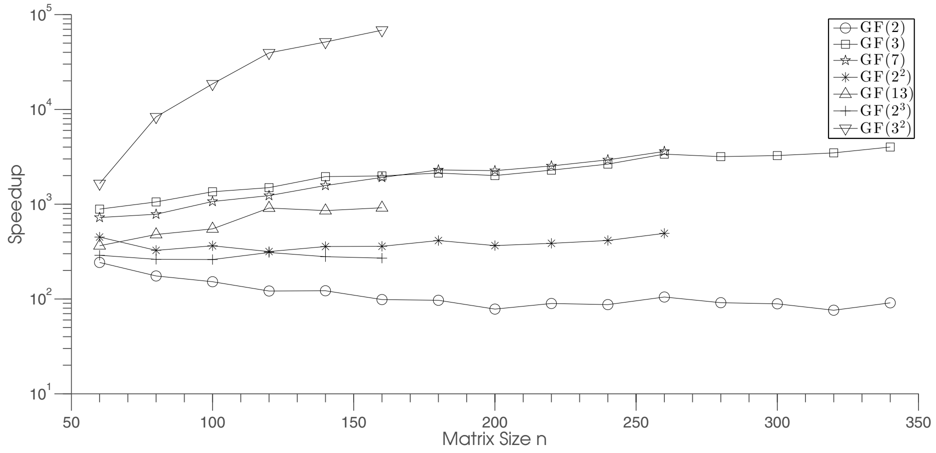


Fig. 8. Speedup of Systolic Array over software (SAGE) for increasing matrix size, Including data transfer times to and from the FPGA device over PCIe

Figure 8 shows the speedup of the SA against SAGE. When operating over $GF(2)$ an initial speedup of over $\times 100$ was observed which then settles to $\approx \times 40$. As a higher speedup could be observed for operations over $GF(2^2)$ and $GF(2^3)$. This is due to it being a slightly more complex process that involves strings of bits instead of just single bit-wise operations. For $GF(p > 2)$ a much larger speedup is observed, and this grows as the problem size increases. A very large speedup is observed for $GF(3^2)$ however this is not a fair comparison as it has obviously not been optimised in software.

The marked difference between $GF(2^k)$ and $GF(p > 2)$ is because $GF(2^k)$ has specific properties, which are aggressively exploited in software by M4RI through the use of fast parallel bit-wise instructions via SIMD instructions. They are not exploited by the FPGA SA which uses the same code for all fields. In principle this FPGA design could be further optimised for $GF(2)$, for example by processing blocks of four bits in parallel, but the more interesting practical case is actually for $GF(p > 2)$, as this is currently the slowest operation in software.

6 Conclusion

This paper has demonstrated that FPGAs can be used to greatly accelerate the time consuming MMM operation over $GF(p > 2)$ where p is prime, moderately accelerate for $GF(2^k)$ and less but still significantly so for $GF(2)$. Through examining a number of various algorithms it has shown that a systolic array design, which is a 2D mesh of processing elements, was most suitable. Design details were given and optimisations aiming to reduce the resource requirements were shown. These optimisations included, packing the multiplication and addition operations into look-up tables and reducing the resource requirements for resetting and controlling the system over $GF(p > 2)$.

The design was then placed and routed onto a target Virtex-7 XC7V2000T device, and the total execution time including IO transfers was measured. This was compared to the software package SAGE for $GF(p = 2)$, $GF(p > 2)$ and $GF(2^k)$ on a single Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz.

Speedups up to $\times 100$ were shown for operations over $GF(2)$, while for $GF(2^k)$ and $GF(p)$ speedups of up to $\times 300$ and upwards of $\times 1000$ were observed respectfully. Limitations in the design were shown by the maximum realisable field width supported being $2 < b \leq 5$ and the maximum matrix size being 360×360 for $b = 2$ and decreasing to 60×60 for $b = 5$.

Further work could be, investigating the expansion of using this as an MMM kernel for Strassens algorithm with allowing the ability to perform the MMM operation for any sized matrix, and examining large sparse blocked MMM operations where dense blocks are computed on the SA and sparse blocks are computed on the host computer. Ultimately this design could be integrated with the SAGE package to improve the performance of these operations.

References

1. Albrecht, M.: Algorithmic Algebraic Techniques and their Application to Block Cipher Cryptanalysis. PhD thesis, Royal Holloway, University of London (2010)
2. Thomas, D., Luk, W.: Fpga-optimised uniform random number generators using luts and shift registers. In: 2010 International Conference on Field Programmable Logic and Applications (FPL), pp. 77–82. IEEE (2010)
3. Zhuo, L., Prasanna, V.: Scalable and modular algorithms for floating-point matrix multiplication on fpgas. In: Proceedings of the18th International Parallel and Distributed Processing Symposium, p. 92. IEEE (2004)
4. Bensaali, F., Amira, A., Sotudeh, R.: Floating-point matrix product on fpga. In: IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2007, pp. 466–473. IEEE (2007)
5. Shoup, V.: A computational introduction to number theory and algebra. Cambridge University Press (2008)
6. Shoup, V.: Ntl: A library for doing number theory (2001)
7. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik 13(4), 354–356 (1969)
8. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of symbolic computation 9(3), 251–280 (1990)
9. Kung, H., Leiserson, C.E.: Systolic arrays (for vlsi). Society for Industrial & Applied, 256 (1979)
10. Dumas, J., Gautier, T., Giesbrecht, M., Giorgi, P., Hovinen, B., Kaltofen, E., Saunders, B., Turner, W., Villard, G., et al.: Linbox: A generic library for exact linear algebra. In: Proceedings of the 2002 International Congress of Mathematical Software, pp. 40–50. World Scientific Pub., Beijing (2002)
11. Albrecht, M., Bard, G.: M4ri–linear algebra over gf (2) (2008)
12. Dorsey, P.: Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency. Xilinx White Paper: Virtex-7 FPGAs, 1–10 (2010)

Flexible Design of a Modular Simultaneous Exponentiation Core for Embedded Platforms

Geoffrey Ottoy[1,2], Bart Preneel[2], Jean-Pierre Goemaere[1,3], and Lieven De Strycker[1,3]

[1] KAHO Sint-Lieven, DraMCo Research Group,
Gebroeders de Smetstraat 1, 9000 Gent, Belgium
geoffrey.ottoy@kahosl.be
http://www.dramco.be/

[2] KU Leuven, COSIC and IBT,
Kasteelpark Arenberg 10, bus 2446, 3001 Leuven-Heverlee, Belgium
http://www.esat.kuleuven.be/cosic/

[3] KU Leuven, TELEMIC Research Group,
Kasteelpark Arenberg 10, bus 2444, 3001 Leuven-Heverlee, Belgium
http://www.esat.kuleuven.be/telemic/

Abstract. In this paper we present a flexible hardware design for performing Simultaneous Exponentiations on embedded platforms. Simultaneous Exponentiations are often used in anonymous credentials protocols. The hardware is designed with VHDL and fit for use in embedded systems. The kernel of the design is a pipelined Montgomery multiplier. The length of the operands and the number of stages can be chosen before synthesis. We show the effect of the operand length and number of stages on the maximum attainable frequency as well as on the FPGA resources being used. Next to scalability of the hardware, we support different operand lengths at run-time. The design uses generic VHDL without any device-specific primitives, ensuring portability to other platforms. As a test-case we effectively integrated the hardware in a MicroBlaze embedded platform. With this platform we show that simultaneous exponentiations with our hardware are performed 70 times faster than with an all-software implementation.

Keywords: Montgomery Multiplier, Simultaneous Exponentiation, Pipelining, VHDL, Embedded System.

1 Introduction

Making high-performance implementations of Public-Key Cryptosystems (PKCs) on embedded and mobile devices is a daunting task. PKCs are often used for privacy-friendly identity management, but also in access control or mobile payment applications. Especially for RSA-based protocols, the required multi-base modular exponentiations pose a problem when it comes to execution times on resource-constrained devices (*e.g.* stand-alone terminals or access points). This specific operation is presented in (1).

$$\prod_{i=0}^{j} g_i^{e_i} \bmod m \ . \tag{1}$$

For this kind of computations, we designed an open source VHDL IP core for use in embedded platforms.[1] This IP core is designed with several requirements in mind. First of all, the design should be deployable on a wide range of platforms. We achieved this by writing as much generic VHDL code as possible. We have specifically opted not to use device-specific FPGA primitives to ensure portability to other platforms. The design is described in a structural and modular way so changing particular parts can be done with relative ease. The length of the operands can be chosen before synthesis and to some extent at runtime. Furthermore, the design parameters can be changed to achieve the desired operating frequency.

On the other hand, flexibility should not be an excuse for low performance, both in resource usage (silicon) as in timing. To achieve this we used a pipelined version of a Montgomery multiplier as the kernel of our design. The result is a flexible, yet practical design.

This paper is organized as follows. We start with an explanation of our implementation strategy (Sec. 2). This is done for the complete design as well as for the pipelined Montgomery multiplier. The timing results and resource usage are presented in Sec. 3. Conclusions and proposals for future work can be found in Sec. 4.

2 Implementation Strategy

Simultaneous Exponentiation. The most straightforward way of performing modular exponentiation is by repeated squarings and multiplications to get the final result [1, 2]. The square and multiply step can be performed either in parallel [3] or sequentially [4]. The exponentiation can be extended to an efficient simultaneous exponentiation algorithm. The case with 2 bases is presented in Algorithm 1, where *Mont()* designates a Montgomery multiplication.

The modulus m and the bases g_0 and g_1 all have a length of n bits, whereas the length of the exponents e_0 and e_1 is t bits. Furthermore, it can be seen that the algorithm requires $R^2 \bmod m$ which is $2^{2n} \bmod m$. We assume R^2 can be provided or precomputed.

Looking at Algorithm 1, a logical design choice is to only implement a multiplier and to implement the control logic in such a way that it can either run a single multiplication or run the main loop automatically.

Following a standard design method, we implemented the IP core as a memory mapped peripheral. The software on the embedded processor can manage the

[1] The design is published online on OpenCores; project page: `http://opencores.org/project,mod_sim_exp`

Algorithm 1. Montgomery simultaneous exponentiation

```
Input: $g_0$, $g_1$, $e_0 = (e_{0_{t-1}} \cdots e_{0_0})_2$, $e_1 = (e_{0_{t-1}} \cdots e_{0_0})_2$, $R^2 \bmod m$, $m$
Output: $g_0^{e_0} \cdot g_1^{e_1} \bmod m$
 1: $\tilde{g}_0 := \text{Mont}(g_0, R^2)$, $\tilde{g}_1 := \text{Mont}(g_1, R^2)$, $\tilde{g}_{01} := \text{Mont}(\tilde{g}_0, \tilde{g}_1)$
 2: $a := \text{Mont}(R^2, 1)$                    ▷ This is the same as $a := R \bmod m$.
 3: for $i \leftarrow (t-1)$ downto 0 do
 4:     $a := \text{Mont}(a, a)$
 5:     switch $e_{1_i}$, $e_{0_i}$
 6:         0, 1 : $a := \text{Mont}(a, \tilde{g}_0)$
 7:         1, 0 : $a := \text{Mont}(a, \tilde{g}_1)$
 8:         1, 1 : $a := \text{Mont}(a, \tilde{g}_{01})$
 9: $a := \text{Mont}(a, 1)$
10: return $a$
```

core's behavior by writing to a control register (Fig. 1). Memory is provided to store the required operands. The hardware can signal the processor of certain events using an interrupt line (IRQ).

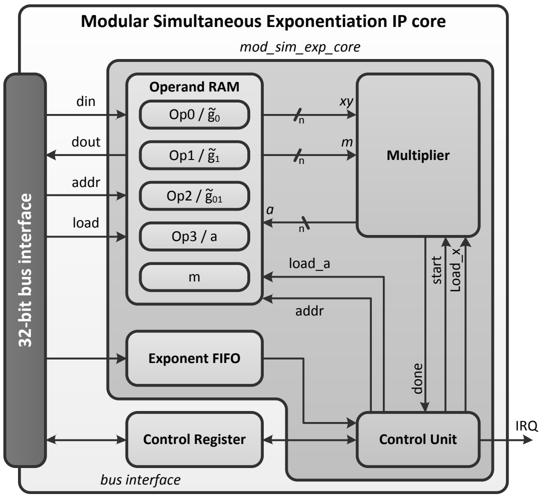


Fig. 1. Block diagram of the Modular Simultaneous Exponentiation IP core

Montgomery Multiplication. As mentioned before, the kernel of our circuit is a pipelined Montgomery multiplier which relies on the popular Montgomery's algorithm (2) [5–11] to perform modular multiplications. This algorithm allows for very efficient hardware implementations, which is an advantage when we strive for a flexible and generic design.

$$R = x \cdot y \cdot R^{-1} \bmod m \tag{2}$$

Nedja and Mourelle [3] have shown that for operands larger than 512 bits, a *systolic array implementation* improves the *time* × *area* product over other implementations. We take their work as a starting point. With their modified algorithm, every bit of the (intermediate) result is driven by just a multiplexer and an adder. Together they form a systolic array cell. A right shift operation – inherent to the Montgomery algorithm – ensures that the intermediate result is never larger than $n + 2$ bit (final carry included).

In our design, systolic array cells are grouped into k stages. This has two advantages. First of all it brakes up the long carry-chain in the adders, so we can achieve higher clock frequencies.[2] Secondly, this approach allows for pipelining, further speeding-up the design. A drawback is that, by increasing the number of stages, more flip-flops will be used.

Pipeline Operation. Because of the right-shift operation, a stage can only compute a step every 2 τ_s, where τ_s is the time it takes a stage to actually complete a step. In this case τ_s is 1 clock cycle. For one multiplication, the total computation time $T_{k,n}$ for an n-bit operand with a k-stage pipeline is given by (3).

$$T_{k,n} = [k + 2(n-1)]\,\tau_s \ . \tag{3}$$

Complete Multiplier. The complete multiplier design is shown in Fig. 2. For every bit of x the multiplier computes an intermediate result. The final result is then reduced to ensure that it is smaller than m. An extra adder computes $m + y$ which is required by the systolic array cells.

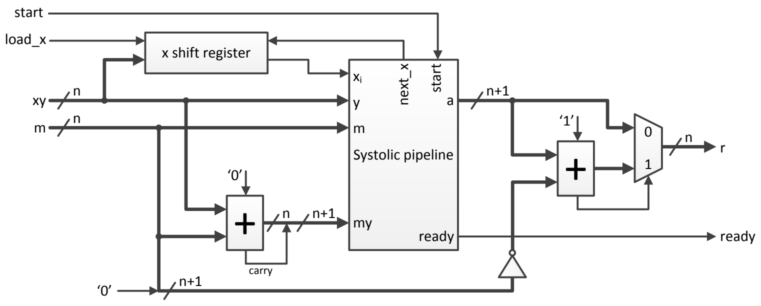


Fig. 2. Multiplier structure. For clarification the *my* adder and reduction logic are depicted separately, whereas in practice they are internal parts of the stages.

3 Core Performance

Resource Usage. The number of flip-flops and LUTs for the multiplier is given by (4) and (5). Note that while the number of FFs is also determined by k, the

[2] The path of the carry in the adders is the critical timing path.

number of LUTs is only dependent on the length of the operands. Because we use RAM to store the operands, the complete IP core uses about the same amount of LUTs and flip-flops as the multiplier.

$$FFs = 5 + 2 \cdot n + 6 \cdot k + \lceil \log_2(n) \rceil + \lceil \log_2(k) \rceil \tag{4}$$

$$LUTs = \begin{cases} 8 \cdot n & \text{for 4-input LUTs} \\ 6 \cdot n & \text{for 6-input LUTs} \end{cases} \tag{5}$$

Timing. For the multiplication, execution time is given by (3), where τ_s is defined by the operating frequency. Since the maximum frequency is highly influenced by the latency in the critical path, we can expect to achieve higher frequencies for shorter stage lengths (see Fig. 3(a)). We obtained this figure from the static timing analysis during the synthesis step in the design process. For $s \leq 4$, we see that f_{max} saturates to a maximum. This is probably due to the slice architecture which, for the Virtex-6, contains 4 LUT-FF pairs. Maximum execution speed will hence be at $s = 4$ (Fig. 3(b)). We can also see that f_{max} is as good as independent of n. Furthermore, when knowing the operating frequency ($f_{op.}$) beforehand, one can choose s in such a way that the number of FFs is minimal while $f_{max} \geq f_{op.}$.

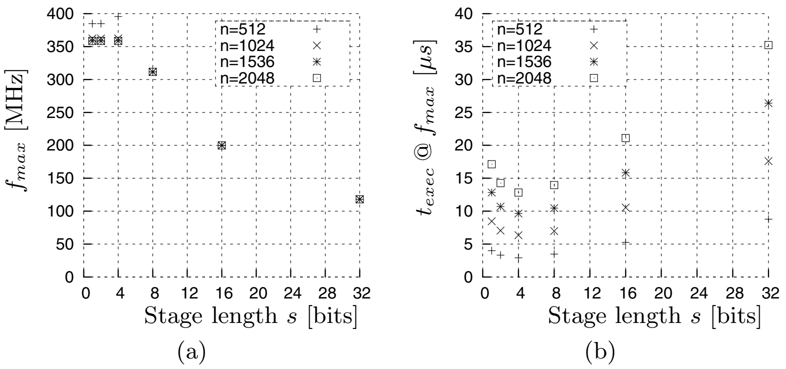


Fig. 3. Influence of s on multiplication f_{max} and t_{exec} for a Virtex-6 xc6vlx240t

Practical Implementation. If we exclude precomputation and postcomputation, the average execution time for simultaneous exponentiation is $\frac{7}{4} \cdot t \cdot T_{k,n}$ and $\frac{3}{2} \cdot t \cdot T_{k,n}$ for a single base exponentiation.

We made a practical setup [12] with our IP core connected to a MicroBlaze processor. Both the processor and the IP core run at the same clock frequency of 100 MHz. To illustrate the execution speed of our implementation, we compared it with a software implementation on the same system.[3]

In theory, for a simultaneous exponentiation with $n = 1536$, $t = 1024$ and $s = 16$ it will take 56.73 ms @ 100 MHz. In practice it takes 68 to 72 ms. This

[3] The software is implemented using the GMP library – Online: `http://gmplib.org/`

comprises of 57 ms for the main computation, 5 ms for computing R^2 in software, 6 ms for precomputation and postcomputation and about 1 ms for reading and writing data to the core's memory.

The same computation takes 4800 to 4900 ms in software. This means that this hardware implementation is roughly 70 times faster than software on the same platform.

4 Conclusions and Future Work

In this paper we have presented a flexible VHDL design of a modular exponentiation core for embedded platforms. The hardware supports simultaneous exponentiation and is designed for implementation in an embedded processor system on configurable hardware. Furthermore, it does not use any device-specific primitives *e.g.* multipliers, making it suitable for use across different platforms. Only RAM is used for storing operands.

We provided insight in how the pipelining is implemented, and in how the stage length affects the operating frequency and resource usage. We also compared the hardware implementation with a software implementation on the same embedded platform.

Our future goal is to further maintain the core and add some enhancements, including support for different types of adders (*e.g.* Carry-Select adders) and bus interfaces. Also the core's resistance against side-channel attacks should be examined. To that extent, implementing countermeasures against differential power analysis is "on the todo list".

The core has been made open source. All developments can be followed at `http://www.opencores.org/project,mod_sim_exp`

References

1. Blum, T., Paar, C.: High-radix Montgomery modular exponentiation on reconfigurable hardware. IEEE Transactions on Computers 50(7), 759–764 (2001)
2. Sutter, G.D., Deschamps, J.-P., Imaña, J.L.: Modular Multiplication and Exponentiation Architectures for Fast RSA Cryptosystem Based on Digit Serial Computation. IEEE Transactions on Industrial Electronics 58(7), 3101–3109 (2011)
3. Nedjah, N., de Macedo Mourelle, L.: Three Hardware Architectures for the Binary Modular Exponentiation: Sequential, Parallel, and Systolic. IEEE Transactions on Circuits and Systems – I: Regular Papers 53(3), 627–633 (2006)
4. de la Piedra, A., Touhafi, A., Cornetta, G.: Cryptographic accelerator for 802.15.4 transceivers with key agreement engine based on Montgomery arithmetic. In: 2011 18th IEEE Symposium on Communications and Vehicular Technology in the Benelux (SCVT), November 22-23, pp. 1–5 (2011)
5. Montgomery, P.L.: Modular Multiplication Without Trial Division. Mathematics of Computation 44(170), 519–521 (1985)
6. Shigemoto, K., Kawakami, K., Nakano, K.: Accelerating Montgomery Modulo Multiplication for Redundant Radix-64k Number System on the FPGA Using Dual-Port Block RAMs. In: IEEE/IFIP Intl. Conf. on Embedded and Ubiquitous Computing, EUC 2008, vol. 1, pp. 44–51 (2008)

7. He, Y., Chang, C.-H.: A New Redundant Binary Booth Encoding for Fast 2^n-Bit Multiplier Design. IEEE Transactions on Circuits and Systems I: Regular Papers 56(6), 1192–1201 (2009)
8. Bajard, J.-C., Didier, L.-S., Kornerup, P.: An RNS Montgomery Modular Multiplication Algorithm. IEEE Trans. on Computers, 766–776 (1998)
9. Phillips, B.: Modular multiplication in the Montgomery residue number system. In: Conf. Record of the Thirty-Fifth Asilomar Conf. on Signals, Systems and Computers, vol. 2, pp. 1637–1640 (2001)
10. Örs, S.B., Batina, L., Preneel, B., Vandewalle, J.: Hardware implementation of a Montgomery modular multiplier in a systolic array. In: Proc. International Parallel and Distributed Processing Symp., April 22-26, p. 184-2 (2003)
11. Blum, T., Paar, C.: Montgomery modular exponentiation on reconfigurable hardware. In: Proc. 14th IEEE Symp. on Computer Arithmetic, pp. 70–77 (1999)
12. Ottoy, G., Martens, J., Saeys, N., Preneel, B., De Strycker, L., Goemaere, J.-P., Hamelinckx, T.: A Modular Test Platform for Evaluation of Security Protocols in NFC Applications. In: De Decker, B., Lapon, J., Naessens, V., Uhl, A. (eds.) CMS 2011. LNCS, vol. 7025, pp. 171–177. Springer, Heidelberg (2011)

Architecture for Transparent Binary Acceleration of Loops with Memory Accesses

Nuno Paulino[1], João Canas Ferreira[1], and João M.P. Cardoso[2]

[1] INESC TEC and Faculty of Engineering, University of Porto, Portugal
nuno.paulino@fe.up.pt, jcf@fe.up.pt
[2] INESC TEC and Department of Informatics Engineering,
Faculty of Engineering, University of Porto, Portugal
jmpc@fe.up.pt

Abstract. This paper presents an extension to a hardware/software system architecture in which repetitive instruction traces, called Megablocks, are accelerated by a Reconfigurable Processing Unit (RPU). This scheme is supported by a custom toolchain able to automatically generate a RPU tailored for the execution of one or more Megablocks detected offline. Switching between hardware and software execution is done transparently, without modifications to source code or executable binaries. Our approach has been evaluated using an architecture with a MicroBlaze General Purpose Processor (GPP) softcore. By using a memory sharing mechanism, the RPU can access the GPP's data memory, allowing the acceleration of Megablocks with load/store operations. For a set of 21 embedded benchmarks, an average speedup of 1.43× is achieved, and a potential speedup of 2.09× is predicted for an implementation using a low overhead interface for communication between GPP and RPU.

Keywords: reconfigurable processor, memory access, Megablock, instruction trace, MicroBlaze, hardware acceleration, FPGA.

1 Introduction

The use of dedicated hardware co-processors is an often-adopted solution to accelerate demanding computational kernels. However, the hardware/software (HW/SW) partitioning steps required to implement such co-processor based systems are time consuming, requiring hardware expertise and integration with a host system. Runtime reconfigurable coprocessor-based systems aim to resolve these issues by automatically and transparently accelerating demanding software kernels [1]. As a vast majority of such kernels operate on one or several input/output data arrays, often of unknown size at compile time, which may have random access patterns and data dependencies, it is important to focus on co-processors capable of performing memory access efficiently.

HW/SW partitioning approaches differ in terms of where the partitioning effort is applied. Typically, HW/SW partitioning applies high-level synthesis techniques to source code, e.g. analysis/modification, to exploit more powerful optimizations to generate HW components. We, however, address a low level approach, usually referred to as binary acceleration, which attempts to find (either online or offline) suitable candidate instruction traces for acceleration when mapped to reconfigurable co-processors [2–5].

Several approaches have considered the use of RPUs acting as co-processors and providing memory operations. Kim et al. use an RPU with local memories and address operation scheduling to reduce access conflicts to the available ports [6]. Data arrays can be distributed among memories to optimize accesses. This requires a code analysis step to determine access patterns. Other authors specifically focus on binary acceleration. Beck et al. propose an approach to transparently map at runtime basic blocks of instruction traces into an RPU [7]. There can be as many concurrent memory accesses as available memory ports. Data access patterns can be random and known only at runtime. Paek et al. propose an offline binary dissassembly step to generate RPU configurations [3]. Acceleration is considered for data-dominant loops with the number of iterations known at compile time. Sequential memory accesses are supported, and data are passed to/from the RPU via a shared memory mechanism.

As an extension of previous work [8], this paper proposes an architecture for transparent binary acceleration which allows for an RPU to have transparent access to a shared main memory. In our approach, kernels to be mapped to the RPU are automatically identified from program execution traces, and are used to generate a dedicated RPU supporting up to two concurrent memory accesses, and including the Functional Units (FUs) needed to exploit the operation-level parallelism of the kernels. This dedicated RPU is then used to accelerate program execution transparently at runtime.

This paper is organized as follows. An overview of the proposed architecture is presented in Section 2. Section 3 details the RPU architecture and the handling of memory operations. Section 4 describes the module for transparent access to data memory by the RPU. Section 5 explains the tool flow, the experimental setup, and presents experimental results. Finally, Section 6 concludes the paper.

2 General Architecture Overview

The architecture and tools described in this paper are extensions of the ones presented in [8] in order to provide RPU support for memory accesses. They support the same four-step dynamic partitioning approach: (1) Loops within computational kernels are identified from execution traces and represented as Megablocks [9]; (2) Selected Megablocks (repeating code patterns) are transformed into a RPU specification and corresponding configurations by a custom toolchain, resulting in a specialized reconfigurable accelerator; (3) during runtime, mapped Megablocks are identified at the start of their execution when the GPP reaches the Megablock code; (4) execution of the mapped Megablocks is migrated transparently to the RPU.

The present work addresses the lack of support for memory accesses by our previous RPUs. Fig. 1 shows the enhanced system architecture, which is composed of a program/data Block RAM (BRAM) with two ports, each connected to a Local Memory Bus (LMB), a GPP, an RPU connected to the GPP via a Processor Local Bus (PLB), two LMB Multiplexer modules, each connected to an LMB, and an LMB Injector module attached to the GPP's instruction bus.

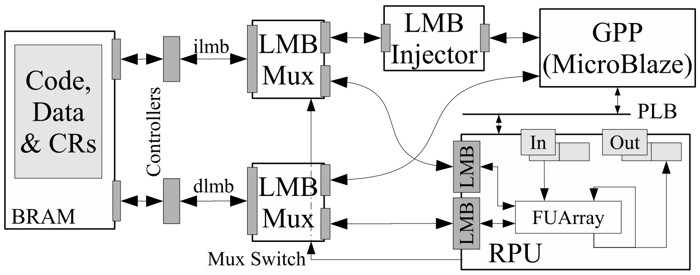


Fig. 1. Architecture overview. The RPU shares BRAM access with the GPP through the LMB Multiplexer.

As in the previous version of our architecture, the MicroBlaze executes unmodified program code from local memories. The LMB Injector is responsible for the migration step, which is accomplished by monitoring and modifying the contents of the instruction bus. If the start address of a region of code mapped to the RPU is detected, the Injector branches the GPP to a special subroutine that handles the communication between the GPP and the RPU.

The Communication Routine (CR) sends operands from the GPP's register file to the RPU through the peripheral bus, followed by a start signal. The RPU then gains control of the Local Memory Bus and accesses the BRAM by asserting the switch signals of the LMB Multiplexers. Each such module allows for two master devices to access a single-master LMB, sharing the entire address space of a BRAM without incurring any overhead, and without introducing data coherency issues. No memory address translation steps are necessary (cf. Sect. 4).

Once RPU execution ends, control of the LMBs is handed back to the GPP, which executes the remainder of the CR, recovering results to its register file and resuming software execution from the memory address where it was migrated.

The RPU can perform up to two simultaneous write/read accesses. Memory accesses can be random with addresses being calculated in the RPU. As multiple independent memory accesses occur in a wide range of Megablocks, access to both ports of the memory by the RPU allows for the exploration of this latent parallelism, with considerable potential speedups.

3 RPU Architecture

Fig. 2 shows the Reconfigurable Processing Unit (RPU) architecture (omitting the PLB interface). The RPU contains an array of Functional Units (FUs) tailored for a specific set of Megablocks [8]. Each row contains a number of FUs able to execute in parallel. The FU layout shown in Fig. 2 represents a synthetic example with 3 rows (details are omitted for clarity). Data are received from the preceding row and propagated to the next. Passthrough components are inserted to enable connections between FUs on non-adjacent rows. An iteration completes when all rows have computed their results, and data is fed back to the first row. FUs are reused between configurations, and the depth (no. of rows) of the RPU equals the longest Critical Path Length (CPL) of the dataflow graphs representing all the implemented Megablocks.

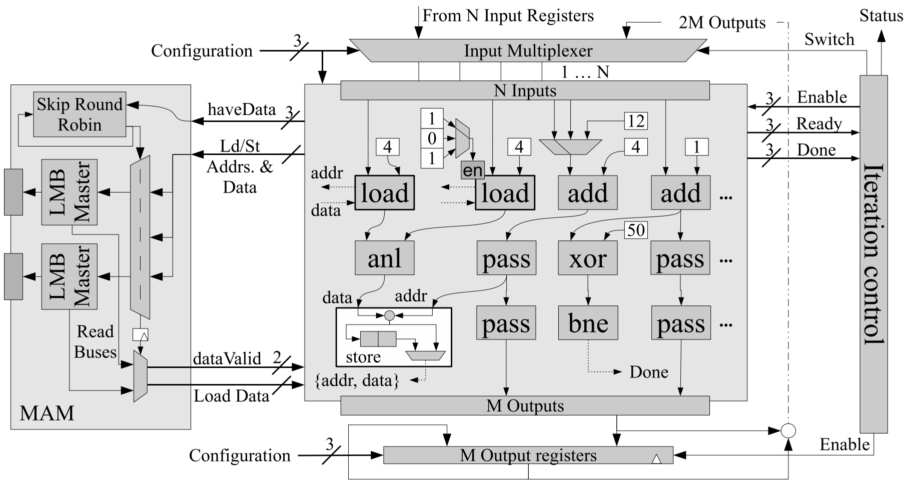


Fig. 2. Simplified diagram of the RPU's internal architecture, including the memory access handling mechanisms

The RPU executes the equivalent of single path instructions traces that cross control-flow boundaries, and so it has one entry point and several exit points. This is exemplified in Fig. 2 by the *bne* operation which triggers a *Done* signal. When any of these exit operations is triggered, the current iteration is discarded and software execution is resumed.

Memory operations are implemented by special FUs. These special FUs are not single-cycle and may have a variable latency. To support multi-cycle operations, each row of the array generates a ready signal which is asserted when all multi-cycle FUs on that row assert their individual ready signals. The *Iteration Control* module issues an *Enable* signal per row, and checks the status of that row's *Ready* signal immediately after issuing the enable. If the row is not ready, the control logic waits until it can issue the enable, thereby stalling the array while memory requests are handled Memory accesses are managed by the Memory Access Manager (MAM) shown on the left in Fig. 2. This module and details about the load/store FUs are presented in Section 3.2. The MAM receives data and addresses from all load/store FUs in the array. To determine the width of some ports, the number of memory operations the MAM can handle is specified at synthesis time. Actual memory accesses are performed via the RPU's two LMB ports, which interface with the LMBs through the LMB Multiplexers. As we currently use a dual-port BRAM, the number of ports in this implementation is limited to two.

3.1 Reconfiguration

Reconfiguration of the RPU is done by re-routing operands and by enabling or disabling FUs. To select a configuration before activation of the RPU, a configuration register is written to during CRs execution by the MicroBlaze.

The toolchain produces per-row HDL specifications of the connections between FUs of adjacent rows, thus minimizing the required resources to support all configurations. Each FU input is driven by a selector which is tailored at synthesis time to output one of a number of sources equal to the number of configurations. The Megablock extraction performs constant propagation, leading to some FU inputs remaining constant for all iterations during an execution. Instead of feeding the RPU with constant values at runtime, they are specified in the configuration. The inter-row selectors can either fetch values from any one of the outputs of the previous row or feed the input they drive with synthesis-time specified constants. These row interconnections are omitted for clarity from the example of Fig. 2, except for the one associated with the *add* FU in the first row, which exemplifies three configurations . In this case, for two configurations, the first input of the *add* is fed with a value from the N inputs available to the array (either values from the input registers or feedback values), and for the third configuration a constant value is supplied to the FU. There is one such selector per FU input. If only one configuration is present, the inter-row connections are optimized into wires.

Not all FUs in the RPU are actually used by each specific configuration. Although data may be fed to operations such as additions and other arithmetic even when their outputs are not used during execution, unused memory and exit FUs must be disabled. Thus, each FU is also driven by an enable signal. One of the *load* operations in Fig. 2 shows the enable signal being driven high by the selected configuration. The following excerpts from the RPU specification generated for the *chgBrghtB* benchmark illustrate how the inter-row selectors and enable signals are specified by tool-generated parameters at synthesis time.

```
parameter [0:32*(N_ROWS*N_COLS)-1]   parameter [0:(33*9*N_CONFIGS)-1]
FU_ENABLES = {                       ROW3_CONFIGS = {
 {32'b11,  (...)  },                  // Config. 1      //Config. 2
 {32'b11,  (...)  },                  {32'h0,    1'b0}, {32'h0, 1'b0},
 {32'b01,  (...)  },                  {32'hffff, 1'b1}, {32'h0, 1'b0},
 (...)                                (...)
 {32'b01,  (...)  }                   {32'h0,    1'b0}, {32'hff,1'b1}
}; // bit encoded enable signals     }; // configurations per FU input
```

The left excerpt shows the enable bits for the FUs of the first column. The first two FUs are used in both configurations, while the third only in the first. This scheme of configuration specification limits the number of possible configurations to 32. The right excerpt shows the specification for 3 selectors of one of the rows of the RPU. Each row of defines one selector and each column represents a configuration. A 32-bit parameter specifies either a constant value or output index of the previous row. A 33rd bit distinguishes between each case. In this example, the second selector feeds a constant (32'hFFFF) to its associated input in configuration 1 and the first output of the previous row in configuration 2.

3.2 Memory Access Management

The RPU supports up to two simultaneous memory accesses by using the LMB Mux to interface with the BRAM ports. The RPU supports read/write operations of bytes, half-words or 32-bit words by using the byte enable signals of the LMB. The byte size of a memory operation is specified at runtime by one of the inputs to the load/store FUs. Since the architecture is not restricted by different address spaces for RPU and GPP, and because the RPU may receive memory addresses as operands from the GPP at runtime, access to heap allocated data is also possible.

Execution of a memory operation on the FU array is decoupled from the memory access itself. That is, store and load FUs only issue memory access requests to the MAM. Thus, more than two memory accesses can be issued in the same clock cycle. The RPU may or may not stall execution until they are processed, depending on the combination of operations and the current state.

If the concurrent memory operations are stores, the values to be written out are either immediately sent to memory (if both ports are free), or are instead. Since stores produce no data for use in the FU array, they introduce no latency, and only stall the RPU if execution reaches their row before any buffered data has not been written. For instance, two store operations occurring every 3 cycles can be written to memory using only one memory port without stalling. Once execution on the RPU ends, no more store operations are issued and buffered data are flushed out to memory before the RPU releases the LMB Mux switches, and allows the MicroBlaze to continue execution. Since the last iteration must be discarded and executed in software, stores must not be scheduled before the first enabled exit operation to maintain coherency. So, we adopt an As Late as Possible (ALAP) scheduling scheme for stores. If dependent loads occur a sufficient number of clock cycles after the issuing of the corresponding stores, no problems occur. The number of cycles required varies with the number of stores to be performed, and with the availability of the LMB ports on the MAM when they are issued. The placement mechanism does not yet analyze RAW dependencies.

Load operations behave differently, because they have no slack (due to the way Megablocks are generated): the loaded value is always required by the following row. When a load operation is issued, the RPU stalls until it is handled. Two load operations in the same row can be handled simultaneously, introducing a latency of 1 clock cycle. If more are present, each one must be handled in order for execution to advance. There is currently no restriction on the number of allowed loads per row. The evaluated benchmarks have a maximum of 3 concurrent loads (*perimeter* benchmark).

The order in which the memory operations are treated is dictated by a selection logic performed by the MAM. Memory operations are treated in the same clock cycle they are issued, if a port is available. Each port is assigned a memory operation to handle in a round-robin fashion, which skips operations that do not have their request signal asserted in order to reduce latency. Both ports cannot choose the same operation simultaneously. For load operations, the selection logic directs the read data bus of a given port to the respective load FU.

Although this architecture allows for the minimum possible latency for both loads and stores, the selection logic can introduce critical path delays, if the FU array contains many memory operations. For the tested benchmarks *fft* and *perimeter*, which have 8 and 6 memory operations, respectively, the maximum operating frequency of each RPU is comparable to the delay introduced by using one hardware multiplier on our target FPGA.

4 The LMB Multiplexer

The LMB Multiplexer, shown in Fig. 3, is a peripheral with three LMB ports. Two ports connect to bus masters and a third connects to the actual Local Memory Bus. This allows for two masters to access a single LMB and its slave devices. The multiplexer is completely transparent. It does not add signals to the bus interfaces or clock cycles to data exchanges between the bus and a master. The transaction behavior on the bus is unaltered, and no modifications are required to either the bus or the master devices.

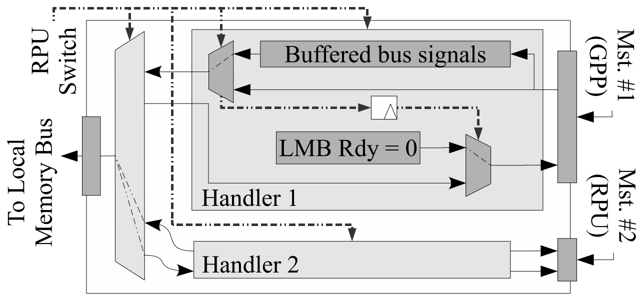


Fig. 3. Two port LMB Multiplexer. Each master device is treated by a handler.

Both ports of the LMB Multiplexer are bidirectional. The module uses the bus signals to perform synchronization and allow for a gracious handover of bus control between the two masters. When a switch is requested, it occurs immediately only if there is no unfinished bus transaction or if a response to the last transaction is already present on the bus lines (so as not to issue another request untimely). When switching, the outputs from the newly selected master are immediately connected to the bus; the bus response signals are only sent to the newly-selected master in the next clock cycle, so that the response to the previous transaction is sent to the previously selected master.

When a master is not selected, its requests are sent to a handler module which buffers up to one access request. The handler buffers all master downstream signals when an address strobe is asserted. This is sufficient, since the LMB interface is blocking. If a request is buffered, the handler module holds the LMB Ready signal low, which halts the master. When a handler has a buffered request and the master of that handler is reselected a one-cycle delay is added. The buffered request must be strobed to the bus first. The master reads back the bus response and is then ready to perform more transactions. Since the MicroBlaze does not timeout when attempting to access the LMB it can wait indefinitely. Switching between bus masters requires no additional handshaking. Each LMB Mux also includes the same address mask as the memory controller of the bus it interfaces with, ignoring any requests that do not match the address range.

5 Experimental Evaluation

The toolchain that supports the presented approach is an extension of the tools described in detail in [8]. The Megablock Extractor processes the executable files, simulates them, and uses the trace information to identify candidate Megablocks. These are passed to the RPU synthesis tool, which generates a parameterized HDL description of the RPU along with the configuration information. A second tool produces additional HDL specifications for the LMB Injector and CR code.

The CRs are added to the executable by being packed into arrays and compiled along with the benchmark, and then linked to predefined memory positions. The toolchain can produce CRs and HDL for a system in which the GPP/RPU communication is done through the PLB, or for a variant where modules are connected by low-overhead point-to-point Fast Simplex Links (FSLs). The results presented here were obtained with a PLB-based implementation.

5.1 Benchmark Results

The RPU's memory access mechanism was tested with 18 benchmarks selected from Texas Instrument's IMGLIB, from the SNU-RT Benchmark Suite and other assorted sources [10–12]. For most of these benchmarks, only one Megablock was implemented. Three additional synthetic benchmarks were written to produce RPUs implementing several Megablocks, for the sake of validation. The resulting RPUs have at least one memory operation.

The test bed was a Digilent Atlys board with a Xilinx Spartan 6 LX45 FPGA. Xilinx EDK 12.3 was used for synthesis and bitstream generation, the system clock was set to 66 MHz, and the MicroBlaze processor was synthesized for minimum instruction latency. Benchmarks were compiled by *mb-gcc 4.1.2* with the -O2 flag.

The chosen kernels operate on data arrays of various sizes. For the *SNU-crc* benchmark there are two load operations, one which performs a load of a half-word and another, a byte addressed load. The RPU often receives operands which are memory positions of data arrays, and the addresses for data accesses are computed during execution. The number of concurrent memory accesses allows for a good use of the RPU's memory ports, as most generated RPU configurations have at most two simultaneous loads/stores. Synthetic benchmarks *synt1*, *synt2* and *synt3* are sequences of calls to routines belonging to other benchmarks. Benchmarks *synt1* and *synt2* have 6 configurations, and *synt3* has 3. To evaluate the worst case scenario, the benchmarks were written so that each set of kernels is called 500 times with an RPU reconfiguration on every call.

Table 1 summarizes the characteristics of the generated RPUs, the Instructions per Clock (IPC) achieved by hardware and the software IPC, for comparison. The *#Lds/Sts* column contains the number of load/store FUs in the RPU (not necessarily concurrent). The *#Ops* and *#Passes* columns specify how many FUs are actual operations (loads/stores included) and how many are passthroughs. The *#Rows* column refers to the number of rows of the RPU.

The *Hw. IPC* is the ratio between the number of enabled FUs (excluding passthrough components) and the number of cycles required to execute all of the rows of the RPU (i.e. one loop iteration). The number of cycles required does not equal the number of rows due to the multi-cycle memory operations (whose latency depends on their concurrency). For the tested benchmarks (excluding synthetic examples), memory operations introduce an average latency of 2.33 clock cycles. For cases with more than one configuration, the *Sw. IPC* was computed as the average of the IPCs for all corresponding Megablocks. The *Hw. IPC* was computed from the average number of enabled FUs per configuration and the average number of clock cycles required to complete an iteration.

The number of execution clock cycles was measured using a custom timer peripheral. The following measurements were made: 1) number of cycles during which the RPU is stalled; 2) number of cycles spent executing operations on the RPU (stall cycles included); 3) number of cycles required to execute the mapped Megablocks (in hardware or software) including communication and other overheads introduced by the migration mechanism; and 4) number of the cycles spent executing the entire benchmark, again including all overheads. Measurements for both hardware and software execution can be taken from the same implementation as the migration step can be easily disabled.

The *Spd.* column refers to the overall benchmark speedup, computed from 4). Overhead introduced by the execution of the CRs (i.e. GPP-RPU communication over the PLB) was derived from 2) and 3). The last column contains the potential overall speedup were this overhead completely removed. For the chosen benchmarks, the overhead accounts for an average of 32.9 % of the time required for migration and RPU execution. Each call of the RPU takes an average of 193 clock cycles. Even so, the average speedup achieved for the Megablocks alone is 1.52×. The overall speedup (with overhead included) is 1.41×. In some cases the overhead imposed an overall slowdown, although the mapped Megablocks were

Table 1. RPU Characteristics and achieved Speedups

Benchmark	#Lds/Sts	#Ops	#Passes	#Rows	Hw.IPC	Sw.IPC	Spd.	Spd.(no OH)
blit1	1/1	10	17	3	2.50	0.92	1.45	1.47
chgBrght1	1/1	11	31	7	1.38	0.92	0.97	1.20
chgBrght2	1/1	11	20	5	1.83	0.91	0.54	1.80
quantize	1/1	11	35	6	1.57	0.92	1.90	2.14
SNU_crc	2/0	16	29	9	1.45	0.92	1.01	1.02
blit[1*]	1/2	14	27	4	2.10	0.92	2.38	2.48
boundary[1]	1/2	12	18	3	3.00	0.93	1.17	3.73
dotprod[1]	2/0	9	11	4	1.80	0.88	1.77	1.80
fir2[1]	2/1	12	17	4	2.00	0.91	1.40	1.78
perimeter[1]	5/1	19	12	3	3.17	0.94	1.82	2.51
bob_hash[2]	1/0	11	24	8	1.22	0.91	1.53	1.55
chgBrghtB[2*]	1/2	16	31	7	1.38	0.92	0.47	2.22
fft[2]	4/4	30	78	7	3.00	0.96	0.52	2.27
motEstim[2]	2/1	13	48	7	1.63	0.93	0.54	1.75
sad_8x8 [2]	2/0	14	39	8	1.56	0.92	0.52	1.73
checkbits[3]	1/1	64	169	16	3.65	0.98	3.56	3.94
compositing[3]	2/1	18	78	10	1.64	0.95	2.09	2.27
pop_array1[3]	1/0	22	94	15	1.38	0.96	1.86	2.11
synt1	6/3	36	27	4	2.36	0.91	2.09	2.52
synt2	6/5	53	88	8	1.79	0.92	0.68	1.64
synt3	4/3	93	180	16	2.00	0.97	1.85	1.97

[1]Included in *synt1* [2]Included in *synt2*; [3]Included in *synt3*.
[*]Benchmark has two Megablocks.

accelerated. By eliminating this overhead (for instance, using a FSL between the GPP and the RPU) the average potential speedup is 2.10×, and a speedup occurs for all cases. These averages do not include synthetic benchmarks.

5.2 Discussion

The benchmarks for which the *Hw. IPC* is largest are the ones with the greatest potential speedup. Memory operations reduce the IPC because of the latency they introduce, which accounts for stall cycles. Stalls only occur when more than 2 simultaneous memory operations are issued. Even then, this depends on the type of memory operation, since stores can be buffered and handled in later cycles. Therefore, latency above the minimum possible memory access time is only introduced when a load operation cannot be handled in the cycle it is issued. This occurs for the *perimeter* benchmark, but its execution still achieves the third best speedup (*w/o* overhead) of all the tested benchmarks. The slowdowns that occur are due to the small number of iterations performed in hardware, which do not make up for the communication overhead.

Stall cycles account for an average 19.5 % of the total number of cycles spent on the RPU, excluding synthetic benchmarks. The largest stall time (57.1 %

of the total execution time) occurs for the *perimeter* benchmark due to two consecutive rows with 3 concurrent accesses each. Execution of the *checkbits* benchmark stalls only 5.91 % of the time, because it has only one load operation. Although the *compositing* benchmark contains two loads and one store, only the two loads occur in the same row, introducing only the minimum latency of one clock cycle and resulting in the third lowest stall time of 9.12 %.

Regarding resources, the generated RPUs use on average 6.3 % of the available 27288 Lookup Tables (LUTs) and 4.3 % of 54576 Flip Flops (FFs). Maximum utilization for these resources is, respectively, 15.1 % (*pop_array1* benchmark) and 8.5 % (*fft* benchmark), while the minimum values are 2.1 % and 1.8 % (both for the *dotprod* benchmark). The average synthesis frequency of the individual RPU was 114 MHz. The lowest frequency, 62 MHz, occurs for the *fft* RPU. The associated critical path delay is due to the MAM selection logic. The highest frequency is 170 MHz for, *chgBrght1*. Although the frequency of all RPUs, save for *fft*, exceeds the base 66 MHz system clock used for most benchmarks, the clock frequency had to be lowered to 50 MHz (for the *perimeter*, *fft* benchmarks) and *synt3* benchmarks) or 33 MHz (for benchmarks *synt1* and *synt2*).

Some of the FUs are reused between configurations. Specifically, a total of 36, 37 and 10 FUs were reused for *synt1*, *synt2* and *synt3*, respectively. This does not account for pass-through components, of which 96, 170 and 216 were reused between configurations for the 3 synthetic benchmarks. With respect to the total FPGA resources, the LUT usage for the three synthetic benchmark RPUs is 25 %, 40 % and 49 %; FF usage is 6 %, 12 % and 17 %.

Benchmarks *synt1* and *synt3* achieve considerable speedups, incurring communication overheads of 17.1 % and 5.7 %. Benchmark *synt2* exhibits a slowdown as expected, since the same occurs for the individual implementations. The overhead for this case is 59.4 %. In the overhead-free scenario, the three synthetic cases show good speedups. These benchmarks show that a good average speedup can be achieved when Megablocks have similar CPLs and significant parallelism.

6 Conclusion

This paper presented a general-purpose computing architecture based on a General Purpose Processor (GPP) and a Reconfigurable Processing Unit (RPU) automatically generated offline from instruction traces. In this architecture, a multiplexer module transparently interfaces with standard memory buses, allowing the RPUs to access the GPP's data memory. We use a two-port data memory to allow parallel memory accesses and to achieve the acceleration of instruction traces with load/store operations. Data memory accesses are easily handled by our RPU through the transparent bus multiplexer, allowing shared access to the entire address space. The RPU can thus operate on any number of data arrays at any address regardless of their size. This allows an efficient memory access scheme that does not introduce costly data transfers between the data memory and the RPU. Future work will focus on extensions to the RPU execution model to enable both pipelining and multipath Megablock execution, and on developing efficient scheduling memory operations in the RPU, including the handling of RAW and WAR dependencies.

Acknowledgments. This work was funded by the European Regional Development Fund through the COMPETE Programme (Operational Programme for Competitiveness) and by national funds from the FCT–Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-022701, and by the European Community's Framework Programme 7 under contract No. 248976.

References

1. Wolf, W.: A decade of hardware/software codesign. Computer 36, 38–43 (2003)
2. Clark, N., Blome, J., Chu, M., Mahlke, S., Biles, S., Flautner, K.: An architecture framework for transparent instruction set customization in embedded processors. In: Proc. of the 32nd Annual Intl. Symposium on Computer Arch. (ISCA 2005), pp. 272–283. IEEE Computer Society, Washington, DC (May 2005)
3. Paek, J.K., Choi, K., Lee, J.: Binary acceleration using coarse-grained reconfigurable architecture. SIGARCH Comput. Archit. News 38(4), 33–39 (2011)
4. Lysecky, R.L., Vahid, F.: Design and implementation of a microblaze-based warp processor. ACM Trans. Embedded Comput. Syst. 8(3), 22:1–22:22 (2009)
5. Noori, H., Mehdipour, F., Murakami, K., Inoue, K., Saheb Zamani, M.: An architecture framework for an adaptive extensible processor. J. Supercomput. 45(3), 313–340 (2008)
6. Kim, Y., Lee, J., Shrivastava, A., Paek, Y.: Memory access optimization in compilation for coarse-grained reconfigurable architectures. ACM Trans. Des. Autom. Electron. Syst. 16(4), 42:1–42:27 (2011)
7. Beck, A.C.S., Rutzig, M.B., Gaydadjiev, G., Carro, L.: Transparent reconfigurable acceleration for heterogeneous embedded applications. In: Proc. of the Conf. on Design, Automation and Test in Europe (DATE 2008), pp. 1208–1213. ACM (2008)
8. Bispo, J., Paulino, N., Cardoso, J.M., Ferreira, J.C.: Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units. International Journal of Reconfigurable Computing (2012) (in press)
9. Bispo, J., Cardoso, J.M.P.: On identifying and optimizing instruction sequences for dynamic compilation. In: Proc. Intl. Conf. Field-Programmable Technology (FPT 2010), pp. 437–440 (2010)
10. Seoul National University: SNU Real-Time Benchmarks, `http://www.cprover.org/goto-cc/examples/snu.html` (accessed December 23, 2012)
11. Texas Instruments: TMS320C6000 Image Library (IMGLIB) - SPRC264, `http://www.ti.com/tool/sprc264` (accessed December 23, 2012)
12. Warren, H.S.: Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)

Parametric Optimization of Reconfigurable Designs Using Machine Learning

Maciej Kurek, Tobias Becker, and Wayne Luk

Department of Computing, Imperial College London

Abstract. This paper presents a novel technique that uses meta-heuristics and machine learning to automate the optimization of design parameters for reconfigurable designs. Traditionally, such an optimization involves manual application analysis as well as model and parameter space exploration tool creation. We develop a Machine Learning Optimizer (MLO) to automate this process. From a number of benchmark executions, we automatically derive the characteristics of the parameter space and create a surrogate fitness function through regression and classification. Based on this surrogate model, design parameters are optimized with meta-heuristics. We evaluate our approach using two case studies, showing that the number of benchmark evaluations can be reduced by up to 85% compared to previously performed manual optimization.

Keywords: optimization, surrogate modeling, PSO, GP, SVM, FPGA.

1 Introduction

Field programmable gate arrays (FPGAs) allow designs that are customized to the requirement of the application. Reconfiguration is an additional benefit which allows the designer to modify designs at run time, potentially increasing performance and efficiency. Unfortunately, the optimization of reconfigurable designs often requires substantial effort from designers who have to analyze the application, create models and benchmarks and subsequently use them to optimize the design. This process often involves adjusting multiple design parameters such as numerical precision, degree of pipelining or number of cores. One could proceed with automated optimization based on an exhaustive search through design parameters which are derived from application benchmarks; however, this is unrealistic since benchmark evaluations involve bitstream generation and code execution which often takes hours of compute time.

It has been shown useful to construct surrogate models of fitness functions representing design quality for computationally expensive optimization problems in various fields [1–5]. As these models are orders of magnitude faster to evaluate than the actual fitness function, they can substantially accelerate optimization thus allowing for an automated approach. This is the motivation behind our development of the MLO tool which we apply to the problem of reconfigurable designs parameter optimization. In [6] we present initial concepts on optimizing parameter configuration of reconfigurable designs using surrogate models.

We now present the formalism and experimental evaluation for our approach. The contributions of this paper are:

- A mathematical characterization of the parameter space for reconfigurable designs as well as a definition of a fitness function based on application benchmarks (Section 3).

  Generic surrogate models to approximate the fitness function using regression and classification. We combine surrogate models with meta-heuristics to provide a new MLO algorithm for automated optimization of reconfigurable designs (Section 4).
- An evaluation of our MLO approach in two case studies: (a) execution time of a run-time reconfigurable software-defined radio with varied degree of parallelism, and (b) and a previously used [6] throughput of a quadrature based financial application with varied precision (Section 5).

2 Background

When developing reconfigurable applications, designers are often confronted with a very large parameter space. As a result the parameter space exploration can take an immense amount of time. A number of researchers approach the problem of high-cost fitness functions and large design spaces in various fields [1–5] by having fitness functions combined with fast-to-compute surrogate models provided by a Gaussian Process (GP) for decreasing evaluation time. However most current surrogate models only consist of a regressor and do not take into account possible invalid configurations within the design space. Furthermore, all of them are evaluated using artificial benchmarks spanned by continuous $\mathbb{R}^n$ spaces, while parameter spaces for reconfigurable applications usually also involve discrete dimensions (e.g. number of cores). Surrogate models approximating fitness functions by substituting lengthy evaluations with estimations based on closeness in a design space have been investigated in reconfigurable computing [7]. The work covers surrogate models for circuit synthesis from higher level languages (HLL), rather than parameter optimization.

GP is a machine learning technology based on strict theoretical fundamentals and Bayesian theory [8, 9]. GP does not require a predefined structure, can approximate arbitrary function landscapes including discontinuities, and includes a theoretical framework for obtaining the optimum hyper-parameters [4]. An advantage of GP is that it provides a predictive distribution, not a point estimate.

A Gaussian process is a collection of random variables, any finite set of which have a joint Gaussian distribution. A Gaussian process is completely specified by its mean function $m(\mathbf{x})$ and the covariance (kernel) function $k(\mathbf{x}, \mathbf{x}')$:

$$\hat{f}(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{1}$$

The $k(\mathbf{x}, \mathbf{x}')$ expresses the covariance between pairs of random variables, and in regression analysis it expresses the relation between input-output pairs. This is based on a training set $\mathcal{D}$ of n observations, $\mathcal{D} = (\mathbf{x}_i, y_i)|i = 1, ...n$, where $\mathbf{x}$ denotes an input vector, y denotes a scalar output. The column vector inputs for all n cases are aggregated in the $D \times n$ design matrix X, and the outputs are collected in the vector $\mathbf{y}$. The goal of Bayesian forecasting is to compute the distribution $p(\hat{f}|\mathbf{x}_*, \mathbf{y}, X)$ of the function $\hat{f}$ at unseen input $\mathbf{x}_*$ given a set of training points $\mathcal{D}$. Using Bayes rule, the predictive posterior for the Gaussian process $\hat{f}$ and the predicted scalar outputs $\hat{f}(\mathbf{x}_*) = y_*$ can be obtained.

Support Vector Machine (SVM) is a maximum margin classifier, which constructs a hyperplane used for classification (or regression) [10]. SVMs use kernel functions $k(\mathbf{x}, \mathbf{x}')$ to transform the original feature space to a different space where a linear model is used for classification. SVMs are a class of decision machines and so do not provide posterior probabilities. There is a training set $\mathcal{D}$ of n observations, $\mathcal{D} = (\mathbf{x}_i, t_i)|i = 1, ...n$, where $\mathbf{x}$ denotes an input vector, t denotes a target value. The column vector inputs for all n cases are aggregated in the $D \times n$ design matrix X, and the targets in the vector $\mathbf{t}$. The goal is to classify an unseen input $\mathbf{x}_*$ based on X and $\mathbf{t}$ by computing a decision boundary.

Particle Swarm Optimization (PSO) is a population-based meta-heuristic based on the simulation of the social behavior of birds within a flock [11]. The algorithm starts by randomly initializing N particles where each individual is a point in the $\mathcal{X} = \mathbb{R} \times ... \times \mathbb{R}$ search space. The population is updated in an iterative manner where each particle is displaced based on its velocity v_{id}. The criteria for termination of the PSO algorithm can vary, and usually are determined by a time budget. The x_{id} represents the dth coordinate of particle i from the set X_* of N particles, where particle is a point within $\mathcal{X}$. In the most basic form of PSO Eq. 2-3 govern movement of particles. $r_1 \sim U(0, 1)$ and $r_2 \sim U(0, 1)$ are two independent uniformly distributed random numbers, c_1 and c_2 are acceleration coefficients and p_{gd} and p_{id} are dth coordinates of the global best and personal best positions. p_{gd} is updated when a new global best fitness is found and p_{id} is updated when a particle improves over its best fitness.

$$v_{id} = v_{id} + c_1 r_1 (p_{id} - x_{id}) + c_2 r_2 (p_{gd} - x_{id}) \tag{2}$$

$$x_{id} = x_{id} + v_{id} \tag{3}$$

3 Optimization Approach

Traditionally, optimization of reconfigurable applications is carried out by building benchmarks and relevant tools, and the associated analytical models [12, 13]. This involves the following steps:

1. Build application and a benchmark returning design quality metrics.
2. Specify search space boundaries and optimization goal.
3. Create analytical models for the design.

4. Create tools to explore the parameter space.
5. Use the tools to find optimal configurations, guided by the models in step 3.
6. If result is not satisfactory, redesign.

In our approach the user supplies a benchmark along with constraints and goals, and the MLO automatically carries out the optimization (Algorithm 1). Our approach consists of the following steps:

1. Build application and benchmark returning design quality metrics.
2. Specify search space boundaries and optimization goal.
3. Automatically optimize design with MLO.
4. If result is not satisfactory, redesign or revised time budget and search space.

Our idea of surrogate modeling is illustrated in Fig. 1. The MLO algorithm explores the parameter space by evaluating different benchmark configurations as presented in the left figure. The results obtained during evaluations are used to build a surrogate model which provides a regression of the fitness function and identifies invalid regions of the parameter space. A meta-heuristic (currently PSO) guides the exploration of the parameter space using the surrogate model.

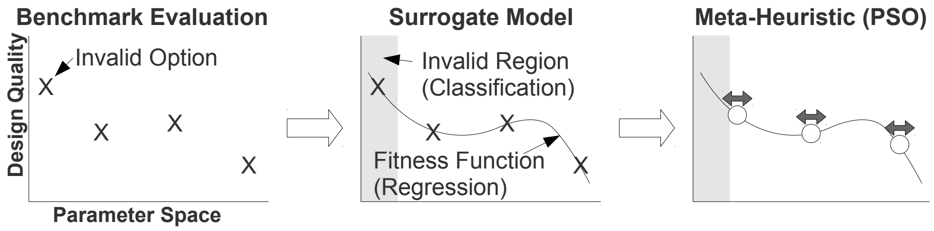


Fig. 1. Benchmark evaluations, surrogate model and model guided search

3.1 Parameter Space

The parameter space $\mathcal{X}$ of a reconfigurable design is spanned by discrete and continuous parameters determining both the architecture and physical settings of FPGA designs. A vector $\mathbf{x}$ represents a parameter configuration within the parameter space $\mathcal{X} = \mathcal{X}_1 \times ... \times \mathcal{X}_D$ such that any $\mathcal{X}_d \subseteq \mathbb{R}$. If $\mathcal{X}_d \subseteq \mathbb{Z}$, its discretization level is independent of other dimensions. $\mathcal{X}_d$ can be bounded with upper and lower limits U_d, L_d such that for all x_d, $L_d \leq x_{id} \leq U_d$. An example of a continuous parameter is core frequency and an example of a discrete parameter is the number of compute cores. For all discrete dimensions the step size, which we define as smallest distance between any two x_{id}'s, can vary. We might only be able to increase memory width in 16 bits increments.

A discrete parameter space has important implications on the PSO algorithm, as the equations governing movements of particles Eq. 2-3 are defined for a continuous $\mathbb{R}^n$ space. In Eq. 2 both r_1 and r_2 are random real numbers, which means that the resulting velocity component used to update position $\mathbf{x}$ cannot be used if x_d is discrete. To discretize the position value of a particle after its movement, we round its value and randomize its rounding error (dithering) presented in Eq. 4. By using dithering instead of truncation PSO particles maintain their velocity component which results in a more thorough exploration.

$$x_{di} = \begin{cases} \lfloor x_{id} \rfloor & U(0, stepsize) < (x_{di} \mod stepsize\ ) \\ \lceil x_{id} \rceil & U(0, stepsize) \geq (x_{di} \mod stepsize\ ) \end{cases} \tag{4}$$

3.2 Fitness Function

Given a parameter setting $\mathbf{x}$, the benchmark $b(\mathbf{x})$ returns a fitness metric which constitutes two values: y, the scalar metric of fitness and t, the exit code of the application. Execution time and power consumption are examples of fitness measures. There are be many possible exit codes t, with 0 indicating valid $\mathbf{x}$'s. The designer can choose to extend the benchmark to return additional exit codes depending on the failure cause, such as configurations producing inaccurate results or failing to build.

We distinguish three different types of exit codes. The first type is exit code 0 indicating a valid design. The second type of exit codes indicate configurations that produce results yet fail at least one constraint making them undesirable. The third type of exit codes is used for configurations that fail to produce any results. The region of $\mathcal{X}$ that defines configurations $\mathbf{x}$ that produce y and satisfy all constraints is defined as valid region $\mathcal{V}$, regions with designs failing at least one constraint yet producing y are part of failed region $\mathcal{F}$, and the region with designs failing to produce y is the invalid region $\mathcal{I}$. If $\mathbf{x}_*$ does not produce a valid result, we assign a value that the designer assumes to be the most disadvantageous. Depending on whether we face a minimization/maximization problem,s either a high max_{val} or low min_{val} value will be assigned.

$$f(\mathbf{x}) = \begin{cases} y & \mathbf{x} \in \mathcal{V} \\ max_{val} \vee min_{val} & otherwise \end{cases} \tag{5}$$

4 MLO Surrogate Model

We integrate a Bayesian regressor $\hat{f}$ and a classifier to create a novel surrogate model for a given fitness function f. As illustrated in Fig 1, the problem we face is regression of f over $\mathcal{V}$ and $\mathcal{F}$ as well as classification of $\mathcal{X}$. We make use of Bayesian regressors to access the probability of prediction of $\hat{f}(\mathbf{x}_*)$ of non-examined parameter configurations $\mathbf{x}_*$. We use classifiers to predict exit codes of X_* across $\mathcal{X}$. Regressions are made using the training set obtained from benchmark execution $\mathcal{D}_r$, while classification is done using the training set $\mathcal{D}_c$. We invoke $regressor(\mathcal{D}_r, \mathbf{x}_*)$ for every particle in $\mathbf{x}_*$ to obtain the regression y_* and its probability $p(y_*|\mathbf{x}_*, \mathcal{D}_r)$, which we denote as ρ for simplicity. Class label t_* of particle $\mathbf{x}_*$ is predicted by the classifier $classifier(\mathcal{D}_c, \mathbf{x}_*)$.

Algorithm 1. MLO

```
1: for $\mathbf{x}_* \in X_*$ do
2:     $\mathbf{x}_*.fit \leftarrow f(\mathbf{x}_*)$        ▷ Initialize with a uniformly randomized set $X_*$.
3: end for
4: repeat
5:     for $\mathbf{x}_* \in X_*$ do
6:         $y_*, \rho \leftarrow regressor(\mathcal{D}_r, \mathbf{x}_*)$
7:         $t_* \leftarrow classifier(\mathcal{D}_c, \mathbf{x}_*)$
8:         if $\rho < min_\rho$ and $t_* = 0$ then
9:             $\mathbf{x}_*.fit \leftarrow y_*$
10:        else
11:            if $t_* = 0$ then
12:                $\mathbf{x}_*.fit \leftarrow f(\mathbf{x}_*)$
13:            else
14:                $\mathbf{x}_*.fit \leftarrow \max_{val}$ or $\min_{val}$
15:            end if
16:        end if
17:    end for
18:    $X_* \leftarrow Meta(X_*)$        ▷ Iteration of the meta-heuristic
19: until Termination Criteria Satisfied
```

We present our MLO in Algorithm 1. The algorithm's main novelty with respect to surrogate-based algorithms is the integration of a classifier to account for invalid regions of $\mathcal{X}$. We initialize the meta-heuristic of our choice with N particles X_* uniformly randomly scattered across $\mathcal{X}$. Each particle has an associated fitness $\mathbf{x}.fit$ and a position $\mathbf{x}$. For all $\mathbf{x}_*$ predicted to lie in $\mathcal{V}$ we proceed as follows. Whenever ρ returned by the regressor is smaller than the minimum required confidence min_ρ we use the y_*; otherwise we assume the prediction to be inaccurate and evaluate $f(\mathbf{x}_*)$. The meta-heuristic will avoid $\mathcal{I}$ and $\mathcal{F}$ regions as they are both assigned unfavorable max_{val} or min_{val} values. We construct the training sets $\mathcal{D}_c$ and $\mathcal{D}_r$ as described in Algorithm 2. Whenever $b(\mathbf{x}_*)$ is evaluated, $(\mathbf{x}_*, t_*)$ is included within the classifier training set $\mathcal{D}_c$. If exit code is valid $(t_* = 0)$, then $(\mathbf{x}_*, y_*)$ is added to $\mathcal{D}_r$.

Although the MLO will converge towards an optimum, it is limited by heuristic search restrictions and as such it cannot guarantee to find the global optimum. Hence, it is crucial to specify the termination criteria. Determining MLO termination criteria is based on the optimization scenario and we present three possibilities where the user:

1. Has a limited compute time budget.
2. Requires only certain design quality.
3. Needs maximum performance, with a large budget.

A user can have a limited compute time budget when optimizing an application and the MLO can terminate once the budget has been reached. For example, we could allocate a number of machines for a 24 hour period. Alternatively, if the user only requires a certain performance, the MLO can be run until a configuration x is found that meets the required performance, and the optimization can be terminated. Lastly, if the MLO is used to maximize performance without a limited compute time budget, the MLO will terminate when the best found solution does not improve during a pre-defined amount of time.

Algorithm 2. $f(\mathbf{x})$

```
1: $t, y \leftarrow b(\mathbf{x})$
2: $\mathcal{D}_c \leftarrow (\mathbf{x}, t)$                ▷ Update the classifier's training set
3: if $t \in \mathcal{F}$ or $t \in \mathcal{V}$ then
4:     $\mathcal{D}_r \leftarrow (\mathbf{x}, y)$            ▷ Update the regressor's training set
5: end if
6: if $t \in \mathcal{V}$ then
7:     return $y$
8: else
9:     return $\max_{val}$ or $\min_{val}$
10: end if
```

5 Evaluation

We use our approach to optimize two designs which are previously optimized with custom analytical models. The first application is a run-time reconfigurable software-defined radio with variable degree of parallelism [13]. The second is a quadrature-based financial application with variable precision [12], for which we show how known analytical models can be used to reduce the number of dimensions that need to be explored. We use GPs utilizing an anisotropic exponential kernel with additive Gaussian noise. We choose SVMs as our classifier with a Radial Basis Function (RBF) kernel. Due to its simplicity and effectiveness we use a velocity clamping version of PSO with c_1 and c_2 set to 2.0. All presented results are averaged over 20 trials. To evaluate the MLO performance in our three scenarios, we terminate when the global optimum is reached. We determine the globally optimal configuration with analytic methods, run the MLO to achieve the same value and then compare the complexity of both approaches.

As shown in Fig. 2 we create a surrogate model of the fitness function. We also classify the design space using SVM as shown in the right figure. We see regions of $\mathcal{X}$ with colour distinguishing different exit codes; dark gray for valid and light gray for inaccurate designs. Black x marks drawn over the design space represent configurations $\mathbf{x}$ which have been evaluated and used to build the surrogate model. The design space includes white circles which represent positions of the particles of the PSO algorithm during the iteration when the image was created.

5.1 Reconfigurable Software-Defined Radio

We construct a benchmark based on studies conducted in [13]. The designer faces a trade-off between reconfiguration time and number of processing elements p.

Larger values of p correspond to designs with higher compute throughput; however, the chip takes longer to reconfigure and our aim is to find the optimal value of p. The reconfigurable radio can run at two different chip reconfiguration bandwidths Φ_{config} of 5 MB/s or 300 MB/s.

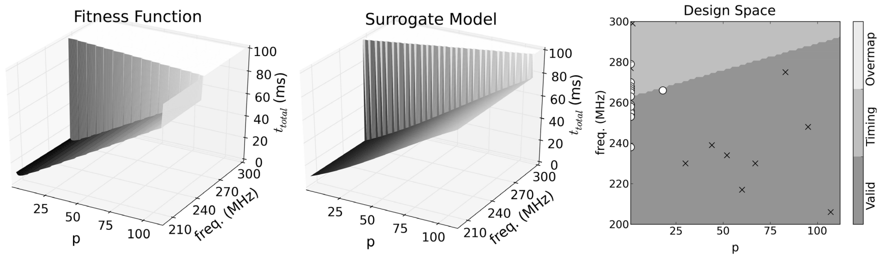


Fig. 2. Reconfigurable radio f ($\Phi_{config} = 5$ MB/s) and its surrogate model

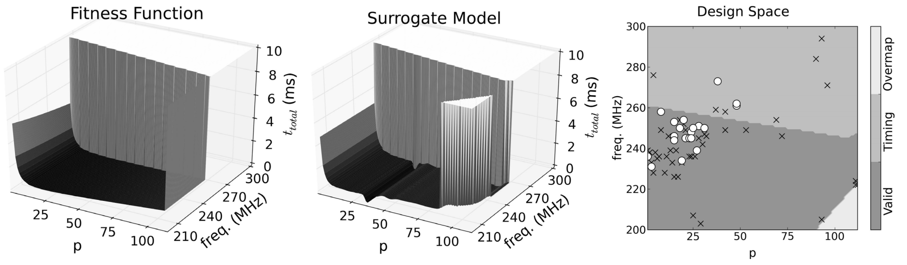


Fig. 3. Reconfigurable radio f ($\Phi_{config} = 300$ MB/s) and its surrogate model

Our parameters are p, Φ_{config} and core frequency $freq$. We define the design space as $\mathcal{X} = \{1-112\} \times \{5, 300\} \times \{freq_{min}-300\}$. We change $\mathcal{X}$ by varying the minimum frequency $freq_{min}$. Although the problem is three-dimensional, due to low discretization level of Φ_{config} we treat it as two separate two-dimensional optimizations. For the $\mathcal{I}$ region constituting timing and FPGA resource overmapping regions, we mark the execution time as undesirable. MLO terminates when $\mathbf{x}$ is evaluated within 2 MHz range of globally optimal solution.

In Fig. 2 we see how the SVM classifies a fraction of the parameter space as $\mathcal{V}$ and how the surrogate model closely matches the fitness function. We also see how particles collapse and explore the optimal region $p \approx 4$ for $\Phi_{config} = 5$ MB/s. In Fig. 3 we observe a similar situation but for $\Phi_{config} = 300$ MB/s with the optimal region $p \approx 20$. Again, the surrogate model resembles the fitness function. The collapse of particles is equivalent to the fine-tuning of the design parameters. We present a visualization of the optimization in Fig. 4, each pair of figures representing subsequent iterations. The top figures show the surrogate model,

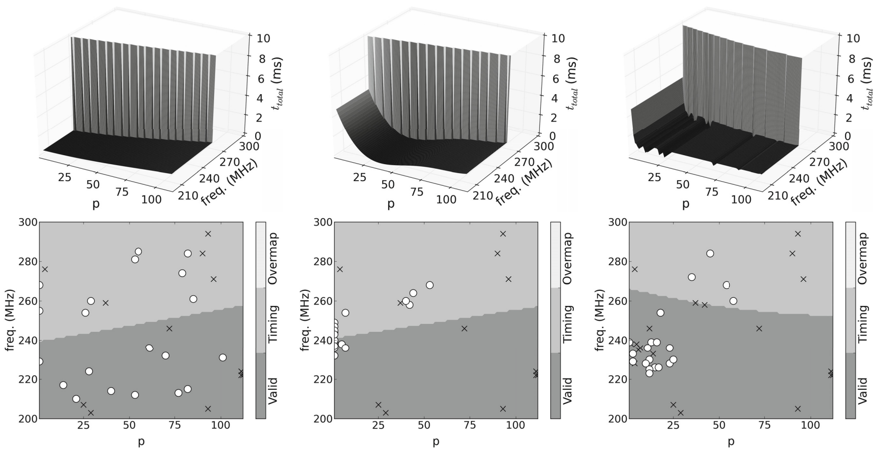


Fig. 4. Optimization of f ($\Phi_{config} = 300\,\text{MB/s}$) after 13, 14 and 22 f evaluations

while the bottom figures represent corresponding visualization of the design space and its classification. During several initial iterations and f evaluations, the particles (shown as white circles) are misled by the surrogate model to explore $p \approx 4$ region. In the last figure we see particles guided by an improved surrogate model moving towards the optimum $p \approx 20$ region.

We use the reconfigurable radio benchmark to determine the impact of design space size on the convergence of the MLO algorithm. In Tab. 1 we see a tendency of the number of f evaluations to decrease along with the design space size. We trim design space by increasing the lower limit of admissible frequency $freq_{min}$. This shows that the designer should select a small parameter range as small design space improves MLO convergence. One outlier of $f = 54$ in the case of Φ_{config}=300 MB/s and $freq_{min}$=200 MHz can be explained with the overall small sample size. Manual optimization is replaced by MLO, which works with nearly no manual input but for the initial design space specification.

Table 1. Average number of f evaluations - Reconfigurable radio optimization

Φ_{config}	$freq_{min}$	150 MHz	200 MHz	220 MHz
5 MB/s		44	37	31
300 MB/s		47	54	45

5.2 Quadrature Method-Based Application

In [12] the designer explores trade-off between accuracy and throughput in an application with three parameters. The first two parameters are mantissa width m_w of the floating point operators and the number of computational cores $cores$. Larger number of m_w bits increases computation accuracy, but limits the maximum number of *cores* that can be implemented on the chip due to the increased size of the individual core. The third parameter is the density factor d_f which specifies the density of quadratures used for integral estimation. It is a software parameter and is independent of the generated bitstream. Density factor d_f increases computation time per integration while improving the accuracy of the results due to finer estimation of the integral.

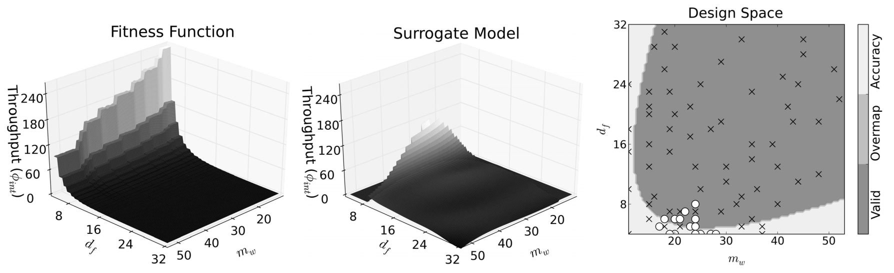


Fig. 5. Quadrature-based application f and its surrogate model

The optimization goal is to find the design offering the highest throughput given a required minimum accuracy defined in terms of root mean square error ϵ_{rms}. The error is defined with respect to results obtained by calculating a set of reference integrals at the highest possible precision. The MLO terminates when the globally optimal configuration for a given ϵ_{rms} is found. The $\mathcal{F}$ region contains the inaccurate result class, as these benchmark evaluations can be reused for regression. The design space $\mathcal{X}$ is defined as $m_w \times cores \times d_f$: $\{11-53\} \times \{1-16\} \times \{4-32\}$. We can explore the whole $\mathcal{X}$ in a three-dimensional scheme or we can reduce the three-dimensional problem into two dimensions using an analytical resource usage estimation model. Resource usage is linearly related to *cores*, and after generating a single *core* bitstream we can create a simple analytical resource model which reduces the parameter space to two dimensions. Density factor d_f is a software parameter while m_w and *cores* affect the bitstream. Varying d_f only involves software execution, as long as a bitstream for the given m_w is already generated. If we evaluate a design with m_w (or m_w, *cores*) that has not been evaluated before, we generate a new bitstream.

We present a visualization of the two-dimensional optimization in Fig. 5, where the ϵ_{rms} limit is set to a value of 0.1. The bottom-left corner of $\mathcal{V}$ contains the global optimum which is difficult to determine without additional benchmark evaluations, as the maximum number of possible *cores* and therefore throughput is limited by FPGA resources and as a result is chip dependent. Regions of space with low d_f or m_w are correctly predicted to offer low accuracy (light gray area).

To measure the algorithms convergence the MLO terminates when the design with the highest throughput at the specified precision is found. The number of required f evaluations is shown in Tab. 2. The previously suggested optimization scheme [12] involves generating bitstreams for the full m_w range. Using our MLO combined with the analytical resource model, we reduce the number of bitstream

Table 2. Average number of f evaluations - Quadrature application optimization

cores	ϵ_{rms} 0.1	0.01	0.001
three-dimensional	138	67	47
two-dimensional	71	43	28

generations as we avoid exploring *cores* and thus decrease the design space. Around 20-50% of f evaluations involve bitstream generation. The number has a high variance between individual runs as the swarm either skips undesirable configurations or thoroughly explore the whole design space.

The optimization scheme presented in [12] involves generating all possible bitstreams with $cores = 1$, and a binary search of the d_f values. Once the optimal (d_f, m_w) tuple is found, the number of *cores* can be determined. It also requires the generation of bitstreams for all m_w resulting in 53-11=42 distinct bitstreams. Furthermore, the number of bitstreams is nearly doubled since after the first generation, usually a second bitstream generation follows to adjust *cores*. Binary search is performed on the d_f range of 32-4=28 distinct values per bitstream, which yields on average $2 \times \lceil \log_2(28) \rceil = 10$ benchmark evaluations per bitstream.

In comparison the MLO performance can be measured both in terms of f evaluations and bitstream generations. Using the optimization approach from [12] we perform a binary search on d_f range for all m_w values resulting on average $10 \times 42 = 420$ f evaluations regardless of the ϵ_{rms} limit. As presented in Tab. 2 for $\epsilon_{rms} = 0.1$ the MLO requires 75 evaluations (85 % less) in the two-dimensional scheme and 138 (67 % less) in the three-dimensional scheme. The number becomes more favorable for the MLO when ϵ_{rms} is reduced as $\mathcal{V}$ is decreased and the MLO needs to explore a smaller area, while average number of f evaluations in their optimization approach stays constant. Not all f evaluations involve bitstream generations: for $\epsilon_{rms} = 0.1$, 50% of f evaluations involve two bitstream generations resulting in 71 bitstreams compared to 82 bitstreams in [12]. In the three-dimensional scheme, MLO further decreases number of bitstream generations, to an average of 69. Our automated approach clearly outperforms manual design both in terms of f evaluations and bitstream generations, although in the second case the results are less dominant.

6 Conclusions and Future Work

We have proposed MLO, a novel tool which can determine optimized parameter configuration of a reconfigurable FPGA design. The MLO can offer superior performance, while reducing effort on analysis and application-specific tool development. The main advantage of using the MLO is a shift from manual optimization to automatic computation. The MLO requires multiple benchmarks for further evaluation, and there are many opportunities for future work; an example is the development of new surrogate models that would allow the reduction of required benchmark samples and efficiently address high dimensional examples. There are numerous cases where level of parallelism, timing and other parameters span tens of dimensions and would benefit from an effective automated approach.

Acknowledgements. This work is supported by the European Union Seventh Framework Programme under grant agreement number 248976, 257906, 287804 and 318521, by UK EPSRC, by Maxeler University Programme, and by Xilinx.

References

1. Jin, Y., Olhofer, M., Sendhoff, B.: A framework for evolutionary optimization with approximate fitness functions. IEEE Transactions on Evolutionary Computation 6(5), 481–494 (2002)
2. Ong, Y.S., et al.: Evolutionary optimization of computationally expensive problems via surrogate modeling. AIAA 41(4), 689–696 (2003)
3. Su, G.: Gaussian process assisted differential evolution algorithm for computationally expensive optimization problems. In: PACIIA, pp. 272–276. IEEE Computer Society (2008)
4. Guoshao, S., Quan, J.: A cooperative optimization algorithm based on gaussian process and particle swarm optimization for optimizing expensive problems. In: CSO, vol. 2, pp. 929–933 (2009)
5. Thi, H.A.L., Pham, D.T., Thoai, N.V.: Combination between global and local methods for solving an optimization problem over the efficient set. EJOR 142(2), 258–270 (2002)
6. Kurek, M., Luk, W.: Parametric Reconfigurable Designs with Machine Learning Optimizer. In: FPT (2012)
7. Pilato, C., et al.: Improving evolutionary exploration to area-time optimization of FPGA designs. J. Syst. Archit. 54(11), 1046–1057 (2008)
8. Seeger, M.: Gaussian processes for machine learning. International Journal of Neural Systems 14, 69–106 (2004)
9. Rasmussen, C., Williams, C.: Gaussian Processes for Machine Learning. MIT Press (2006)
10. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)
11. Van Den Bergh, F.: An analysis of particle swarm optimizers. Ph.D. dissertation, University of Pretoria, South Africa (2002)
12. Tse, A.H.T., Chow, G.C.T., Jin, Q., Thomas, D.B., Luk, W.: Optimising Performance of Quadrature Methods with Reduced Precision. In: Choy, O.C.S., Cheung, R.C.C., Athanas, P., Sano, K. (eds.) ARC 2012. LNCS, vol. 7199, pp. 251–263. Springer, Heidelberg (2012)
13. Becker, T., Luk, W., Cheung, P.Y.K.: Parametric Design for Reconfigurable Software-Defined Radio. In: Becker, J., Woods, R., Athanas, P., Morgan, F. (eds.) ARC 2009. LNCS, vol. 5453, pp. 15–26. Springer, Heidelberg (2009)

Performance Modeling of Pipelined Linear Algebra Architectures on FPGAs

Sam Skalicky, Sonia López, Marcin Łukowiak, James Letendre, and Matthew Ryan

Rochester Institute of Technology, Rochester NY 14623, USA

Abstract. The potential design space of FPGA accelerators is very large. The factors that define the performance of a particular implementation include the architecture design, number of pipelines, and memory bandwidth. In this paper we present a mathematical model that, based on these factors, predicts the computation time of pipelined FPGA accelerators. This model can be used to quickly explore the design space without any implementation or simulation. We evaluate the model, its usefulness, and ability to identify the bottlenecks and improve performance. Being the core of many compute-intensive applications, linear algebra computations are the main contributors to their total execution time. Hence, five relevant linear algebra computations are selected, analyzed, and the accuracy of the model is validated against implemented designs.

1 Introduction

Compute intensive applications, when implemented in a standard CPU, may not achieve an acceptable level of performance for their required purpose. Some of these, including medical diagnosis[1], weather prediction[2], or stock market analysis show great potential in their respective fields, but often require alternative hardware solutions such as GPU or FPGA accelerated implementations to operate within an available time budget.

The core of these applications is very often composed of linear algebra computations such as dot product, matrix-vector multiplication, matrix-matrix multiplication, matrix inverse and matrix decomposition. When implementing some of these on an FPGA, making a sound design decision can be highly time consuming due to the size of the design space. Initially, an architecture must be selected and given the area and bandwidth constraints, the number of pipelines or pipeline size determined. Tools that facilitate this process are highly desired.

In this paper we present a mathematical model that allows us to calculate the performance of several pipelined linear algebra designs and assists in the configuration of the size or number of pipelines. Out of the literature [3–6], successful architectures for these computations are selected and discussed. Then, model parameters for these architectures are defined and are used to calculate the execution time. The mathematical model is based on factors such as the number of pipelines and memory bandwidth limitations. The accuracy of our mathematical model is verified by comparing against the actual simulated hardware implementations. We discuss the results and identify the performance limiting factors such as computational ability or memory bandwidth.

The main contributions of this work are:

- A novel mathematical model that provides the execution time of pipelined FPGA accelerators.
- Examples of pipelined architectures decomposed into equations.
- Evaluation of the approach on accelerator designs, showing that the model provides clear indicators of performance limitations due to system design factors.

2 Related Work

Performance optimized source code for Linear Algebra (LA) computations is found in two commonly used software libraries: Basic Linear Algebra Subprograms (BLAS)[7] and Linear Algebra PACKage (LAPACK)[8]. Parallel versions of these libraries are also available for GPUs in the form of Compute Unified BLAS (CUBLAS)[9] and Matrix Algebra on GPU and Multicore Architectures (MAGMA)[10].

FPGA implementations of these computations have been studied extensively with various design goals in mind. Akella *et al.* [11] designed a sparse matrix-vector multiplication architecture for a single FPGA. Lin *et al.* [12] designed and evaluated a design for matrix-matrix multiplication that is optimized for the extra control required for sparse matrices. In their subsequent work, Lin *et al.* [13] presented a model specifically designed for sparse matrix-vector and matrix-matrix multiplication and considered dense matrices only as a special case of sparse matrices. Prasanna *et al.* [14] evaluate various computations including matrix-vector multiply for dense and sparse matrices but don't investigate different pipeline sizes.

Zhuo *et al.* [3] designed architectures for dot product and matrix-vector multiplication and analyzed the trade-offs using architecture parameters for high performance. This work also presented a lower bound for latency of these computations for both the memory and pipeline usage. Matrix-matrix multiplication has been researched by many including [3, 4, 12] among others. Sotiropoulos *et al.* [4] designed a fast block-based matrix-matrix multiplication architecture that accelerated the computation using digital signal processing (DSP) units. An architecture for Cholesky decomposition by Yang *et al.* [6] modified the standard calculation to remove square roots and divisions from the pipelines. This design shares a single divider among all of the pipelines efficiently regardless of the number of pipelines. Edman *et al.* [5] studied a linear array design for matrix inversion using a single stage pipeline and multiple stage pipelines for a 4x4 input matrix. This matrix inverse design improved upon existing systolic arrays by converting to a linear structure that used less resources, yet still performed just as well given that the array length is fixed to the matrix size.

In modeling of FPGA architectures for design space exploration prior to architecture design, Holland *et al.* [15] presented a method to analyze the amenability of an algorithm for implementation in an FPGA. They further expanded the model for multi-FPGA systems in [16]. Their goal was to predict the performance without any architectural design details of the algorithm. In their verification, they presented modeling error rates up to 15% compared to their baseline performance. We argue that, for applications that are iterative in nature such as heterogeneous computing simulators where

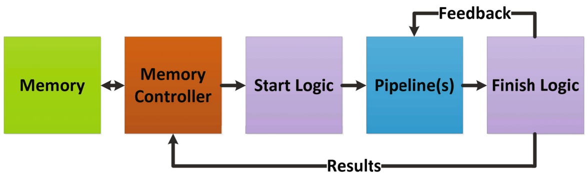


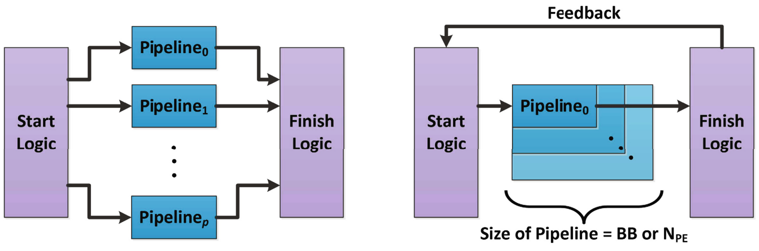
Fig. 1. Embedded System Diagram

error compounds, a more accurate model is required. Furthermore, once the architectural details are known we can more accurately model the performance of the design.

3 FPGA Pipeline Model

Before development of the model, we made a few assumptions based on the memory systems available for FPGAs [12, 17]. First, we assume that the memory controllers abstract out the control necessary to utilize any bursting or other performance enhancing aspects. Second, we assume these interfaces handle the issues such as reading/writing individual elements, avoiding bank conflicts, and buffering up accesses [18]. Finally, we assume that the data stored in memory is organized such that the operands required by the pipelines are able to be read out in sequential fashion.

We consider an embedded system with a memory interface, compute pipelines, and start/finish control logic. The start and finish logic represent any data ordering or precomputing that must be done to all data before or after the compute pipeline(s). The compute pipelines are made up of stages connected together, each containing one or more *functional units* that perform operations such as adding, subtracting or multiplying. Computations can be performed using one or more pipelines by cycling data through them. The architectures are classified into two types: multiple pipelines and scalable pipelines. Multiple pipeline structures are defined as a single pipeline replicated one or more times. Example architectures of this type are dot product and matrix-vector multiply. On the other hand, scalable pipeline structures are those architectures that have data dependencies or reuse results iteratively . Examples of scalable pipelined architectures are matrix-matrix multiply, matrix inverse, and matrix decomposition.



(a) Multiple Pipeline Structure (b) Scalable Pipeline Structure

Fig. 2. Types of pipelined architectures

The size of the pipeline can be represented as the size of some basic block (BB) size or number of processing elements (N_{PE}).

The total execution time (t) for a given computation is determined by: $t = n_c/f_p$ where n_c is the total number of cycles required and f_p is the pipeline's frequency.

3.1 Determining the Total Number of Cycles

Let L_p be the pipeline latency, which is the number of cycles starting from when the first data is ready at the input until the first result is available at the output. L_p can be computed as the sum of each stage's individual latencies (L_{Stage}), for all stages (N_{Stages}) in the pipeline. Let Op_{max} be the speed of reading operands from memory given the usable memory bandwidth Mem_{BW}, and the size of each operand, Op_{size}:

$$L_p = \sum_{i=1}^{N_{Stages}} L_{Stage,i} \qquad\qquad Op_{max} = \frac{Mem_{BW}}{Op_{size}} \tag{1,2}$$

Let Op_{req} be the required number of operands per pipeline. We calculate Mem_{ratio}, the ratio of Op_{req} at the frequency of f_p to Op_{max} as shown in Equation 3. This ratio determines the percentage of the bandwidth a single pipeline will consume. The inverse of this gives the maximum number of pipelines M_p, that can be provided with new data at every cycle:

$$Mem_{ratio} = \frac{Op_{req} \times f_p}{Op_{max}} \qquad\qquad M_p = \frac{1}{Mem_{ratio}} \tag{3,4}$$

The number of pipelines that can be used to compute a particular operation will be limited by either area (hardware resources) or memory bandwidth. Let A_p be the number of available pipelines that can be implemented on a particular FPGA. Let M_p be the number of pipelines that can be provided with new data from memory every cycle. Let U_p be the number of usable pipelines given by the minimum of M_p and A_p:

$$U_p = \min\{M_p, A_p\} \qquad\qquad c_c = \left\lceil \frac{U}{U_p} \right\rceil \times I \tag{5,6}$$

Given a particular pipeline architecture, a computation can be represented as some amount of reuse of the pipeline, and the cost for each use. The amount of reuse, denoted as uses (U) can be spread over a number of pipelines for efficient parallel computation. The number of cycles required for each use is referred to as the number of iterations (I). With these variables, we define the number of compute cycles c_c as shown in Equation 6. Let n_c be the total number of cycles required to complete a single computation. n_c can be calculated as the sum of the c_c and L_p plus the control latency L_{ctrl} that results from the start and finish logic (see Figure 1) and twice the memory access latency L_{mem} for each: reading the first set of values, and writing the last result.

$$n_c = c_c + L_p + L_{ctrl} + 2L_{mem} \tag{7}$$

3.2 Determining the Pipeline's Operating Frequency

The calculation of the pipelines' required bandwidth is based upon the number of usable pipelines U_p, Op_{req} and Op_{size} running at the frequency f_p as shown in Equation 8. The ratio between the pipeline's required bandwidth and the available memory bandwidth, BW_{ratio} is:

$$BW_{req} = Op_{req} \times Op_{size} \times U_p \times f_p \qquad BW_{ratio} = \frac{Mem_{BW}}{BW_{req}} \tag{8,9}$$

This ratio is used to determine the maximum frequency of the FPGA pipelines, f_p. This value can be limited by two factors, the memory bandwidth or the max post place and route frequency f_{max}. It is computed as the minimum of these two factors:

$$f_p = \min\{f_{max}, BW_{ratio} \times f_{max}\} \tag{10}$$

If f_{max} is not known an estimated value can be used. Once known the actual value can be substituted and used for any input data size. This maximum frequency takes into account all of the system variables including memory, resources, computation, and architecture design.

4 Computations and Pipelines

The five computations selected for this work (dot product, matrix-vector multiplication, matrix-matrix multiplication, matrix inversion, and Cholesky decomposition) represent types of algorithms that are constrained by different factors: memory bandwidth dependent, high computational complexity, and complex control flow. Out of the various FPGA pipelined architectures that have been designed, one was selected [3–6] for each computation and analyzed using the model. Other architectures can be analyzed with this model as well.

We have derived the equations representing the number of uses U and iterations I required to complete a single computation for all of the computations. The results of these equations are based on the size of the input matrices (MxN and NxP) or vector (N). Due to space constraints, only the derivation of the equations for matrix inverse are presented. The calculation for matrix inverse is based on the number of processing elements (N_{PE}) which defines the maximum amount of parallelism available due to hardware resources. The equations for matrix inverse are shown in Table 1.

Matrix Inversion. The matrix inversion pipeline [5] is composed of processing elements (PEs) that each contain all the required components to perform the function of either an edge cell or an internal cell from previously designed systolic array based architectures. A PE contains a subtractor, multiplier, and divider functional units that are multiplexed as needed. One PE can be used to complete an entire computation, but more can be used for increased performance. The pipeline of PE(s) computes the inversion of one row of a matrix at a time for square NxN matrices. This means that the total number of uses of the pipeline is equal to the number of rows, N. Intermediary values are stored within the PEs and used for the remaining rows. In the equations

Table 1. Computation *Uses* and *Iteration* Equations

Computation	***Uses*** (U)	***Iterations*** (I)
M-V Multiply	$U = N$	$I = M$
M-M Multiply	$U = \frac{M \times P}{BB^2}$	$I = N$
Matrix Inverse	$U = N$	$I = 4\sum_{i=1}^{S-1} \left\lceil \frac{i}{N_{PE}} \right\rceil + \left\lceil \frac{N_{ops} - (2 \times S \times (S-1))}{S} \right\rceil \times \left\lceil \frac{S}{N_{PE}} \right\rceil$ $N_{ops} = N \times \frac{N+1}{2}$; $S = \lceil \frac{N}{2} \rceil$
Cholesky	$U = 1$	$I = \left[\sum_{i=1}^{S-1} i \times N_{PE}\right] + \lfloor \frac{N}{S} \rfloor \times S$; $S = \left\lceil \frac{N}{N_{PE}} \right\rceil$

for matrix inverse in Table 1, S represents the number of stages and thus the amount of parallelism present in the computation given a particular number of PEs. However, since this design passes values between cells a schedule is needed and the number of operations to compute (N_{ops}) is based on the matrix size.

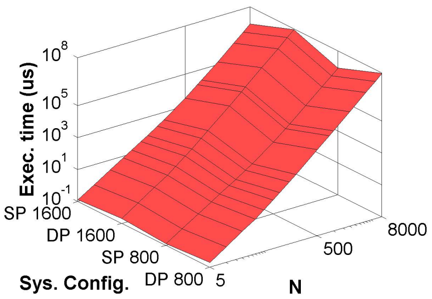
5 Verification and Results

The model presented in Section 3 is used to predict the performance of pipelined FPGA accelerators. The model requires the following factors to calculate execution time: matrix/vector size, number of pipelines, memory bandwidth, and maximum clock frequency of the design. The linear algebra computational architectures presented in Section 4 were examined in detail to optimize each of the designs for high performance. All of the designs were implemented for a wide range of pipelines, BBs, and PEs limited by the resources of the device. Each was evaluated using 5x1 to 8000x1 vectors and 5x5 to 8000x8000 square matrices. We used square matrices without loss of generality by using a block-based approach for matrix computation. The post place and route clock frequencies for each of these designs in a Virtex-6 240T and Virtex-7 485T device were recorded and used in the pipeline model calculations. We evaluated the designs for both single and double precision floating point in a device with an $800 Mbps$ memory bandwidth (configurations: Virtex-6, SP 800 and DP 800) and a device with a $1.6 Gbps$ memory bandwidth (configurations: Virtex-7, SP 1600 and DP 1600).

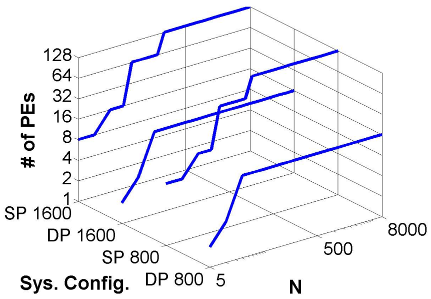
Verification of the model's accuracy was confirmed by comparing simulations of the hardware designs to the model's predictions. A linear regression analysis of the experimental data from implementation with the model's estimated performance resulted in a coefficient of determination (R^2) of 1 for all computations. This result confirms the validity of our model's ability to accurately predict the performance of pipelined architecture designs.

Results of modeling the five selected linear algebra computations are displayed in two parts each. The first graph (a) shows the estimated best execution times of the four system configurations for a range of data sizes. The second graph (b) shows the number of pipelines or pipeline sizes used to achieve that best execution time.

Matrix Inverse. The matrix inverse computation was implemented using PE counts from 1 to 128. This design only requires one new operand from memory per cycle for



(a) Execution Times for NxN matrices



(b) Pipeline sizes to achieve max. perf.

Fig. 3. Matrix Inverse Performance Results

any number of PEs. The memory bandwidth was not a limiting factor since it was never fully utilized for this computation. The performance was exclusively reliant on the number of computational units and speed. For single precision, the larger data sizes from 200 to 8000 used 128 PEs for optimal performance. For double precision, the associated clock frequencies were much lower and for data sizes from 20 to 8000 used only 16 PEs for optimal performance. The results in Figure 3 show that the trend lines for SP and DP have the same pipeline sizes. This clearly illustrates that the performance of matrix inverse is much more dependent on the computational ability of the hardware than any other computation in our comparison. The saturation of the number of PEs for the larger data sizes shows all hardware resources being used.

Other Computations' detailed results have been omitted due to space constraints. A summary of those results follows. We found that with an increase in the data size the optimal number of pipelines may be less than for the smaller data size. This was due to a mismatch of the data size to the number pipelines. The source of this stems from Equation 6, where the division of the number of uses by the number of usable pipelines is rounded up. This increase in the number of cycles coupled with a reduction in overall FPGA clock frequency for a larger number of pipelines can result in reduced performance. A smaller number of pipelines offer a finer granularity than that of the larger designs, and this allows designs with less pipelines to better compute a wider range of data sizes. In many cases we found that the optimal match between computational ability, memory bandwidth, and the FPGA clock frequency produced the highest performing designs. Once a design with a particular number of pipelines fully utilizes the memory bandwidth, any increase in the number of pipelines will reduced performance.

6 Conclusions

We have presented a model that can accurately predict the performance of pipelined FPGA architectures. Our results show that the performance of a computation depends on the implementation's efficient use of available resources. Through the model calculations, the factors that constrain the performance are identified. We concluded that each of the five linear algebra computations require a different combination of system factors to achieve best performance. We validated the predicted performance of the model with implementations of each architecture and accurately correlated the experimental results with the predicted results. Compared to the previous FPGA models [15, 16] our model achieves better accuracy by incorporating elements from the architecture through the uses, iterations, and latency of each stage, L_{Stage}. The performance across a wide range of data sizes and pipline sizes were displayed. We showed that the size of the pipeline is a critical parameter that must be tuned to achieve maximum performance. Using the model, we can compute the time required given the particular matrix size. For different computations, the best accelerator can be configured to meet the needs of the application. New pipelined architectures can be added and evaluated using this model to compare performance without implementation. This model can be extended by using cost-benefit analyses to compare different design strategies for optimal performance, resource and power utilization.

References

1. Wang, L., et al.: Physiological-model-constrained noninvasive reconstruction of volumetric myocardial transmembrane potentials. IEEE TBME 57(2) (2010)
2. Bengtsson, T., et al.: Adaptive methods in numerical weather prediction. In: First Spanish Workshop on SpatioTemporal Modeling of Environmental Processes (2001)
3. Zhuo, L., et al.: High-performance designs for linear algebra operations on reconfigurable hardware. IEEE Transactions on Computers 57(8) (2008)
4. Sotiropoulos, I., et al.: A fast parallel matrix multiplication reconfigurable unit utilized in face recognitions systems. In: Field Programmable Logic and Applications (FPL) (2009)
5. Edman, F., et al.: Implementation of a highly scalable architecture for fast inversion of triangular matrices. In: IEEE Intl. Conf. on Electronics, Circuits and Systems (ICECS) (2003)
6. Yang, D., et al.: Compressed sensing and cholesky decomposition on FPGAs and GPUs. Parallel Computing 38(8) (2012)
7. Dongarra, J.J., et al.: A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software (TOMS) 16(1) (1990)
8. Anderson, E., et al.: LAPACK Users' guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)
9. nVidia Corporation, CUDA toolkit 4.2 CUBLAS library documentation (2012)
10. U. of Tenessee Innovative Comp. Lab. Matrix algebra on GPU and multicore architectures, Knoxville, TN, USA (2012), `http://icl.cs.utk.edu/magma/`
11. Akella, S., et al.: Sparse matrix-vector multiplication kernel on a reconfigurable computer. In: 2005 IEEE High Performance Extreme Computing Conference (HPEC 2005) (2005)
12. Lin, C.Y., et al.: Design space exploration for sparse matrix-matrix multiplication on FPGAs. In: Field-Programmable Technology (FPT) (2010)
13. Lin, C.Y., et al.: A model for matrix multiplication performance on FPGAs. In: Field Programmable Logic and Applications (FPL) (2011)
14. Prasanna, V.K., et al.: Sparse matrix computations on reconfigurable hardware. Computer 40(3) (2007)
15. Holland, B., et al.: RAT: RC amenability test for rapid performance prediction. ACM Transactions on Reconfigurable Technology and Systems (TRETS) 1(4) (2009)
16. Holland, B., et al.: An analytical model for multilevel performance prediction of multi-FPGA systems. ACM Transactions on Reconfigurable Technology and Systems (TRETS) 4(3) (2011)
17. Xilinx Inc., Memory Interface Generator - White Paper WP260, (February 2007)
18. Xilinx Inc., Xilinx Virtex 6 Memory Interface Solutions - User Guide UG406 (June 2011)

Fast Template-Based Heterogeneous MPSoC Synthesis on FPGA

Youenn Corre[1], Jean-Philippe Diguet[1], Loïc Lagadec[2], Dominique Heller[1], and Dominique Blouin[1]

[1] Lab-STICC, Université de Bretagne-Sud, 56100 Lorient, France
[2] Lab-STICC, ENSTA Bretagne, 29200 Brest, France

Abstract. Our contribution lies in offering a fast and parametrized domain-space exploration to the designer, whose expertise drives the whole process while staying the actor of added-value creation. In this paper, we present two new features and two important improvements of our H-MPSoC synthesis framework. The first one is a new template-based approach for automated design space exploration and synthesis. A template describes an architecture model for a specific domain and has three levels of specifications each representing a different level of design expertise. We also rely on the Model-Driven Architecture (MDA) paradigm to provide flexibility, reusability and code generation for different FPGA targets. We have refined the communication models to get more accurate performance estimations. Finally, we also improved our mapping decision algorithm that drastically reduces the simulation time. The output is the synthesizable code of the hardware architecture, the adapted C code of the application and the project files for FPGA design tools. We use an MJPEG decoder as a case-study to validate our framework on a Xilinx FPGA.

1 Introduction

Current trend in embedded systems development is to clearly increase the complexity of the system while keeping size and power consumption low. This trend has lead to the development of Heterogeneous MPSoCs, which provide the necessary energy efficiency for embedded systems. The heterogeneous nature of MPSoCs makes their design complex since it has to deal with processing units of various kinds including General Purpose Processors (GPP) and specialized co-processors. The key issue to reach a high quality standard in hardware design is to unleash the unbeatable skills of experienced designers. Our contribution lies in combining state-of-the-art tools to set up a full ESL flow that promotes usability i.e. designers needs, simplicity to avoid unnecessary information, automation of tedious tasks with no added value, and incremental improvements since practice makes success. As a consequence, the designer is put at the center of the value-added process, while being freed from tasks that do not deserve his/her expertise. Fast and parametrized domain-space exploration is offered as a facility to the designer. Besides, interesting subsets can be further implemented including specific coprocessors.

Software design often relies on Model-driven approach, typically combining UML models with object-oriented programming languages. This approach is widely used in the software industry, and has proven its efficiency on the field. Such an approach allows for correct-by-construction code generation and can also be used to apply automated design verifications by checking predefined design rules. It thus leads to a huge boost in productivity by ensuring the validity of the design thus reducing time consuming debug tasks. In this work such an approach is applied to H-MPSoC synthesis. Considering the strategic importance of system-level design reuse, offering a library of architecture templates makes sense. Each template targets a specific application domain (DSP, video, etc.). A template is an architecture model made of two types of components (processor, communication link, memory, peripheral). The first ones are pre-instantiated while the second ones are available and may or may not be implemented. The designer can specify design constraints and, if needed, tweak the design for its own application. The objective of the framework is to automatically explore the design space, to fill the template blanks and to generate the complete system. The use of templates thus relieves designers from tedious tasks with no compromise on the ability for the designers to provide manual choices according to their level of expertise. Jointly, the MDA approach also provides a solution for design reusability, which is actually limited due to the lack of standards in FPGA design.

Given these templates, along with the designer constraints and annotations, it is possible to perform real-life design space exploration, with, as a result, a restricted range of tradeoffs between cost and performances. The final system selected by the designer is then generated including the synthesizable code of the hardware platform, the adapted software code of the input application, and the project files corresponding to the FPGA backend tools.

Efficient and useful design space exploration also requires fast and accurate performance and cost estimators. It has to take into account the two main factors that impact the performances. The first one is the model of processing units (either general purpose processors or dedicated hardware accelerators). The second one is the model of interconnects and memory architectures that may reflect possible contentions and bottlenecks.

The paper is structured as follow: we first compare our approach to existing solutions in Section 2 and give an overview of our framework in Section 3. We then present our first contribution, namely the template-based solution in Section 4. We introduce our improved communication model and its validation in Section 5. In Section 6 we present our new data and task mapping strategy which achieves significant DSE time speedup compared to our previous work. Finally a case study of an MJPEG video decoder is presented in Section 7 and we conclude in Section 8.

2 Related Work

Several tools and languages exist for modeling of system platforms. Compared to other approaches such as MARTE, our modeling-driven approach is domain-specific and focuses on H-MPSoC for embedded systems.

In [1] a DSE method is proposed for MPSoC using Model Driven Engineering. Descriptions of the application and platform are given in the form of UML models and are then used for DSE. DSE is implemented as a heuristic which explores the following dimensions: communication, memory, performance and power consumption. The mapping is then performed, relying on model-based estimates. While this approach is quite similar to ours, our framework also takes into consideration the automated exploration of hardware accelerators and the exploitation of data parallelism.

In [2] is presented an MDE-based DSE for MPSoC. The tool starts from the MARTE description of a system and the DSE process is performed through successive simulations at different granularity. At first, a lot of solutions are quickly estimated with a fast coarse granularity simulation, then from these estimations a selection is made which is then evaluated more accurately with a slower finer grained simulator. MDE is here used for automated model-to-model transformations, more specifically from the MARTE description to the simulation description language. We also use model transformations, but we use it as well for architecture generation. This framework is not as complete as ours, since they do not implement an automated hardware accelerators exploration and synthesis, nor do they generate the final implementation files to feed FPGA backend tools.

Our framework is partly based on the Daedalus framework [3] and borrows its performance simulation tool as well as its automatic transformation of the application into Kahn Process Network (KPN) [4]. This implies that we use the same formalism as input. However we have added several improvements, which are the automated exploration, based on HLS, of the possible hardware accelerators, the exploration of the data parallelism and a faster DSE algorithm.

3 Overview of the Framework

Our framework includes original contributions over a set of existing tools, using standards in order to maximize modularity and flexibility. A number of tools have been considered in order to deal with already solved problems. The Figure 1.a shows an overview of our framework flow.

The inputs of our tool are the architecture template, which is detailed in section 4 and the C code of an application, which must be a static affine nested loop program [6], i.e. the loop indexes must remain static through the execution and its values must evolve following an affine function. The application must be split into tasks to express the parallelism of the program with a granularity at the function level. This splitting decision is up to the designer. Then, the code is automatically transformed into tasks following a Kahn Process Network [4] formalism. This is achieved by using the KPNGen [7] tools from the Daedalus framework. KPNGen produces several C code source files corresponding to the task-based application. Design Space Exploration requires to score performances of every task. Hence, the code is automatically adapted in order to perform a profiling on a single softcore target implemented on the FPGA target. Each task of the application is transformed into a POSIX thread and is run on the Xilinx' kernel, Xilkernel [8].

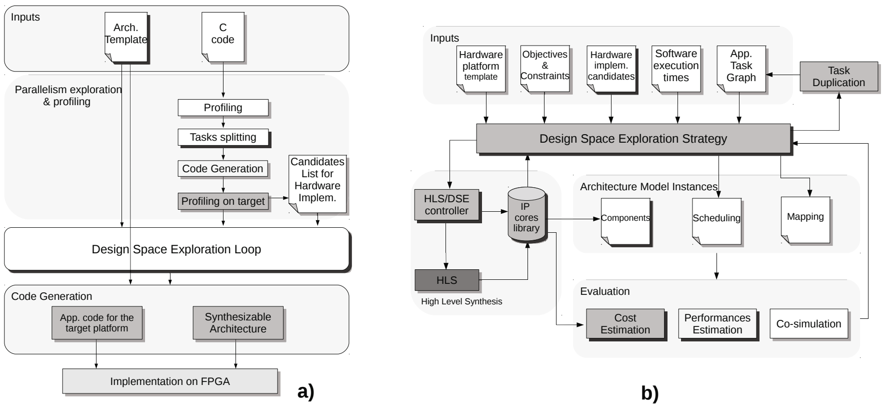


Fig. 1. a) Flow of our framework: our tools are in blue, Xilinx tools in green and the Daedalus tools in yellow. b) DSE flow: Our contributions are in blue, borrowed tools or formalisms from the Daedalus frameworks are in yellow and the HLS tool (GAUT, [5]) is in red

Additionally, APIs controlling the timers responsible for profiling performances are also inserted to the code. This was done by modifying the code generation part of ESPAM. The collected information is then used to sort the tasks in resource consumption order. The obtained accurate values are used to calibrate the performance estimation of software implementation during DSE.

A detailed explanation of the DSE algorithm that mainly deals with architecture exploration is given in [9]. The DSE flow is also illustrated in Figure 1.b. In this paper, we introduce a new data and task mapping algorithm which is explained in section 6. DSE is based on a scalable algorithm, allowing to balance the necessary speed to produce a satisfying result versus the size of the explored design space. The scalability covers the full range of possibilities from a depth-first search returning the first result satisfying the designer's constraints to an exhaustive search that will return the optimal solution. The DSE algorithm explores several dimensions such as the number of processors, the number and type of dedicated co-processors, hardware specialization, communication links and memories as well as mapping and scheduling. Several pruning decisions are also introduced in the DSE algorithm. In addition to these typical design dimensions two additional dimensions are explored in order to improve the performance of the system. The use of dedicated co-processors is introduced by means of High-Level Synthesis (HLS) and the possible data-parallelism through task duplications. Since not all tasks can be accelerated into hardware or can benefit from data-parallelism, the designer has to specify which tasks are eligible to such optimizations. The integration of HLS in the DSE loop allows to generate several hardware accelerators for the same function, thus offering a tradeoff between area and performance during design decision. Another important point is that communication through data buffers are based on a zero-copy mechanism.

The architecture exploration provides N_1 architecture candidates for the following steps. The same exploration strategy is then followed, data mapping produces N_2 solutions based on the N_1 candidates. Then task mapping produces N_3 solutions based on the N_2 candidates. The idea is to launch the greedy scheduling simulation step on a limited number of N_3 solutions. Once the exploration is over, the architecture that fits the designer's constraints is selected and the corresponding synthesizable hardware code is generated. The C code of the application is also adapted to the architecture, by adding calls to accelerators and API for synchronization and communication routines.

4 MDA for H-MPSoC Design

In order to provide maximum flexibility, a Model Driven Architecture approach (MDA) has been adopted. This method, widely used in software industry, provides an easy way to adapt design to different target architectures and thus provides an easy way for portability and adaptation to target evolutions.

4.1 Template-Based Systems Design

There is no underlying platform architecture on FPGA and therefore no standard interfaces [10]. It means that the development of FPGA-based embedded systems remains a tedious task for usual designers. This situation explains the weak penetration of FPGA on the embedded system market. We propose to solve this issue by means of architecture models specified as templates. Our model-driven approach relies on templates that provide pre-parametrized designs according to the target domain (DSP, video, etc.). Such an approach favors reusability, code generation and performance estimations. By allowing reusability, our template-based approach avoids some tedious design steps and thus relieves designers from repetitive and potentially error-prone tasks.

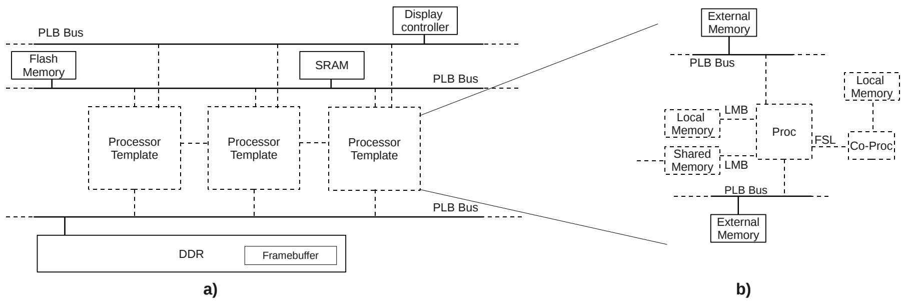


Fig. 2. An example of a graphical representation of an architecture template for an MJPEG decoder and a detailed view of the processor template. Solid lines indicate fixed part (level 1) of the architecture while dotted lines represent potential instantiations (levels 2 and 3) that are decided by our tool during DSE.

In order to let designers focus their expertise on high added value points, we have divided our template into three levels of specifications. A first (*static*) level represents domain-specific elements that are fixed and cannot be modified during DSE.

A second level (*DSE bounds*) is where the designer has to provide information needed for the DSE. These specifications provide the constraints that will bound the design space:

– architecture template choice;
– minimum and maximum numbers of processors as well as the available types;
– available memories and their maximal size;
– mapping of input and output data;
– mapping of specific task(s) (input and output tasks typically);
– specification of application tasks that can be accelerated through hardware accelerators and/or that can be duplicated for data parallelism exploitation.

A third level (*expert*) of specifications targets all the decisions that are taken by the framework during DSE. If needed, the designer can force some attributes of these specifications to fixed values that will thus not be changed during DSE. These specifications include:

– task and memory mappings;
– number and type of processors;
– HW accelerators exploration, synthesis and integration with SW calls;
– task duplication for data-parallelism exploitation;
– scheduling.

Templates must be created by first specifying the fixed parts of the system, corresponding to the level one specifications. It is possible to use templates of components such as processor, memory or external peripherals of the system. An example of a processor template is given in Fig.2.b. These templates can be stored in a template database for future reuse. Once the designer has specified the parts of the template that needs to be completed (level two and possibly level three), the template is used as an input for DSE.

A graphical representation of an example architecture template for a video-decoder design is presented in Fig.2.a. In this example, the input (e.g. reading a video file on a flash drive) and the output (e.g. writing to the frame buffer) will most likely be the same from one design to another of the decoder. Hence the template pre-instantiates in the architecture the components necessary for the input and output operations. In a video-decoder, the writing of the decoded output video is done in real-time and since it is in a raw uncompressed format, it requires a large bandwidth. It is thus also required to specify the instantiation of a dedicated bus to the frame buffer.

4.2 Architecture Modeling

To model the system architecture, AADL is used (*Architecture Analysis & Design Language*, [11]). This is a domain model language for embedded systems, including both software and hardware representations. AADL provides base components categories for representing processors, buses, memories, devices, systems, processes, threads, thread groups, data and subprograms. Since in our case it is not needed to model software, only hardware components are considered. Each component is described with two distinct types of declaration:

– a type declares the external interface of the component through which it can be connected to other components (I/O ports, data/bus access, etc.);
– an implementation represents the internal composition of the component. It can include one or several subcomponents, and also specify properties.

A set of properties are predefined in the AADL standard to specify configuration parameters of components. For example, for a generic memory component, the predefined properties are the size, the access rights, the word size, read and write times, etc. These predefined properties can be extended by the addition of other property sets for representing properties of more specific components such as a MicroBlaze processor. This is illustrated in Fig. 3 where an excerpt of the AADL representation of a MicroBlaze processor is presented.

AADL component and property sets declarations are used to constitute a library of components to be used in FPGA-based designs. Declarations for Xilinx-specific components such as MicroBlaze, buses, controllers, etc. have been added to the library. This is necessary to specify the values for parameters specific to Xilinx IPs.

From AADL models, we are able to generate the necessary files for implementing the solution architecture in synthesis and implementation tools such as Xilinx XPS or Altera's Quartus. The Xilinx tools have been selected to demonstrate the proposed approach, so the generated files representing the system are the *.mhs* and *.mss* files. The first one describes the synthesized hardware platform with its components, their parameters and their connections. The second one describes the drivers specifications in order to call the hardware components from the software

```
Package xilinx_components
public
with microBlazeProperties;

processor microblaze
  features
    reset : in event port;
    interrupt : in data port;
    data_plb : requires bus access;
    inst_plb: requires bus access;
    data_lmb : requires bus access;
    inst_lmb: requires bus access;
    debug : requires bus access;
    master_fsl: requires bus access;
    slave_fsl: requires bus access;
  properties
  --default values
    microblazeProperties::FSL_links => 0;
end microblaze;
end xilinx_components;
```

Fig. 3. Excerpt of the AADL model of the MicroBlaze

code. The selected architecture resulting from the DSE algorithm, is represented by an instantiation of the AADL model, which contains the instantiated components with their associated parameters and the connections between them. This representation is then automatically transformed, into the corresponding project files for the targeted FPGA design tools for an immediate implementation on the FPGA. Using this MDE technique, only the grammar and the model transformations used for code generation need to be changed to target a different FPGA design tool.

5 Model of Communication and Estimation

Since communications have a major impact on system performances, we have refined their modeling in order to detect early memory and bus possible bottlenecks. The template-based approach allows a precious improvement of this estimation, which mainly rely on the modeling of the underlying architecture.

5.1 Memory Model

Since communication can be a bottleneck, it is important to minimize its cost. Hence it is necessary to explore data mapping on available memory components along with task mapping and coprocessor instantiation. In the template-based approach, input and output data mappings are specified as parts of the input constraints. In the domain of embedded systems, local memories (e.g. BRAM) can be considered to store tasks binaries, this is for example a possible level one specification. However the mapping of the transferred data between the tasks of the application comes from the DSE algorithm. In order to perform data mapping, models must be provided, one per type of memory. They are characterized with the following parameters:

– *Type*: DDR, SRAM, BRAM, Flash.
– $CommunicationInterface$: The available communication interface to reach the memory (e.g. PLB bus).
– *ReadLatency* & $WriteLatency$: number of cycles necessary to prepare data before a burst.
– *Bandwidth*: transfer capacity of the memory in Mbyte/s.
– *MaximumSize*: maximum memory size, this is necessary for instance to evaluate the cost of synthesized memories such as BRAM.

Using these parameters, it is possible to compute the capacity for each kind of memory. For example for a DDR2 memory targeting on the Xilinx XUPV5 FPGA board, the set of specifications would be {*DDR2*, {*PLB*, *XCL*}, *14*, *8*, *337*, *256000000*} (the specifications are in the same order as in the list above).

5.2 Communication Model

The communication model has to reflect the real communication cost in order to accurately estimate the global performance. This is a key point to find the best mapping. Our communication model takes into account the bus type, the memory mapping (i.e. in which memory data is stored) and the use of a DMA.

The communication cost thus depends on the used interconnects between the processing unit and other peripherals. In Xilinx designs, these possible interconnects are for instance:

– Shared Bus: PLB (Processor Local bus);
– Point to point connections: LMB (Local Memory Bus), FSL (Fast Simplex Link), XCL (Xilinx Cache Link).

Each interconnect is characterized by two properties: the *bandwidth* in bits per cycle and the *latency* in cycles. For example, the set of specifications for a PLB bus would be {*PLB, 32, 3*}. During the mapping decision phase, the interconnects as well as the types of memories used are taken into account in order to compute performances and cost of mappings. Such an approach does not make sense unless it has been validated experimentally to reliably reflect the performances of the final circuit.

During our first experimentation, we have observed that the performances of the final implementation did not match the estimated processing performances by an order of magnitude. After investigation, we found out that this was due to contentions in the accesses to buses and memories. These contentions were not precisely taken into account during the first performance estimation phase. Consequently, we have adapted our framework and added communication models to check during DSE that buses and memories have enough transfer capacity according to requirements. Our DSE algorithm can now detect if, for a given mapping, the buses and memories have a sufficient capacity so that estimated performance will correspond to the final FPGA implementation. The details of the data mapping algorithm is given in section 6.

6 Data-Task Mapping Decision Algorithm

The faster the DSE run, the larger the explored solution space is. In our previous implementation, for all candidate architectures, mappings were generated exhaustively and then trimmed following a set of rules. Moreover the data mapping, which has a great impact on performance, was a consequence of the task mapping. The remaining mappings were then all evaluated. Since simulation was the most time consuming part of the DSE, we have implemented a different approach. The new strategy is firstly to produce exactly one mapping by architecture. It means that the chosen mapping must follow a set of guidelines so that it provides an acceptable level of performances. Secondly, we apply data-mapping before task mapping. The mapping algorithm is described as a pseudo-code in Fig. 4.

First of all, some mapping decisions are the results of prior decisions in the DSE: tasks that are accelerated through hardware are automatically mapped on the processor linked to the corresponding co-processor. Then for each of the N_1 architecture, several data mappings are generated. Data are mapped onto

```
TaskClusters = Set of tasks to be mapped gathered as clusters of independent tasks
ProcSet = Set of processors in the architecture
DataSet = Set of data representing communication between two tasks sorted by decreasing size
MemorySet = Set of memories sorted in decreasing order of speed

All hardware accelerated Tasks $T$ are mapped on $Accelerator_i$
//Make data mapping with randomized latency values
for all $N_1$ architecture solutions do
    //Generate $N_2$ data mappings
    //Map biggest data on fastest memories first
    //Consider several sizes for synthesized memories (e.g. BRAM)
    //Randomize memories latencies
    for all Data $D$ in $DataSet$ do
        for all Memory $M$ in $MemorySet$ do
            while M has enough space for D do
                mapDataOnMem(D, M)
            end while
        end for
    end for
    //Map smallest data on fastest memories first
    for all Data $D$ in $ReverseDataSet$ do
        for all Memory $M$ in $MemorySet$ do
            while M has enough space for D do
                mapDataOnMem(D, M)
            end while
        end for
    end for
    //Selection of the $N_2$ best data-mappings
    for all $N_2$ selected mapping solutions do
        //Make first task mapping with less loaded processors
        for all Task T in $TC_1$, the first element of $TaskClusters$ do
            for all Processor P in ProcSet do
                if P is the less loaded proc then
                    mapTaskOnProc(T, P)
                end if
            end for
        end for
        //Hungarian Algorithm
        for all Task cluster $TC_i$ of $TaskClusters$ do
            for all Task $T$ in $TC_i$ do
                for all Processor $P$ in $ProcSet$ do
                    for all Memory $M$ in $MemorySet$ do
                        costMatrix = computeCostMatrix()
                    end for
                end for
            end for
            applyHungarianAlgorithm($TC_i$, costMatrix)
            checkForCongestion()
        end for
    end for
    //Selection of the $N_3$ best task-mappings
end for
```

Fig. 4. Task and data mapping algorithm

the fastest memory as long as it has enough space to store the data. If the fastest memory cannot stored the data then, it is tested on the second fastest and if it still does not fit then it is tested on the next fastest and so on, until a memory with sufficient remaining storage capacity is found. Two strategies are implemented for the mapping of the data: largest data first and smallest data first. Data size is computed by multiplying the size of of the data exchange two tasks by the number of times they should be called within a second, with respect to the performance objective. In order to provide several varieties in data mapping we also randomize the latencies of each memories (with a modifier coefficient within 0.5 and 2). Since some memories can also a variable size since they are synthesize (typically BRAM), we considered several size values for those memories. The result is N_2 data mappings which are pruned of possible doubles.

These data mappings are then used to consider task mappings. The tasks that are not already mapped are then clustered into independent tasks, i.e. tasks that do not communicate directly. This is done in order to maximize the probability of two communicating tasks to be mapped onto the same processor, thus reducing the communication cost. Then each task of the first cluster is mapped on processors having the less load and among those the fastest one for the current task. Load of processors are updated each time a task is mapped. Then for the tasks of the remaining clusters, the Hungarian method [12] is applied. It is thus necessary to compute the cost matrix for each task in the cluster and the available processors. The cost is for a Task T and a coupling of Processor P and a memory M is given by this formula: $(\alpha \times execTime_{(T,P)} + \beta \times comm_{(T,M)}) \times \gamma \times procLoad_{(P)} \times \delta \times MemLoad_{(M)}$, where α, β, γ and δ are coefficients that can be set by designer sto choose which parameters to favor. The result is that each task is mapped on a processor and its communication with a previously mapped task are also mapped on a task.

Once the task mapping is done, data that are mapped in "dynamic" memory, such as BRAM, are assigned to a processor. Depending on the task mapping, if data that corresponds to a communication between two tasks mapped on the same processor, then the BRAM is implemented as a local memory of the processor (as seen as in Figure 2.b). Otherwise the BRAM is implemented as a shared memory associated with the processor that produces the data. Finally a check is performed that computes the load of the buses and other interconnects in order to detect possible congestions.

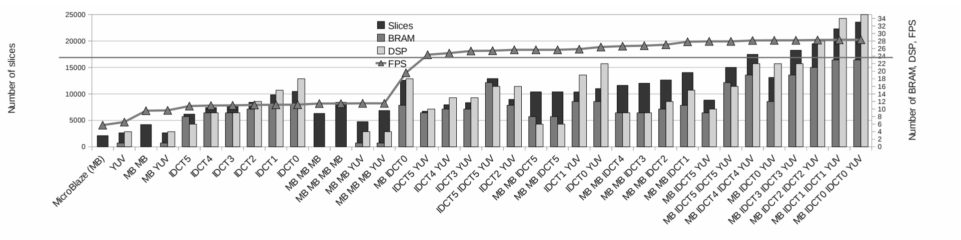


Fig. 5. Selection of results of the DSE algorithm sorted by increasing performances. The horizontal red line shows the 24 FPS objective.

7 Case Study

To validate the performances of our tools, we used an MJPEG decoder as a case study. We used the template previously mentioned (cf. Figure 2). The decoder was split into five tasks: Decode, IQZZ, IDCT, YUV, Display. The target FPGA board was a XUPV5. During the DSE, the explored options were the acceleration of the IDCT and YUV tasks, resulting in five versions of the IDCT IP and one of the YUV IP. The IDCT task was also considered for duplication. In the template, some design decisions are specified for the data mapping: the code of the binary is mapped in SRAM, the heap and stack of each task is mapped into the local memory of the processor the task was mapped on.

The communication model was validated and during DSE several designs that were not respecting the design constraint due to poor communication performances were detected. For example, the case of an architecture with two processors, where Decode and IQZZ are mapped on the first processor and the other tasks are on the second. The data buffer for the communications between the two processors (i.e. between IQZZ and IDCT) was mapped on the DDR memory and connected through a dedicated PLB bus. In addition to the data buffers, the frame buffer was also mapped on the DDR and thus the raw data of the decoded video must be written and read from there. Without the communication model, the given estimated performance respected the 24 FPS constraint, however the estimation provided by the communication model reveals that the final performance would be 15 FPS. Here the bottleneck is not due to the DDR or to the PLB bus, but to the load in the processor induced by the communications which slows down the program execution. This more accurate communication estimation is possible thanks to the template approach we use, which is more precise than the high-level communication models classically used in codesign tools. The observed performance of the generated architecture after implementation on the FPGA was 15 FPS, which confirms that the estimation was correct.

A selection of the results of the exploration is shown in Figure 5. The best solution is the one with two MicroBlaze processors, with two hardware accelerators: one for the YUV and one for the IDCT. Our new mapping algorithm brought a massive improvement over the previous technique. For a total of 35 evaluated architectures, the performance simulation before our improvement took around 40 minutes since over 3000 mappings were estimated, whereas now it only takes around 30 seconds, which is 80 times faster. This is due to the fact that only one mapping par architecture is generated, i.e. in this case 35 mappings. For this case study, this brings the overall time taken by our tool for the DSE to a time under one minute to get the final implementation.

8 Conclusion

In this paper we have presented our template-based design framework that fits usual practices and real-life needs in embedded systems. It brings in the domain of FPGA, expected features such as flexibility, reliability and design reuse while using standard formalisms well-known from designers. It also targets several levels of expertise: the designer can choose between letting the framework make all the design exploration or relying on his own expertise by specifying extra-constraints. Additionally, we have detailed some important improvements of our H-MPSoC design framework. Firstly we have refined our communication model to accurately detect possible memory and bus contentions. Secondly we have significantly improved (x80) the speed of our mapping decision algorithm and get extremely fast DSE and consequently large solution exploration.

References

1. Oliveira, M., Brião, E., Francisco, A., Wagner, R.: Model driven engineering for MPSOC design space exploration. In: Proc. of the 20th Annual Conf. on Integrated Circuits and Systems Design, SBCCI 2007, pp. 81–86. ACM (2007)
2. Atitallah, R., Piel, E., Niar, S., Marquet, P., Dekeyser, J.: Multilevel MPSoC simulation using an MDE approach. In: IEEE Intl. SOC Conf. (2007)
3. Thompson, M., et al.: A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: 5th Conference on Hardware/Software Codesign and System Synthesis (2007)
4. Kahn, G.: The semantics of a simple language for parallel programming. Information processing 74, 471–475 (1974)
5. Coussy, P., et al.: GAUT: A High-Level Synthesis Tool for DSP applications. Springer (2008)
6. Nikolov, H., et al.: Systematic and automated multiprocessor system design, programming, and implementation. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 27(3), 542–555 (2008)

7. Verdoolaege, S., Nikolov, H., Stefanov, T.: Pn: a tool for improved derivation of process networks. EURASIP Journal on Embedded Systems 2007(1), 19 (2007)
8. Xilinx, OS and Libraries Document Collection (UG 643), `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/oslib_rm.pdf`
9. Corre, Y., et al.: A framework for high-level synthesis of heterogeneous mp-soc. In: Proc. of the Great Lakes Symp. on VLSI, pp. 283–286. ACM (2012)
10. Benkrid, K., Akoglu, A., Ling, C., Song, Y., Liu, Y., Tian, X.: High Performance Biological Pairwise Sequence Alignment: FPGA versus GPU versus Cell BE versus GPP. International Journal of Reconfigurable Computing (2012)
11. Feiler, P.H.: The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document (2006)
12. Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly 2(1-2), 83–97 (1955)

Configurable Fault-Tolerance for a Configurable VLIW Processor

Fakhar Anjam and Stephan Wong

Computer Engineering Laboratory,
Delft University of Technology,
Mekelweg 4, 2628CD, Delft, The Netherlands
{F.Anjam,J.S.S.M.Wong}@tudelft.nl

Abstract. This paper presents the design and implementation of configurable fault-tolerance techniques for a configurable VLIW processor. The processor can be configured for 2, 4, or 8 issue-slots with different types of execution functional units (FUs), and its instruction set architecture (ISA) is based on the VEX ISA. Separate techniques are employed to protect different modules of the processor from single event upsets (SEU) errors. Parity checking is utilized to detect errors in the instruction and data memories and the general register file (GR), while triple modular redundancy (TMR) approach is employed for all the synchronous flip-flops (FFs). At design-time, a user can choose between the standard non fault-tolerant design, a fault-tolerant design where the fault tolerance is permanently enabled, and a fault-tolerant design where the fault tolerance can be enabled and disabled at run-time. These options enable a user to trade-off between hardware resources, performance, and power consumption. A simulation based technique is utilized for testing purposes. The processor is implemented in a Xilinx Virtex-6 FPGA as well as synthesized to a 90 nm ASIC technology. Compared to the permanently enabled fault-tolerance, in scenarios, where fault-tolerance is not required at some point in time, considerable power savings (up to 25.93% for the FPGA and 70.22% for the ASIC) can be achieved by disabling the fault-tolerance at run-time.

Keywords: Softcore, VLIW processor, SEU error, Configurable fault-tolerance.

1 Introduction

Very long instruction word (VLIW) processors exploit instruction level parallelism (ILP) by means of a compiler. A VLIW processor can execute multiple operations (a long instruction) per cycle to increase performance [5]. When the data path of a processor gets larger and complex, the probability of errors (such as radiation-induced soft errors) also increases. Because VLIW processors can provide high performance at low power, they are gaining wide-spread utilization not only in general-purpose embedded systems but also in safety-critical systems such as biomedical, space, military, communication, industrial, and automotive systems. Therefore, it is important to employ fault-tolerant techniques in these processors for guaranteeing high reliability and dependability of the safety-critical systems. Run-time detection plays an important role in dependable systems, where it is needed that the computed data is either correct or an error signal is generated whenever there is a possible error.

In this paper, we present configurable fault-tolerant techniques for a softcore VLIW processor (ρ-VEX) [21]. The processor is parameterized and different parameters such as the issue-width, the number and types of different FUs, number of registers, memory buses, and latencies for the FUs can be chosen at design-time. The processor is implemented in VHDL and the fault-tolerance techniques are implemented at hardware level. The processor employs different fault-tolerance techniques such as parity checking and TMR to increase the reliability and dependability of the system. The processor is implemented in a Xilinx Virtex-6 FPGA as well as synthesized to a 90 nm ASIC technology.

Apart from the general parameters such as the issue-width, number of FUs, etc., the fault-tolerance is also configurable. At design-time, a user can choose to implement a processor with no fault-tolerance, a processor with the fault-tolerance permanently enabled, or run-time configurable. The permanently enabled and the run-time configurable designs consume almost similar dynamic power. The advantage of the latter design is that the fault-tolerance can be disabled at run-time, resulting in reduced dynamic power consumption (25.93%, 12.43%, and 5.55% for the FPGA and 65.92%, 67.30%, and 70.22% for the ASIC for the 2-, 4-, and 8-issue processors, respectively). The fault-tolerance can be enabled/disabled by executing an instruction on the processor. For applications which can tolerate some bit flips such as audio/video decoding, the fault-tolerance can be disabled at run-time to reduce the dynamic power consumption. On the other hand, applications which are susceptible to even a single bit flip such as sending/receiving DTMF tones on a mobile device, can enable the fault-tolerance at run-time. The configurable processor provides a trade-off for hardware resources, performance, and power consumption.

The remainder of the paper is organized as follows. Related work is discussed in Section 2. Section 3 briefly introduces our configurable ρ-VEX softcore VLIW processor. The fault-tolerant design of the ρ-VEX processor is presented in Section 4. Experimental results are presented in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Recently, fault-tolerance for microprocessor systems is gaining increasing importance. Transient errors are considered as the main source of errors in processor systems. Different on-line detection and mitigation techniques are proposed for detecting and correcting transient error faults. These techniques are mainly based on the redundancy approaches. In these techniques, instructions are replicated and computed, and then the results of the original and the duplicated instructions are compared to check for errors. Mainly, there are two approaches for redundancy; software-based and hardware-based.

A software-based redundancy approach utilizes a compiler to generate duplicate/triplicate instructions. This approach increases code size and power consumption and reduces performance [13]. The advantage is that no hardware modification is needed. Compiler-based software redundancy schemes with the effect of increased code size and performance degradation are presented in [3][10]. Similar techniques to detect errors in VLIW and superscalar processors are discussed in [9][2][14]. A software method to detect transient and common-mode faults in statically-scheduled VLIW processor in presented in [18].

A hardware-based redundancy approach requires changes to the architecture and additional hardware for managing replication, re-computation, and comparing results to detect errors. The advantage is that there is no need to change the code or the compiler, and that there is little or no performance degradation and no code size overhead. At the hardware level, one solution is to replicate the complete processor system, and then implement a majority voter to select between the three results [11][20]. In that case, there is no need to change the processor architecture, with the disadvantage that a fine-grain control over instruction-level checking is not possible. Another solution is to modify the architecture, implement additional FUs and other control hardware units to perform the execution of replicated instructions [7][15][16]. A technique that utilizes additional FUs to detect and correct transient errors generated in combinational logic is presented in [4]. The author in [8] triplicates the sequential elements in the processor to detect and correct SEU errors. Recently, hybrid approaches (software and hardware) for error detection and correction were presented in [4][17].

3 The Configurable ρ-VEX VLIW Processor

The VEX ISA, developed by the Hewlett-Packard (HP) and STMicroelectronics [6] is a 32-bit clustered VLIW ISA that is scalable and customizable to individual application domains. The ISA is loosely modeled on the ISA of the HP/ST Lx (ST200) family of VLIW embedded cores [5]. Based on *trace scheduling*, the VEX C compiler is a parameterized ISO/C89 compiler. A flexible programmable machine model determines the target architecture, which is provided as input to the compiler. The compiler reads a machine configuration file and schedules the code according to it. A VEX software toolchain including the VEX C compiler and the VEX simulator is made freely available by HP Laboratories [1].

The ρ-VEX is a configurable (design-time) open-source softcore VLIW processor [21]. The ISA is based on the VEX ISA [6]. Different parameters of the processor such as the issue-width, the number and type of different FUs, supported instructions, memory-bandwidth, register file size etc., can be chosen at design time. The processor is a 5-stage pipelined processor consisting of *fetch*, *decode*, *execute0*, *execute1/memory*, and *writeback* stages. Regular functional units include arithmetic logic units (*ALUs*), multiplier units (*MULs*), branch or control unit (*CTRL*), and load/store or memory unit (*MEM*). The fetch stage

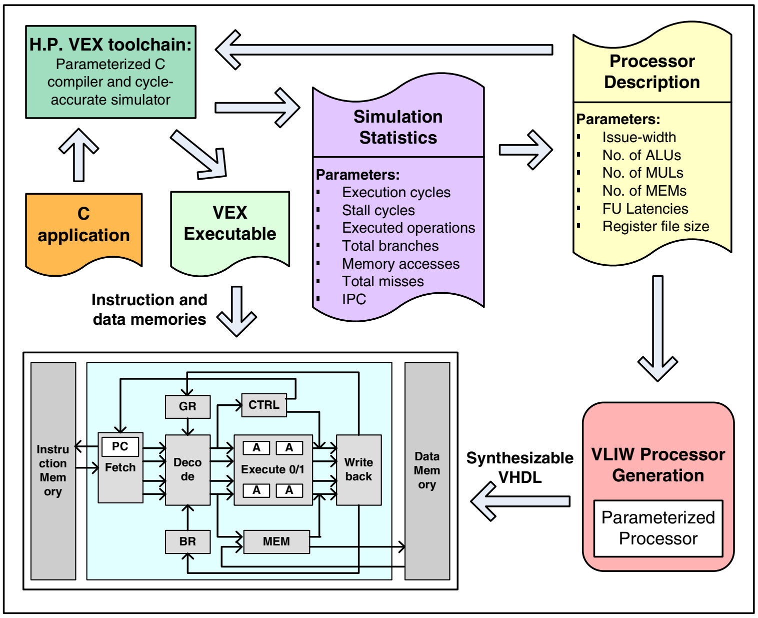


Fig. 1. Methodology to generate a ρ-VEX VLIW processor

fetches a VLIW instruction from the attached instruction memory, and splits it into syllables that are passed on to the decode stage. Here, instructions are decoded and register contents used as operands are fetched from the GR register file. Branch operations take place in the CTRL unit located in the decode stage. The ALU/MUL operations take place in either the execute0 (1-cycle latency) and execute1 (2-cycle latency) stages. Load/store operations takes place in the MEM units in the execute1/memory stage. All write activities are performed in the writeback stage to ensure that all targets are written back at the same time. The processor has a 64×32-bit multiported general register file (GR) and an 8×1-bit multiported branch register file (*BR*). The number of GR and BR registers can be changed at design time (maximum 64 for GR and 8 for BR). Figure 1 depicts the methodology to generate a ρ-VEX processor. The user can profile an application to determine a suitable processor, and then quickly implement it along with the executable (instruction and data memories). The only difference between the FPGA and ASIC implementations is that, for FPGA, the GR register file is implemented with dual-port synchronous RAMs (DP_RAMs), while for ASIC it is implemented with flip-flops (FFs).

4 The Fault-Tolerant ρ-VEX VLIW Processor

Single event upset (SEU) effects a memory cell or FF. It is a bit flip caused by a charged particle. The noise induced by a radiation when exceeds the threshold voltage, a bit flip may occur. Due to wire process shrinking, the threshold voltage is decreasing, and hence, electronic systems are becoming more susceptible to SEUs. When an SEU occurs in a memory (storage or configuration), it is called as *permanent error*. When it occurs in a flip-flop, it is referred to as a *transient error*. To recover from the permanent error, reconfiguration or re-loading of the configuration data to the configuration memory is required. For a memory used as a general storage (e.g., instruction memory), the permanent error could be checked and corrected by parity checking and some error correcting code (ECC). TMR technique is used widely to recover from a transient error. When TMR mitigation techniques are adopted, the same circuit is triplicated and a majority voter is implemented between the three computed results. In this way, a single fault occurring in one part of the TMR circuit is protected as the result is obtained from the other two circuits.

For this paper, we consider SEU errors that occur due to a direct hit in a flip-flop or a memory element used as a general storage (instruction and data memories, and the GR register file). We do not consider the FPGA configuration memory, and assume that it is protected by other techniques. According to [12], the probability that SEU errors in combinational logic can propagate to a register on a clock is very low, therefore, we do not consider such permanent SEU errors in combinational logic. The ρ-VEX processor utilizes two types of sequential cells for its implementation: synchronous DP_RAMs for instruction/data memories and the GR register file (in case of FPGA implementation), and synchronous FFs used for other storage such as general registers, pipeline registers, state machines, and status/control functions. We employ different SEU protection techniques for the DP_RAMs and FFs. The hardwired DP_RAMs in the Xilinx and Altera FPGAs provide an extra bit per byte of data which can be used as a parity bit. Hence, for a 32-bit word, up to 4 parity bits are available and can be used without increasing the number of DP_RAMs. In case of an ASIC, additional area is required to implement parity bits in instruction and data memories. Following, we discuss different modules of the fault-tolerant ρ-VEX processor which utilize different error protection techniques.

4.1 Instruction Memory

For the ρ-VEX processor, each operation called syllable is encoded in a 32-bit word. Multiple syllables are combined to make a long instruction which is executed every clock cycle. The instruction width for a 2-, 4-, or 8-issue ρ-VEX processor is 64-bit, 128-bit, and 256-bit, respectively. Our design provides configurable number of parity bits (1, 2, and 4) per 32-bit instruction (syllable). Hence, for every 8 bits of instruction, a parity bit is available. The parity bits are statically calculated by *XOR* operations in the assembler tool and stored along with the instructions in the dedicated parity bits of the memory. Instructions are read and passed through the fetch stage to the decode stage. The parity bits are checked in the decode stage in parallel with instruction decoding to minimize the timing overhead. If a parity error is detected for an instruction, the decode and the fetch stages are flushed, and the pipeline is halted. The correct instruction can then be copied from the higher level memory (Flash card, on-board memory, etc.) to the local instruction memory, and the pipeline can then be restarted.

4.2 Data Memory

The data width of the ρ-VEX processor is 32-bit whatever the issue-width may be. The data memory is implemented with DP_RAMs. Additional bits are utilized as parity bits. Because the ISA has memory operations that can operate on words, half-words, and bytes, therefore, we utilized 1 parity bit per byte of the data. Initially, parity bits are generated statically in the assembler tool and placed along with data in the external memory. During initialization, the data and the parity bits are copied from the external memory to the local data memory. During a store operation, the parity bits are calculated and written to the data memory together with the new data. The parity bits are generated in the MEM unit which resides in the execute0 stage. During a load operation, a data word is read from the data memory along with the parity bits. The parity of the data word is checked in the writeback stage before writing the word to the GR register file. If there is a parity error, a data error trap is generated and the pipeline is halted. The simplest method to recover from this error is to reload the whole data memory for the program from the external memory and start the program from the beginning. Other complex error recovery methods such as roll back to the instruction which modified the data location may also be considered but implementing such methods are out of scope of this paper.

4.3 GR Register File (FPGA Implementation)

The 2-, 4-, and 8-issue ρ-VEX processors require 64×32-bit GR register files with 2-write-4-read (2W4R) ports, 4W8R ports, and 8W16R ports, respectively. The hardware resource requirement for these register files grows large when implemented with an FPGA's configurable look-up tables (LUTs), therefore, the register files are implemented with 18 Kbits DP_RAMs. Each DP_RAM is configured in simple dual port (SDP) mode with 1W1R port, and for multiple ports, DP_RAMs are organized into multiple banks and data is duplicated across various DP_RAMs. In this design, the number of write ports defines the number of banks and the number of read ports defines the number of DP_RAMs per bank. Figure 2 depicts the GR register file for a 2-issue ρ-VEX processor with 2 banks each having 4 DP_RAMs. The *direction table* is a small register table having the same number of ports as the original register file. For the 2W4R, 4W8R, and 8W16R ports register files, the direction table is 64×1-bit, 64×2-bit, and 64×3-bit, respectively. The direction table is implemented with FFs, and to provide SEU error protection, TMR approach is utilized as discussed in Section 4.4. Each FF is triplicated and a majority voter is implemented. Hence, an SEU error in a single FF can be tolerated.

Each write port is associated with a bank and all the DP_RAMs in a bank are simultaneously updated. Each DP_RAM is organized in a 32-bit wide aspect ratio and parity bits are design-time configurable (1, 2, or 4 for each 32-bit word).

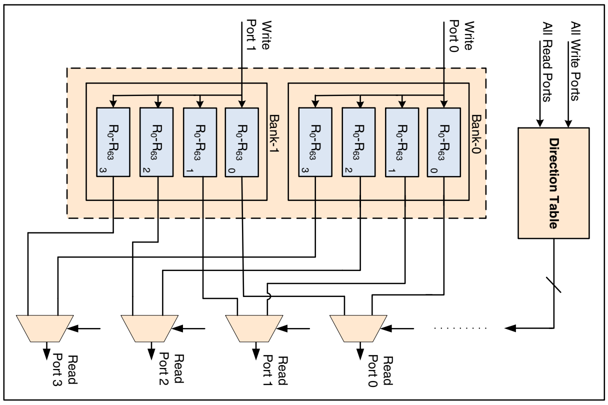


Fig. 2. 64×32-bit 2W4R ports GR register file for 2-issue ρ-VEX processor implemented with FPGA's DP_RAMs

The parity bits are generated in the writeback stage and written together with the data. The register data is accessed in the decode stage but the parity check is done in the execute0 stage to avoid the timing overhead. If a parity error is detected on the read data on a register file port, the pipeline is flushed and the error correction procedure is started. We implemented a simple mechanism to correct the corrupted data. For each write port, the written data is already duplicated in multiple DP_RAMs each associated with a read port (4, 8, and 16 DP_RAMs for 2-, 4-, and 8-issue processors, respectively). When a parity error is detected in a data on a read port, the same data is read on another port from a different DP_RAM in the same bank. The parity for this data is also checked. If the parity is correct, it is assumed that this data is correct. This data is then written to all the DP_RAMs in the bank where the corrupted data was present in a DP_RAM. The pipeline is then restarted at the point of the failing instruction and normal execution resumes. Currently, we check only one neighbor DP_RAM for the correct data instead of all the DP_RAMs in a bank to simplify the design. If a data word cannot be corrected in this way (e.g., if the same location in all the DP_RAMs in a bank is corrupted at the same time), an unrecoverable error trap is generated.

4.4 TMR Approach for Flip-Flops

In the ρ-VEX processor, flip-flops are used for different purposes such as data holding registers, status registers, pipelines latches/registers, state machine registers, etc. The VEX ISA specifies a 1-bit 8-element multiported branch register file (BR) for a multi-issue VLIW processor. For 2-, 4-, and 8-issue ρ-VEX processors, the ISA requires BR register files with 2W2R ports, 4W4R ports, and 8W8R ports, respectively. In case of ASIC implementation, the GR, BR, and link register (LR) files for the processors are implemented with FFs. TMR approach is utilized to protect against the SEU errors in all the FFs used in the processors. Each FF is triplicated and a majority voter is implemented for it, and hence, an SEU error in a single FF can be tolerated. Because the FFs are continuously clocked, any SEU error can be removed within one clock cycle with the output of the voter providing the correct (glitch-free) value. The robustness of the TMR scheme can further be increased by providing a separate clock tree for each lane of the TMR FFs. In this way, all of data errors resulting from an SEU hit on one clock tree can be tolerated and automatically corrected on the next clock edge.

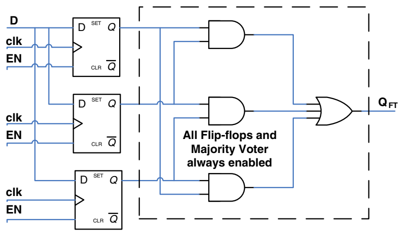
4.5 Working of the Configurable Fault-Tolerant System

We implemented fault-tolerance techniques that can be configured to be enabled or disabled at run-time. The user can specify to include or not include the fault-tolerance in the processors at design-time. Additional to this, the user can specify at design-time to implement a fault-tolerant design of a processor in which the fault-tolerance is permanently enabled. Figure 3(a) depicts the TMR scheme and the majority voter for the permanently enabled fault-tolerant design. If an application requires that fault-tolerance should always be enabled, this design has the advantage of requiring less hardware resources, consuming less dynamic power, and running at higher clock frequency.

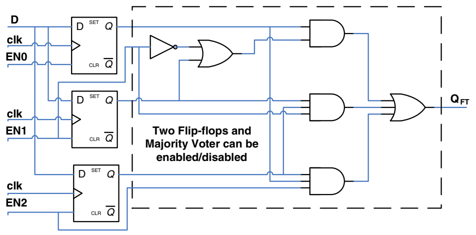
On the other hand, if an application requires fault-tolerance only at specific instances of time (e.g., to execute certain specific modules or when the device has to be used in an increased radiations environment) and does not require fault-tolerance at other times, the system should be able to turn off the fault-tolerance circuit to avoid consuming additional dynamic power due to triplication of the FFs. In our case, at design-time, the user can also specify to implement a processor in which the fault-tolerant circuit can be enabled and disabled at run-time. The fault-tolerance can be enabled and disabled be executing an instruction on the processor and this control can be given to the user or to a program. Figure 3(b) depicts the TMR scheme and the majority voter for the run-time enabled/disabled fault-tolerant design. In this case, the additional two FFs and the majority voter can be enabled/disabled by controlling the *EN1* and *EN2* signals. This design slightly increases the hardware resources and the critical path. The advantage is that dynamic power consumption can be reduced at run-time if an application does not require fault-tolerance at some point in time.

4.6 Test Methodology

To test our fault-tolerant design, we utilized the simulation-based fault-injection method presented in [19]. The advantage of this method is that it does not require any additional hardware setup and allows fast and easy implementation of the fault injection platform but limits the number of experiments due to its high computational requirements and long simulation time. Using the VHDL description of the ρ-VEX processor, we can perform realistic emulation of the faults and detailed monitoring of the system. We utilized *ModelSim simulation tool (version 64-bit SE 6.6e)* to perform fault injection and record the results.



(a) Permanently enabled scheme.



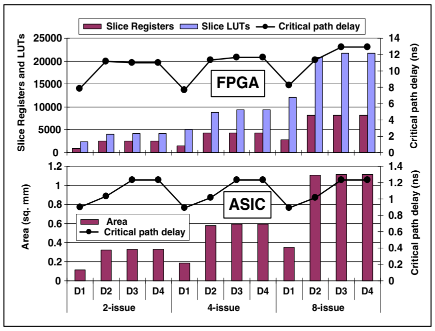
(b) Run-time enabled/disabled scheme.

Fig. 3. Two approaches used for TMR

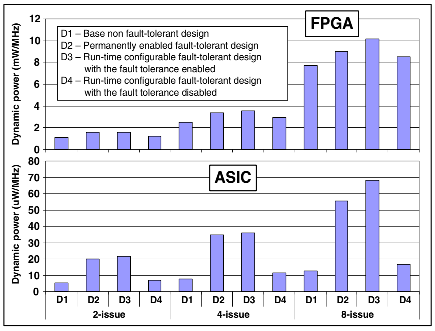
We have written special non-synthesizable routines that generate faults in different regions of the processor at different clock edges, and then record the results. Single bit errors are induced in the pipeline registers and other sequential elements of the processors. Injecting errors in FFs does not require stalling a processor and the execution can continue as normal. To inject errors in the GR register file (FPGA implementation) of a processor or the instruction or data memory requires stalling the processor. We injected 3000 single bit errors in the 2-, 4- and 8-issue ρ-VEX processors running *matrix multiplication* and *sorting* applications. All the errors in the TMR structure (FFs) were corrected. Errors in the instruction and data memories were detected and the processor was stopped to correct these errors. All of the correctable errors in the GR register file (FPGA implementation) were corrected and the non-correctable errors generated a trap halting the processor execution.

5 Experimental Results

Figure 4 depicts the hardware resources, critical path delay, and dynamic power consumption results for the base and the fault-tolerant ρ-VEX processors without instruction and data memories. For the FPGA implementation, we utilized the *Xilinx ISE (version 13.3)* and the Virtex-6 *XC6VLX240T-1FF1156* FPGA, whereas for the ASIC, we utilized the *Synopsis Design Compiler (version G-2012.06-SP2)* and targeted a 90 nm technology. The GR and BR register files in all cases are 64×32-bit and 8×1-bit, respectively. The 2-, 4-, and 8-issue cores have 2, 4, and 8 *ALUs*, and 2, 2, and 4 *MULs*, respectively. Each type of core has a single load/store (*MEM*) unit. In Fig. 4, *D1* and *D2* represent the base non fault-tolerant and the permanently enabled fault-tolerant designs, respectively. *D3* and *D4* (both having same area in terms of hardware resources) represent the processor design in which fault-tolerance can be enabled/disabled at run-time. *D3* represents the fault-tolerance enabled scenario, while *D4* represents the fault-tolerance disabled scenario. The parity bits are design-time configurable, i.e., 1, 2, or 4 bits per 32-bit of word. The results presented in Fig. 4 are for 4 bits of parity per 32-bit word, i.e., 1 bit per byte of data.



(a) Resources/area & critical path delay.



(b) Dynamic power consumption.

Fig. 4. Hardware resource utilization, critical path delay, and dynamic power consumption for the ρ-VEX VLIW processors for the Xilinx Virtex-6 FPGA and 90 nm ASIC technology. In addition to the mentioned resources, the 2-, 4-, and 8-issue cores utilize 4, 16, and 64 RAMB36s for GR register files, and 4, 4, and 8 DSP48E1s modules, respectively, in the FPGA implementation

As can be observed from Fig. 4(a), adding fault-tolerance to a processor requires more hardware resources especially the FFs (which are triplicated for TMR approach) and the additional logic gates for implementing majority voters. For the FPGA implementation, the number of DP_RAMs for the GR register files and instruction and data memories remain the same because we utilize the available extra parity bits in the DP_RAMs. For the ASIC, the area required for implementing additional parity bits for instruction and data memories increases. In terms of bits increase, it is 1, 2, or 4 bits per 32-bit of word for instruction and data memories depending upon the desired number of parity bits. Designs *D3*/*D4* utilize slightly more hardware resources and runs at less frequency compared to *D2*. The additional logic gates utilized for majority voters in *D3*/*D4* may be accommodated in the already utilized LUTs (FPGA implementation), therefore, the critical path delay remains almost the same as that for *D2*. It is also to note that the critical path delay in FPGAs is also dependent upon the placement and routing congestion. In case of the ASIC, the increase in the critical path delay can be clearly observed when moving from *D1* to *D2* to *D3*/*D4* due to adding additional logic gates in the path (majority voters).

Instead of measuring the absolute power consumption for certain applications, we measure the average dynamic power at typical operating conditions utilizing 10% switching activities, as presented in Fig. 4(b). We utilized the Xilinx *XPower Analyzer* tool and the *Synopsis Design Compiler* to measure the power consumption for the *XC6VLX240T-1FF1156* FPGA and the 90 nm technology, respectively. As can be observed from the figure, implementing fault-tolerance in the processors increases the dynamic power consumption due to increased hardware resources. Designs *D2* and *D3* (fault-tolerance enabled) consume almost similar dynamic power, while *D4* (fault-tolerance disabled) consumes considerably less power compared to *D2*. In case of the FPGA implementation, the *D4* consumes 25.93%, 12.43%, and 5.55% less dynamic power compared to the 2-, 4-, and 8-issue *D2* designs, respectively. In the larger issue-width cores, the GR register files require increased number of DP_RAMs due to increased number of ports. In FPGAs, DP_RAMs contribute more to the dynamic power compared to FFs, therefore, for the 8-issue processors in our case, the dynamic power consumption does not reduce considerably when moving from *D2* to *D4*. This effect is not visible in the ASIC results, as the GR register files are implemented with FFs, not DP_RAMs. For the ASIC implementation, the *D4* consumes 65.92%, 67.30%, and 70.22% less dynamic power compared to the 2-, 4-, and 8-issue *D2* designs, respectively. This is considerable power reduction, and if fault-tolerance is not required at some point in time, it can be turned off to reduce the dynamic power consumption.

6 Conclusions

In this paper, we presented configurable fault-tolerance techniques for the ρ-VEX softcore VLIW processor. The issue-width of the processor can be configured to be 2-issue, 4-issue, or 8-issue with different mix of functional units. The fault-tolerance designs can detect and correct SEU errors. The designs are implemented in a Xilinx Virtex-6 FPGA, as well as synthesized to a 90 nm ASIC technology. Parity checking is utilized to detect errors in the instruction and data memories, and the general register files (FPGA implementation). For all other sequential elements, the TMR technique with majority voting is implemented. Different designs for fault-tolerance scheme such as permanently enabled at design-time or with run-time options for enabling and disabling, were presented. These options enable a user to trade-off between hardware resources, performance, and power consumption. When fault-tolerance is not required at some point in time, disabling the fault-tolerance at run-time results in considerable dynamic power reduction (up to 25.93% for the FPGA and 70.22% for the ASIC) compared to the permanently enabled fault-tolerant design.

References

1. H.P. Labs. VEX Toolchain, `http://www.hpl.hp.com/downloads/vex/`
2. Blough, D., Nicolaun, A.: Fault Tolerance in Super-scalar and VLIW Processors. In: IEEE Workshop on Fault Tolerant Parallel and Distributed Systems, pp. 193–200 (1992)
3. Bolchini, C.: A Software Methodology for Detecting Hardware Faults in VLIW Data Paths. IEEE Transactions on Reliability 52(4), 458–468 (2003)
4. Chen, Y., Leo, K.: Reliable Data Path Design of VLIW Processor Cores with Comprehensive Error-coverage Assessment. Elsevier Journal of Microprocessors and Microsystems 34, 49–61 (2010)
5. Faraboschi, P., Brown, G., Fisher, J., Desoli, G., Homewood, F.: Lx: A Technology Platform for Customizable VLIW Embedded Processing. In: International Symposium on Computer Architecture, pp. 203–213 (2000)

6. Fisher, J., Faraboschi, P., Young, C.: Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools. Morgan Kaufmann (2005)
7. Franklin, M.: A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors. In: International Workshop on Defect and Fault Tolerance in VLSI Systems, pp. 207–215 (1995)
8. Gaisler, J.: A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture. In: International Conference on Dependable Systems and Networks, pp. 409–415 (2002)
9. Holm, J., Banerjee, P.: Low Cost Concurrent Error Detection in a VLIW Architecture using Replicated Instructions. In: International Conference on Parallel Processing, pp. 192–195 (1992)
10. Hu, J., Li, F., Degalahal, V., Kandemir, M., Vijaykrishnan, N., Irwin, M.: Compiler-Directed Instruction Duplication for Soft Error Detection. In: Design, Automation and Test in Europe Conference and Exhibition, pp. 1056–1057 (2005)
11. Ichinomiya, Y., Tanoue, S., Ishida, T., Amagasaki, M., Kuga, M., Sueyoshi, T.: Memory Sharing Approach for TMR Softcore Processor. In: Becker, J., Woods, R., Athanas, P., Morgan, F. (eds.) ARC 2009. LNCS, vol. 5453, pp. 268–274. Springer, Heidelberg (2009)
12. Liden, P., Dahlgren, P., Johansson, R., Karlsson, J.: On Latching Probability of Particles Induced Transients in Combinational Networks. In: International Symposium on Fault-Tolerant Computing, pp. 340–349 (1994)
13. Nickle, J., Soman, A.: REESE: A Method of Soft Error Detection in Microprocessors. In: International Conference on Dependable Systems and Networks, pp. 401–410 (2001)
14. Oh, N., Shirvani, P., McCluskey, E.: Error Detection by Duplicated Instructions in Super-scalar Processors. IEEE Transactions on Reliability 51(1), 63–75 (2002)
15. Rashid, F., Saluja, K., Ramanathan, P.: Fault Tolerance Through Re-execution in Multiscalar Architecture. In: International Conference on Dependable Systems and Networks, pp. 482–491 (2000)
16. Sato, T., Arita, I.: Evaluating Low-cost Fault-tolerance Mechanism for Microprocessors on Multimedia Applications. In: Pacific Rim International Symposium On Dependable Computing, pp. 225–232 (2001)
17. Scholzel, M., Mulleri, S.: Combining Hardware- and Software-Based Self-Repair Methods for Statistically Scheduled Data Paths. In: International Workshop on Defect and Fault Tolerance in VLSI Systems, pp. 90–98 (2010)
18. Sterpone, L., Sabena, D., Campagna, S., Reorda, M.: Fault Injection Analysis of Transient Faults in Clustered VLIW Processors. In: IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, pp. 207–212 (2011)
19. Touloupis, E., Flint, J., Chouliaras, V., Ward, D.: Study of the Effects of SEU-Induced Faults on a Pipeline-Protected Microprocessor. IEEE Transactions on Computers 56(12), 1585–1596 (2007)
20. Vasudevan, V., Waldeck, P., Mehta, H., Bergmann, N.: Implementation of Triple Modular Redundant FPGA based Safety Critical System for Reliable Software Execution. In: Australian Workshop on Safety Related Programmable Systems, pp. 113–119 (2006)
21. Wong, S., Anjam, F.: The Delft Reconfigurable VLIW Processor. In: International Conference on Advanced Computing and Communications, pp. 242–251 (2009)

Hierarchical and Multiple Switching NoC with Floorplan Based Adaptability

Debora Matos[1], Cezar Reinbrecht[1], Marcio Kreutz[2], Gianluca Palermo[3], Luigi Carro[1], and Altamiro Susin[1]

[1] Federal University of Rio Grande do Sul – UFRGS, Porto Alegre, Brazil
{debora.matos,cezar.reinbrecht,carro,susin}@ufrgs.br
[2] University of Rio Grande do Norte - UFRN, Natal, Brazil
kreutz@dimap.ufrn.br
[3] Politecnico di Milano – Dipartimento di Elettronica e Informazione, Milano, Italy
gpalermo@elet.polimi.it

Abstract. The Networks-on-Chip paradigm has been seen as an interconnect architecture solution for complex systems. However, performance and energy issues still represent limiting factors for Multi-Processors System-on-Chip. Moreover, the execution of different applications requires flexible and transparent interconnection solutions, and this feature is best provided by a self-adaptable system. In this paper we propose HASIN, an architecture that explores the suitable switching architecture according to the traffic in each region of the system, in a hierarchical manner. The proposed interconnection allows adapting the network at runtime using three switching possibilities to reconfigure itself according to the floorplan information. HASIN allows increasing the throughput up to 77% and reducing the power consumption up to 76% when compared to a packet-switched mesh network-on-chip.

Keywords: NoC, Hierarchy, Adaptability Switching, Circuit Switching.

1 Introduction

Technology scaling has allowed a large integration capacity. In such context, a single chip can be composed by many processing elements (PEs), the called Multi-Processors System-on-Chip (MPSoCs) [1]. Several heterogeneous elements can integrate these systems, presenting different bandwidths and quality-of-service (QoS) requirements [2].

Over the past years, Network-on-Chip (NoC) designs have been studied as an appropriate solution for such complex hardware systems due to their scalability, parallelism and QoS [1, 7]. However, as the complexity of current and future MPSoC designs increases at a fast pace, current systems are requiring other interconnection alternatives.

One current requirement of complex systems is to provide different levels of communications in a hierarchical manner since, in heterogeneous systems, there are regions with specific requirements in terms of bandwidth [3, 8]. In this work we propose the use of a hierarchical interconnection in order to explore the communication locality, while ensuring the communication rates required by the processing elements. The clusters of our hierarchical NoC were implemented by crossbar switches and each one can be designed with a specific number of ports. However, the use of this approach is not suitable for systems that can present changes at runtime in the system communication pattern. Consequently, in this scenario, the network-on-chip must also be able to adapt itself. As one solution for this problem, we integrated on the global level of the hierarchy, the possibility to adapt at runtime the switching mechanism.

In this paper we propose a new architectural concept called HASIN (Hierarchical Adaptive Switching Interconnection Network). This proposal combines both concepts of hierarchical topology organization and adaptive switching. The HASIN network topology uses crossbars switch in the local level and a mesh topology in the global level. On the global level, our strategy allows three switching configurations: packet switching (PS), buffered circuit switching (BCS) and unbuffered circuit switching (UCS). The definition of the switching mode considers the current status of the network and it does not use the conventional initial setup, as proposed by other architectures. Moreover, this adaptability is dynamic and considers the floorplan parameters to define the circuit-switching mode.

Many recent proposed routers show a complex architecture with virtual channels (VCs), tables, and expensive controls in order to increase the interconnection performance [4-7]. However, with the addition of such resources, the router power consumption could be impracticable for the embedded domain [3]. In our proposal the power consumption is reduced since small crossbar switches are used to compose the clusters. As the crossbars present a simple architecture and do not require buffers, the power consumption is much smaller than a conventional router architecture. This hierarchical NoC topology not only reduces the number of hops and explores the communication locality, but is also able to provide low power consumption. Our proposal still differs from the others since HASIN allows three switching possibilities that are dynamically reconfigurable to avoid long interconnections.

2 HASIN Architecture

Thanks to its hierarchical approach, HASIN can cope with specific communication behaviors. However, it also presents flexible features to support different traffic patterns. An example of the HASIN topology is illustrated in Fig. 1.

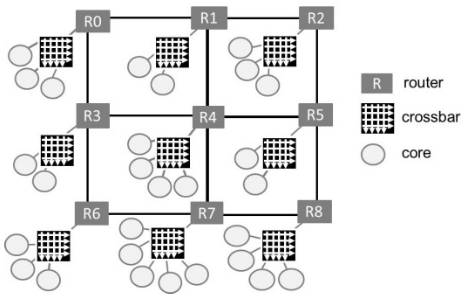


Fig. 1. HASIN topology example composed with 28 cores

HASIN local level is basically a crossbar interconnected in the local channel of a NoC router. In this case, each class of applications needs to have an appropriate mapping and crossbar granularity to compose the clusters. The HASIN architecture is composed by three main elements, the adaptive router, the crossbar switch - SWIX (Switching Interconnection Crossbar) and the bridges that interconnect the two hierarchical levels.

On the top level, HASIN presents configurable mesh routers. The router architecture is illustrated in Fig. 2. Each router port is composed by *Input Channel*, *Output Channel* and *Operation Mode Controller*. There is a multiplexer to select if the data will be send by packet, unbuffered or buffered circuit. Each router port uses a control called *Long Link Controller* (LLC) to identify the wire length between two routers according to the floorplan information. This mechanism uses the knowledge of the wire length from the first router defined as UCS mode until the last router crossed over. This information is stored in a specific field of the packet header, generating total wire length estimation. Then, the control checks if the estimated delay for this total wire length is higher than the clock period, indicating the need to change the UCS to BCS mode.

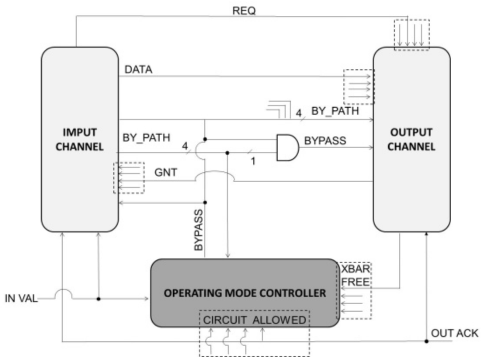


Fig. 2. HASIN Router Architecture

Operation Mode Controller (OMC) is responsible to define the packet or circuit switching modes, managing the *bypass* enable signal of each input port. The *OMC* architecture takes into account three issues: if the input channel has flits to receive (*in_val*), if the selected arbiter is free (*xbar_free*) and if the input port of the destination router can receive data (*circuit_allowed*). Therefore, if the conditions above are met, the circuit switching mode can be enabled. However, there are some more considerations in the CS enable managing. The first consideration is related to the fact that the router needs one cycle to define the output port destination, which makes it impossible to know whether the crossbar is free or not. To deal with this, our router assumes all output ports are initially free. In this case, the incoming data through the circuit path are accepted, applying a speculative strategy. If this situation is true, the bypass continues set. Otherwise, the bypass is deactivated, and the multiplexer from input port receives the data. The first flit is not lost because it was previously stored in a buffer. There is no latency penalty in a prediction mistake, because it uses an internal buffer to recover the data in time to use in a packet switching flow.

The SWIX architecture aims at allowing the communication among the cores by multiplexer switches. Parallel communications can occur if the data are sent for different output ports. However, if there is a conflict for the same port, a Round Robin (RR) arbiter is used to avoid starvation. A handshake flow control is applied for each port, by the requisition signal (*REQ*) and the granted signal (*GNT*). The control unit is composed by RR arbiter which takes only two cycles to verify the requisition of each port, independently of the crossbar switch size. As only small crossbars are used in this topology, the maximum operating frequency is guaranteed.

Bridges are used in the connection between the global and local levels, leaving this boundary transparent for each hierarchical level. The reader flit contains the information of the XY routing algorithm, the local header (with the identification of the SWIX destination) and the information of long links used by the adaptive router. The bridges adapt the handshake protocol between SWIX and router and vice versa. When a message arrives in a cluster, the global reader is removed and only the SWIX destination identification is considered to transmit the message to the destination core.

3 Experimental Results

The performance results were obtained with a cycle-accurate traffic simulator described in Java. We have considered two benchmarks for these experiments: TVOPD [9], and the NCS [10]. Besides, we considered a tool to define the appropriate mapping for each application considering the floorplanning for the hierarchical architecture [11].

Simulations were performed for NoC with 16-bit link wide, 4-flit deep buffers, 5-flit packets for a total of 100 packets/ node and considering the operation frequency equal to 1GHz for all experiments. As the cores are heterogeneous, the results were obtained increasing the injection rate of the communications in the same proportion. Thus, the injection rate plotted in the graphs take into account the core with higher communication rate of each application. Analyses of long wires interconnections to define the circuit switching mode were considered and, in the average, the BCS is set to each 2 routers.

Throughput results are depicted in Fig. 3 for the NCS and TVOPD benchmarks. This figure shows the comparison results among HASIN, a conventional mesh topology with 5 pipeline stages and a network with the same adaptive router switching, but without the use of the hierarchical topology.

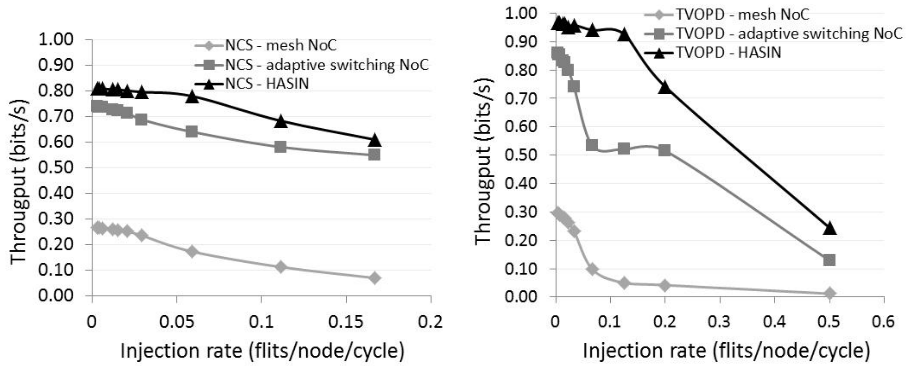


Fig. 3. Throughput results comparison for NCS and TVOPD benchmark

For these experiments, HASIN allows to increase the throughput up to 77% for NCS benchmark and up to 75% for TVOPD benchmark, when compared with a conventional mesh topology. When it is compared with the adaptive swiching NoC, the throughput is increased up to 18% for NCS and up to 44% for TVOPD. As expected, the hierarchical adaptive switching NoC presents much better results than a mesh NoC. These gains are possible thanks to the communication locality allowed by the hierarchical NoC combined with a suitable mapping for the application and by the dynamic adaptability in the top level. As we can observe, only the use of the adaptive switching strategy is not enough to reach these optimal results. It is clear the use of different techniques need to be considered to increase the throughput.

Synthesis results for 65nm of process technology were analyzed for the complete system of each application considered in this paper. In order to obtain accurate link lengths, we have considered the core areas as black boxes in the synthesis. In this case, the correct wire information and architectural costs were considered in the logical synthesis (obtained with the RTL Compiler tool) from the parasitic extraction in the physical synthesis (obtained with the First Encounter tool).

Average power and area results are presented in Table 1. According to these synthesis results it is possible to observe a large reduction in the power dissipation when compared to a conventional mesh NoC or to a NoC with adaptive switching strategy. In this comparison, the packet switching mode of the different architectures has the same number of pipeline stages.

The main gains of our strategy are obtained from the adaptive NoC combined with a very low cost architecture, like the use of SWIXs and smart routers to compose the hierarchical proposal. Moreover, with the proposed topology, the performance of the system is improved, since when the cores are in the same cluster, the latency to transfer the packets for many routers is removed, and, in this case, a simple switch protocol is used.

Table 1. Area and Power comparison results for the NCS and VOPD applications

		NCS	**TVOPD**
Average Power (mW)	mesh NoC	58.08	173.44
	adaptive switching NoC	92.87	259.59
	HASIN	15.17	41.28
Power reduction (%)	HASIN x mesh NoC	73.88	76.20
	HASIN x adaptive switching NoC	83.67	84.10
Area (mm^2)	mesh NoC	0.56	1.62
	adaptive switching NoC	0.57	1.64
	HASIN	0.12	0.32
Area reduction (%)	HASIN x mesh NoC	78.41	80.04
	HASIN x adaptive switching NoC	78.65	80.26

4 Conclusions

In this work we have shown the HASIN architecture which explores hierarchy with adaptability. The hierarchy is composed by routers and crossbar switches. The cluster performance is increased thanks the direct communications among the cores and the reduction in power is obtained due to the use of a simple cluster architecture crossbar-based. The relevant consideration of our strategy is that both application mapping and switching mode selection consider floorplan information. The adaptability is provided at the top level, where a NoC mesh is used, supporting three dynamic switching modes. Two of these modes were defined for the circuit switching (buffered or unbuffered) and the possibility to use these strategies together is new in the literature. The appropriate selection of each mode takes onto account the long link interconnections. In our design, the resources are widely reused in the router, reducing the extra circuitry to a minimum, but with extensive configuration possibilities. With this architecture was possible to obtain a large improvement in throughput and reduction in power dissipation when compared with other NoC architectures.

References

1. Silvano, C., et al.: Low Power Networks-on-Chip, 1st edn., p. 300. Springer (2011)
2. Stensgaard, M., Sparso, J.: ReNoC: A Network-on-Chip Architecture with Reconfigurable Topology. In: NoCS, pp. 55–64 (2008)
3. Das, R., Eachempati, S., et al.: Design and Evaluation of a Hierarchical On-Chip Interconnect for Next-Generation CMPs. In: HPCA, pp. 175–186 (2009)
4. Modarressi, M., et al.: Virtual Point-to-Point Connections for NoCs. TCAD 29(6), 855–868 (2010)
5. Jerger, N., et al.: Circuit-Switched Coherence. In: NoCS, pp. 193–202 (2008)
6. Modarressi, M., et al.: A Hybrid Packet-Circuit Switched On-Chip Network Based on SDM. In: DATE, pp. 566–569 (2009)
7. Yoon, Y., et al.: Virtual Channels vs. Multiple Physical Networks: a Comparative Analysis. In: DAC, pp. 162–165 (2010)
8. Chou, S.-H., et al.: Hierarchical Circuit-Switched NoC for Multicore Video Proc-essing. In: Microprocess. Microsyst. (2010)
9. Murali, S., et al.: Synthesis of networks on chips for 3D systems on chips. In: ASP-DAC, pp. 242–247 (2009)
10. Tino, A., Khan, G.: Power and Performance Tabu Search Based Multicore Network-on-Chip Design. In: Intl.Conf. on Parallel Processing, pp. 74–81 (2010)
11. Matos, D., et al.: Floorplanning-Aware Design Space Exploration for Application-Specific Hierarchical Network-on-Chip. In: NoCARC (2011)

HTR: On-Chip Hardware Task Relocation for Partially Reconfigurable FPGAs

Aurelio Morales-Villanueva and Ann Gordon-Ross

NSF Center for High-Performance Reconfigurable Computing (CHREC)
Dept. of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611
{morales,ann}@chrec.org

Abstract. Partial reconfiguration (PR) enables shared FPGA systems to non-intrusively time multiplex hardware tasks in partially reconfigurable regions (PRRs). To fully exploit PR, higher priority tasks should preempt lower priority tasks and preempted tasks should resume execution in any PRR. This preemption/resumption requires saving/restoring the preempted task's execution context and relocating the task to another PRR, however, prior works only provide partial solutions and impose limitations and/or overheads. We propose on-chip hardware task relocation (HTR) software, which enables a task's execution state to be saved, relocated to, and restored in *any* PRR with sufficient resources. The HTR software executes on a soft-core processor in the FPGA's static region, and is thus portable across any system/application. Experimental results evaluate HTR execution times, enabling designers to tradeoff task/PRR granularity and HTR execution times based on application requirements.

1 Introduction

Partial reconfiguration (PR) of FPGAs improves a shared system's functionality and performance via enhanced, fine-grained device reconfigurability and hardware multiplexing. The FPGA's fabric is partitioned into one static region and multiple partially reconfigurable regions (PRRs). Hardware tasks can be *scheduled* to execute in any PRR with sufficient resources—any *candidate* PRR—and if the *scheduled* PRR is executing a lower priority task, task preemption/resumption enables the lower priority task's execution state—*context*—to be paused (i.e., context save (CS)) and resumed (i.e., context restore (CR)) in another PRR. CS reads the task's execution state and saves the context to a CS bitstream, and CR merges the CS bitstream with the task's initial partial bitstream (created at synthesis) using bitstream manipulations and reconfigures the scheduled PRR with this merged bitstream.

There exists little prior work on CS and CR—context save and restore (CSR), collectively—to the same PRR [8][9], which forces a preempted task to resume execution in *only* the task's originally scheduled PRR, rather than *any* candidate PRR. Hardware task relocation (HTR) enables preempted tasks to be relocated and resumed in any candidate PRR, which can improve system performance, task throughput, and maximizes device resource utilization for application domains such as target tracking, dynamic load balancing, shared servers, etc. Since HTR is more challenging than CSR and must consider the task's physical relocation on the fabric, prior CSR work is not directly applicable. HTR is relatively easy between homogenous PRRs—PRRs with the same size, shape, and resources, but different fabric locations—requiring simple bitstream manipulations to specify the new fabric location [11][14]. HTR between heterogeneous PRRs—PRRs with different sizes, shapes, resources, and/or fabric locations—is more challenging, requiring complex bitstream manipulations to specify the new fabric location and relocate the task's functionality to this location's resources and/or resource layout. Similar to HTR, bitstream (core/module) relocation (BR) [1][2][3][4][5][7][16][18] enables a task to be relocated to any PRR, however, BR does not save/restore/resume the task's execution state, thus requiring the task to be restarted, which may incur seconds/minutes/hours of re-execution.

HTR can be implemented either off- or on-chip. In off-chip HTR, an attached CPU executes HTR software, which incurs significant overhead due to lengthy communication delays between the CPU and FPGA. Alternatively, on-chip HTR hardware can eliminate off-chip communication overhead, but introduces device resource overhead, lacks task/system portability, and reduces the tasks' maximum operating frequencies. To alleviate these overheads, we propose on-chip HTR software for heterogeneous PRRs that executes on a soft-core processor in the FPGA's static region, which, as compared to prior work, eliminates off-chip communication overhead and PRR overhead/constraints, is application/system independent, and does not alter the application/system design flow. Our HTR software uses the FPGA's internal configuration access port (ICAP) for reconfiguration. We detail HTR constructs and methodologies, which enables designers to incorporate HTR into their systems, and present implementation results for a Virtex-5 LX110T with a MicroBlaze (we note that the fundamentals of our HTR software is portable to newer Xilinx device families). Results show that HTR execution times are on the order of milliseconds, and vary based on the tasks'/PRRs' sizes. These analyses enable designers to tradeoff HTR execution times and task/PRR granularity based on application requirements.

2 Related Work

There exists little prior work in CSR, of which few leverage PR. Landaker et al. [15] and Simmler et al. [17] presented off-chip CSR software but since these works did not leverage PR, CSR reconfigured the entire FPGA. Joswik et al. [9] presented off-chip CSR software for PR FPGAs and reduced CSR times using direct memory access (DMA) for the ICAP, but this work only performed CSR to the same PRR. Kalte et al. [11] and Koester et al. [14] augmented the off-chip CSR software to include on-chip custom hardware for relocating tasks to different, homogeneous PRRs, however, both methods were for one-dimensional PR on older Xilinx devices, and are not applicable to newer Xilinx devices that support two-dimensional PR.

Koch et al. [13] and Jovanovic et al. [8] eliminated off-chip communication overhead with on-chip CSR hardware for both non-PR [13] and PR FPGAs [8], and reduced CSR times using different versions of scan-path chains of flip-flops (FFs), which is a technique used in design for testability (DFT) for very large scale integrated (VLSI) circuits. However, the CSR hardware incurred device overhead, lacked portability, reduced the system's maximum operating frequency, and required changes in the design tool flow. On-chip CSR software would alleviate these drawbacks, but would not include task relocation.

BR enables task relocation, but prior works did not relocate the task's context. Horta et al. [7] and Blodget et al. [3] presented off-chip BR software and Kalte et al. [10][12] presented on-chip BR hardware for homogeneous PRRs, however, these methods still incurred the same drawbacks as off- and on-chip CSR, respectively.

Becker et al. [1][2] and Carver et al. [4] presented on-chip BR software for heterogeneous and homogeneous PRRs, respectively, however, these methods constrained the static region's logic routing from passing through the PRRs. Where as this constraint reduced the number of partial bitstreams to one per task, as opposed to one partial bitstream for each task-to-PRR mapping, the constraint introduced area and performance overheads [4][6]. Corbetta et al. [5], Sudarsanam et al. [18], and Santambrogio et al. [16] presented custom on-chip BR hardware for homogeneous PRRs, which was orchestrated using an on-chip soft-core processor.

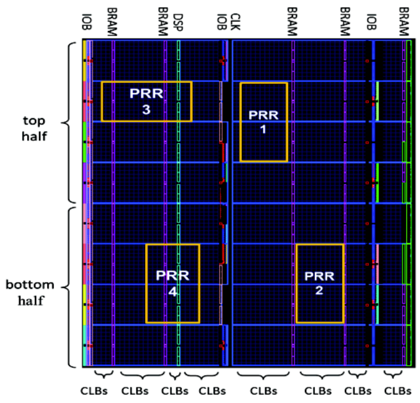


Fig. 1. Virtex-5 LX110T FPGA fabric layout

3 Virtex-5 FPGA Architecture

Since CSR and HTR are complex processes that require detailed device knowledge, we review the Xilinx Virtex-5 FPGA architecture (complete details are available in [19]), which will assist designers in incorporating HTR into their systems.

3.1 Device Architecture

Fig. 1 depicts the Virtex-5 LX110T fabric layout, the device used in our experiments, with four sample PRRs: PRR1 and PRR2 are homogeneous and PRR3 and PRR4 are heterogeneous. The device supports two-dimensional PR, which allows PRRs to occupy a rectangular fabric area. Device resources (CLBs, BRAMs, IOBs, DSPs, CLK) are distributed in a row/column organization. The device is logically divided into two halves—top and bottom—and each half contains four rows and each row contains the same number of columns. Columns contain groups of frames and the number of frames per column depends on the type of resource in that column. A frame is the minimum unit of information used to write/read to/from the device, and a Virtex-5 frame contains 41 32-bit words.

3.2 Device Configuration

The Virtex-5 can be configured using external interfaces, such as JTAG (serial) or SelectMAP (parallel), or the internal ICAP interface (parallel). Full or partial bitstreams configure the entire device or a single PRR, respectively. The bitstream's configuration information is organized in configuration frames and is stored in the FPGA's internal configuration memory. A configuration frame establishes a particular column's resource configuration and the routing information to access the resources. CLB, BRAM, DSP, IOB, and CLK columns have 36, 30, 28, 54, and 4 configuration frames, respectively [19].

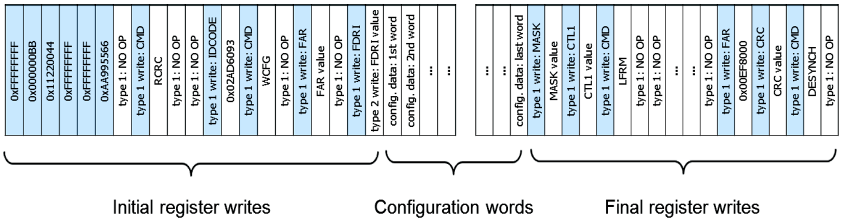


Fig. 2. Initial partial bitstream used in HTR for Virtex-5 FPGAs

Since HTR uses the ICAP, all partial bitstreams must be 32-bit word aligned. Fig. 2 depicts the initial partial bitstream structure used in HTR for the Virtex-5, which is the same as the bitstream generated by the Xilinx tools except that the initial comments (the name of the native circuit description file (*.ncd) from which the bitstream was generated and the bitstream creation date) are removed, resulting in a 32-bit word aligned file that can be used with the ICAP. The initial partial bitstream consists of a sequence of initial register writes, including the bus width words (0x000000BB and 0x11220044), the synchronization word (0xAA995566), RCRC, IDCODE (0x02AD6093), WCFG, FAR (specifies the first frame address of a PRR), and FDRI, followed by the configuration words (number of which is specified by the FDRI), and ending with the final register writes, which include MASK, CTL1, LFRM, CRC, and DESYNCH. [19] contains a complete description of these commands and special words. Note that the FAR included in the final register writes (0x00EF8000) is not associated with any PRR and is specific for the Virtex-5 LX110T.

For CS, the type 1 registers COR0 and CMD GCAPTURE are sent to the device via the ICAP to capture the FFs' values on a single edge transition of the main clock. After capturing the PRR's FFs' values, CMD RCAP is sent via the ICAP to enable future CSs [19]. CR requires initializing the PRR's FFs' values with the saved FFs' values without interrupting the static region or the other PRRs' execution. In order to initialize a PRR with new FF values, the internal global set reset (GSR) signal in the Xilinx user primitive STARTUP_VIRTEX5 [19] must be toggled, which forces the startup sequence [19]. Since this toggle would re-initialize the entire device with the initial values defined in the full bitstream, a protection/unprotection mechanism must be provided. A PRR/FPGA can be protected using the block type '010' and a special frame, sent to all PRR/FPGA columns [19]. Protecting the entire FPGA only needs to be done once, while unprotection/protection of the PRRs is required for each CR.

4 On-Chip Hardware Task Relocation (HTR) Software

Since the main contribution of our work is the HTR software, we assume that prior to execution, the applications have already been synthesized and partitioned into hardware tasks, the PRRs and soft-core processor have been created, the system contains a scheduler that maps and schedules incoming tasks to PRRs, and all full and initial partial bitstreams, including all task and candidate PRR combinations, and necessary files have been generated. We refer to a task executing in a PRR as a PR module (PRM). Even though a PRR may contain a mixture of resources, we detail HTR for PRMs that use CLBs only, however, our HTR is fundamentally applicable to heterogeneous PRRs that contain BRAMs, DSPs, and/or IOBs not in use by the PRM.

4.1 HTR Overview

We explain HTR using two heterogeneous PRRs and three PRMs: PRR1 is a candidate PRR for PRM1 and PRM2, and PRR2 is a candidate PRR for PRM2 and PRM3. Fig. 3 depicts the CSR and HTR flows (for resumption to the same or different PRR, respectively) assuming that PRM2 has already executed in PRR1, PRM2 was preempted and PRM2's context was saved, PRM3 is currently executing in PRR2, and PRR1 is ready to execute in PRM1. T_x denotes each step's execution time.

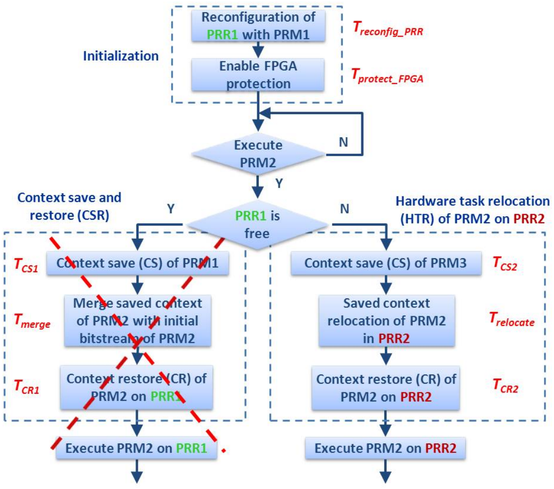

Initialization
Reconfiguration of PRR1 with PRM1
$T_{reconfig\_PRR}$
Enable FPGA protection
$T_{protect\_FPGA}$
Execute PRM2
N
Y
Context save and restore (CSR)
Y
PRR1 is free
N
Hardware task relocation (HTR) of PRM2 on PRR2
T_{CS1}
Context save (CS) of PRM1
T_{merge}
Merge saved context of PRM2 with initial bitstream of PRM2
T_{CR1}
Context restore (CR) of PRM2 on PRR1
Execute PRM2 on PRR1
Context save (CS) of PRM3
T_{CS2}
Saved context relocation of PRM2 in PRR2
$T_{relocate}$
Context restore (CR) of PRM2 on PRR2
T_{CR2}
Execute PRM2 on PRR2

Fig. 3. On-chip context save and restore (CSR) and hardware task relocation (HTR) flows

Initialization reconfigures PRR1 with PRM1 and enables FPGA protection to prevent re-initialization of the static region's and PRRs' FFs and BRAMs (Section 3.2).

When PRM2 is ready to resume execution, PRM2 can either be resumed in PRR1 or relocated to PRR2. Since CSR is faster than HTR, PRM2 will first attempt to resume execution in PRR1. For example, if PRR1 is free or PRR1 is executing a lower priority task and can be preempted by PRM2 (i.e., PRM1 is lower priority than PRM2), CSR will resume PRM2 in PRR1 by: 1) CS of PRM1; 2) merging PRM2's saved context (CS bitstream) with PRM2's initial partial bitstream to create the merged bitstream for PRR1; and 3) CR of PRM2 on PRR1. If PRR1 is not free or is executing a higher priority task (i.e., PRM1 is higher priority than PRM2), and PRR2 is available or executing a lower priority task (i.e., PRM3 is lower priority than PRM2), HTR will relocate PRM2 to PRR2 by: 1) CS of PRM3; 2) relocate PRM2's saved context to PRR2; and 3) CR of PRM2 on PRR2. Since CSR is not a contribution of this paper, the following subsections detail HTR only.

4.2 Context Save (CS)

Before reading a PRM's FFs' values, the PRR's clock is stopped to avoid potential setup/hold violations. Next, the capture process is initiated (T_{pre_CS}) and an HTR software loop captures/reads the PRM's FFs' values on a frame-by-frame basis (T_{CS_ICAP}), releases the ICAP (T_{post_CS}), and saves these values (i.e., the PRM's context) in the CS bitstream ($T_{CS\_bitstream}$). The CS bitstream size in 32-bit words is $1+N+N*41$, where N is the number of frames read and that contain the PRM's FFs' values and relative position inside the frame. The first word in the CS bitstream specifies N's value, the following N words specify the N different frame address values that contain the FFs' values, and the final $N*41$ words are the contents of the N frames. Thus, the total execution time required for CS is: $T_{CS} = T_{pre\_CS} + T_{CS\_ICAP} + T_{post\_CS} + T_{CS\_bitstream}$.

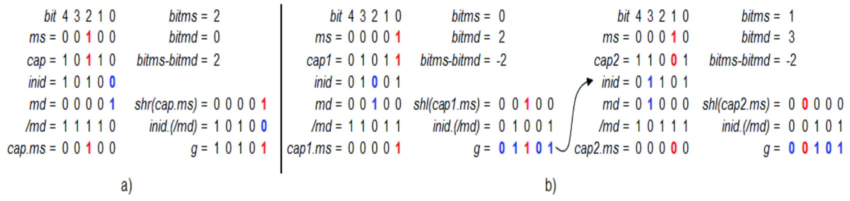


Fig. 4. Bitstream manipulations for context relocation (HTR)

4.3 Saved Context Relocation—HTR

HTR's bitstream manipulations are similar to CSR's merge except that HTR must update the PRM's FFs' values in the scheduled PRR with the PRM's FFs' values from the CS bitstream. Fig. 4 depicts the HTR bitstream manipulations, which merge the CS and initial partial bitstreams at the 32-bit word level based on whether a single or multiple FF values need to be updated. Fig. 4 a) and b) show the update of a single or multiple FF values for HTR, respectively. All examples have been reduced to five bits for clarity. A single FF update for CSR's merge can be expressed as $f = cap * msk$

+ *ini*(/msk),* where *cap* is the captured value, *ini* is the FF's value in the initial partial bitstream, and *msk* denotes if the bit is part of the saved context where $msk = 1$ updates *ini* with *cap* and $msk = 0$ retains *ini*'s value. However, *f* cannot be used for HTR because HTR requires two *msk*'s: one for the saved context and the other for the initial partial bitstream in the scheduled PRR.

HTR's context relocation is expressed as $g = cap*ms + inid*(/md)$, where *cap* is the captured value, *ms* denotes if the bit is part of the saved context, *inid* is the FF's value in the scheduled PRR's initial partial bitstream, and *md* is the bit to be updated in the merged bitstream. $md = 1$ updates *inid* with *cap*, provided that $ms = 1$, and $md = 0$ retains *inid*'s value. In Fig. 4 a) and b), *bitms* and *bitmd* denote the bit position of *ms* and *md* and the expression (*bitms - bitmd*) denotes the bit-distance between these bits' positions in a 32-bit word. If ($bitms - bitmd \geq 0$), $cap*ms$ is right-shifted (*bitms - bitmd*) bit positions using *shr(cap*ms)*, else *cap*ms* is left-shifted (*bitmd - bitms*) bit positions using *shl(cap*ms)*. Updating multiple FFs in a word boundary in the merged bitstream (Fig. 4 b)) is done sequentially, and each update does not have the same *cap* and *ms* words as shown. An HTR software loop executes this merge and relocation process and saves the merged bitstream to a file with a total execution time denoted by $T_{relocate}$.

4.4 Context Restore (CR)

Before CR, the scheduled PRR must be unprotected ($T_{unprotect\_PRR}$) to allow the PRR's FFs to be initialized with the new values in the merged bitstream, but the rest of the FPGA must remain protected. Next, the scheduled PRR is reconfigured (T_{update_PRR}) by interleaving the initial register writes (Fig. 2), the merged bitstream (Section 4.3), and the final register writes (Fig. 2). After the scheduled PRR is reconfigured with the PRM's relocated context ($T_{startup}$), the scheduled PRR is protected ($T_{protect\_PRR}$) to prevent future startup sequence phases for another PRR from re-initializing the scheduled PRR's FFs' values. Thus, the total execution time required for CR is: $T_{CR} = T_{unprotect\_PRR} + T_{update\_PRR} + T_{startup} + T_{protect\_PRR}$.

5 Experimental Results

5.1 Experimental Setup

We used the Xilinx XUPV5-LX110T board and the Xilinx ISE 12.4, XPS 12.4, and PlanAhead 12.4 tools. We partitioned the fabric into two heterogeneous PRRs and the static region executed a 100 MHz MicroBlaze soft-core processor running a Linux-like OS 2.6.37 based on BusyBox. We generated the executable binaries for the MicroBlaze using the GNU tools. A XPS HWICAP interfaced the MicroBlaze and the ICAP, the SDRAM provided external storage for the bitstreams, binaries, and the HTR files. The XPS timer was used to measure the T_x execution times and we averaged the execution times over five executions. Two XPS GPIOs provided parallel interfaces between the MicroBlaze and the two PRRs (one XPS GPIO per PRR).

We note that the MicroBlaze's configuration (e.g., instruction and data cache parameters), the XPS HWICAP's configuration, and the memory controller used to access the SDRAM files introduce overheads that affect the results, however, these components' configurations do not impact HTR's functionality, and in our analysis we note the impacts of different component configurations and hardware overheads on the results' trends.

We verified HTR's correct operation using two interfaces per PRR: one connected to the MicroBlaze and one in the PRM for transferring the PRM's FFs' values to the MicroBlaze. For testing purposes, PRM1, PRM2, and PRM3 implemented a 32-bit up counter, down counter, and pipelined adder/accumulator, respectively. We tested HTR using the flow in Fig. 3, verifying that the first value of each register in PRM2 after CR on a different PRR (i.e., the task was relocated) corresponded to the last value of each register in PRM2 prior to CS.

In order to generate thorough results for various PRR sizes in a timely manner (manual creation and testing for our experiments would have required an exorbitant amount of time), we used the following process, which did not affect the validity of our results and analyses. We created a project with two small heterogeneous PRRs containing CLBs and selected two empty areas (areas with no CLBs and routing resources in use) on the fabric. In these empty areas, we created *pseudo-PRRs*, *pseudo* initial partial bitstreams, and *pseudo* logic location files (*.ll) for *pseudo-PRM*s.

Our experiments evaluate small-to-large and large-to-small PRR HTR. We denoted the pseudo-PRR with the PRM's context as the *source* PRR and the pseudo-PRR with the relocated context as the *destination* PRR. The pseudo-PRRs' sizes contained one row and multiple columns ranging from one to twelve, which is the largest number of contiguous CLB columns on the Virtex-5 LX110T. Since the number of experimental combinations given our pseudo-PRR sizes is 144, we subset the results to show the 12 combinations where the small pseudo-PRR had half the number of columns as the large pseudo-PRR, which is sufficient to show the execution times' trends. In the large pseudo-PRRs, we evenly distributed the PRM's FFs across the CLB columns, which simulated the effects of the Xilinx tool's FF distribution done during placement and provided realistic execution times.

5.2 Execution Times

Table 1 through Table 3 show the execution times in milliseconds for the significant HTR steps. The tables contain two ranges of number of PRM FFs: a fine-grained range spanning 20 to 160 FFs in a single CLB column in 20 FF increments, and a coarse-grained range increasing the number of CLB columns, resulting in 160 FF increments. Fig. 5 plots the coarse-grained range tables' results, where each point is identified by a box with the number of rows, columns, and PRR/PRM frames depending on the graph's reported execution time.

Table 1 and Fig. 5 a) summarize $T_{reconfig\_PRR}$, which depends on the number of PRR frames (36 per CLB column). In the fine-grained range, $T_{reconfig\_PRR}$ is constant (there is only one CLB column). In the coarse-grained range, $T_{reconfig\_PRR}$ shows a linear behavior up to 960 PRM FFs. We discuss the trend above 960 FFs later in this section.

The execution time for $T_{protect\_FPGA}$ is constant and depends on the number of rows and columns in the device, which is 67.72 ms for the test device.

Table 1. Execution times (ms) for $T_{reconfig\_PRR}$

rows	1														
columns	1								2	3	4	5	6	8	12
PRR frames	36								72	108	144	180	216	288	432
PRM flip-flops	20	40	60	80	100	120	140	160	320	480	640	800	960	1280	1920
$T_{reconfig\_PRR}$	0.974	0.978	0.983	0.980	0.978	0.983	0.987	0.979	1.518	2.047	2.671	3.150	3.706	4.841	7.412

Table 2. Execution times (ms) for CS (T_{CS}), context relocation ($T_{relocate}$), and CR (T_{CR}) for small-to-large HTR

src PRR	rows	1												
	columns	1								2	3	4	5	6
	PRM frames	1							2	4	6	8	10	12
dst PRR	rows	1												
	columns	2								4	6	8	10	12
	PRM frames	2				4				8	12	16	20	24
PRM flip-flops		20	40	60	80	100	120	140	160	320	480	640	800	960
T_{CS}		4.79	4.79	4.79	4.79	4.79	4.79	4.79	5.17	5.85	6.50	7.20	7.83	8.48
$T_{relocate}$		11.53	14.96	18.46	21.38	24.99	28.89	31.92	36.02	76.79	144.81	213.34	297.81	393.81
T_{CR}		6.25	6.25	6.23	6.24	6.23	6.23	6.25	6.24	8.01	9.60	11.18	12.83	14.74

Table 3. Execution times (ms) for CS (T_{CS}), context relocation ($T_{relocate}$), and CR (T_{CR}) for large-to-small HTR

src PRR	rows	1												
	columns	2								4	6	8	10	12
	PRM frames	2				4				8	12	16	20	24
dst PRR	rows	1												
	columns	1								2	3	4	5	6
	PRM frames	1							2	4	6	8	10	12
PRM flip-flops		20	40	60	80	100	120	140	160	320	480	640	800	960
T_{CS}		5.21	5.21	5.22	5.22	5.94	5.95	5.94	5.94	7.24	8.60	10.39	11.64	13.05
$T_{relocate}$		10.90	14.43	17.86	20.62	24.38	27.73	31.42	35.79	71.77	120.18	176.31	239.15	309.52
T_{CR}		5.47	5.47	5.47	5.48	5.49	5.48	5.48	5.46	6.31	7.11	8.07	8.79	9.68

Table 2 and Table 3 summarize the execution times for T_{CS}, $T_{relocate}$, and T_{CR} for small-to-large and large-to-small HTR, respectively, and Fig. 5 b), c), and d) plot these execution times. For brevity, we omit the detailed breakdown of T_{CS} and T_{CR}, which depends on the number of PRM frames that contain used FFs in the source pseudo-PRR and the number of PRR frames in the destination PRR, respectively.

Capturing and saving the context in a small PRR shows a nearly linear increase in T_{CS}. T_{pre_CS} and T_{post_CS} for both small-to-large and large-to-small HTR are 0.54 and 1.39 ms, respectively. For small-to-large HTR, T_{CS_ICAP} ranges from 0.85 to 3.53 ms and $T_{CS\_bitstream}$ ranges from 2.01 to 3.02 ms, and for large-to-small HTR, these values range from 1.15 to 6.91 ms and from 2.13 to 4.21 ms, respectively. $T_{relocate}$ depends on the number of PRM FFs used in the PRR. The CS and merged bistreams are randomly accessed, resulting in high data cache miss rates and overheads for accessing SDRAM, which explains $T_{relocate}$'s non-linear behavior above 160 PRM FFs. Finally, T_{CR} depends on the interleaved creation of the new initial partial bitstream (Fig. 2 and Section 4.4) and sequential reconfiguration, thus T_{CR} is larger than $T_{reconfig\_PRR}$ for the same number of PRM FFs and PRR frames. $T_{startup}$ is fixed and is 0.70 ms. For small-to-large HTR, $T_{unprotect\_PRR}$, T_{update_PRR}, and $T_{protect\_PRR}$ ranges from 2.01 to 3.36 ms, 2.07 to 7.87 ms, and 1.47 to 2.81 ms, respectively, and for large-to-small HTR, these values range from 1.87 to 2.63 ms, 1.54 to 4.34 ms, and 1.36 to 2.01 ms, respectively. These results also reveal that large-to-small HTR is faster than small-to-large HTR (i.e., $T_{relocate}$ and T_{CR} are faster). Even though T_{CS} is slower for large-to-small HTR as compared to small-to-large HTR, $T_{relocate}$ is slower than T_{CS} and T_{CR}.

The resources required by the static region, including the MicroBlaze, XPS HWICAP and GPIOs, and SDRAM controller are 12,898, 44, and 4 FFs, BRAMs, and DSPs, respectively, which represent 19%, 30%, and 6%, respectively, of the test device. We note that this area overhead is reduced for devices with a dedicated on-chip hardcore processor.

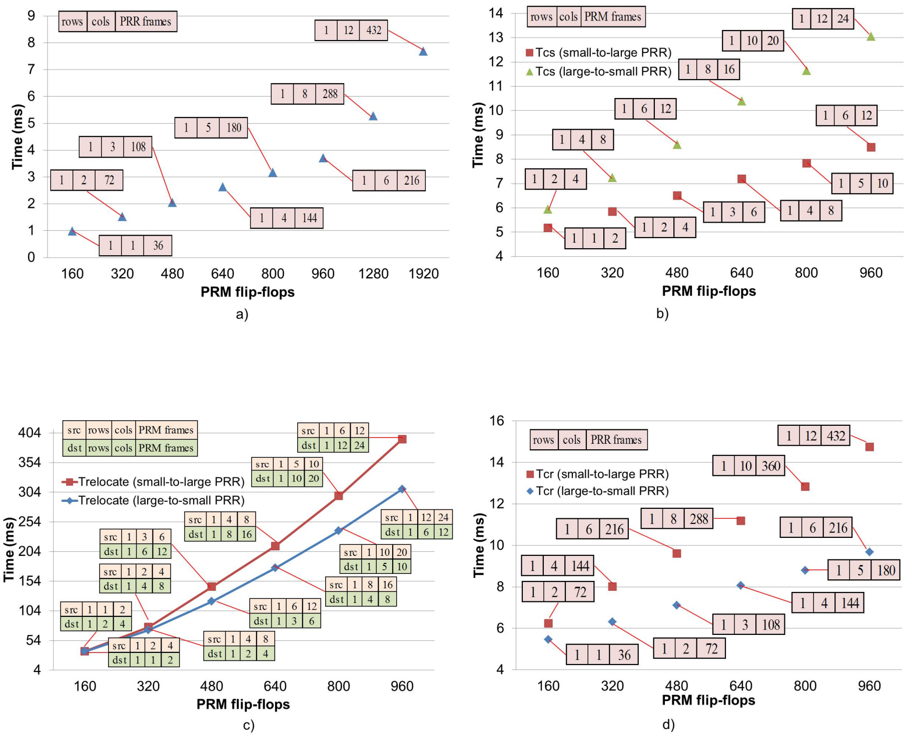


Fig. 5. Execution times (ms) for a) $T_{reconfig\_PRR}$ b) CS (T_{CS}), c) context relocation ($T_{relocate}$), and d) CR (T_{CR}) with respect to the number of PRM FFs. The adjacent rectangles indicate the number of rows, columns, and PRR/PRM frames, respectively.

Increasing the PRR's number of rows and reducing the number of columns while maintaining the same number of PRM FFs would reveal similar results as shown in the tables and figures. However, for PRRs using more than 960 PRM FFs, high data cache miss rates, SDRAM overheads when accessing the bitstreams, and the XPS HWICAP's configuration introduce a non-linear increase in the growth rate of these execution times. All HTR times may be improved by adding a custom DMA and enlarging the internal storage to the XPS HWICAP, saving the CS and initial partial bitstreams in BRAMs, or increasing/modifying the data cache size/configuration. However, BRAMs are limited and these options incur hardware overhead that may affect the system's performance, and some of these modifications would not be portable to other systems. Therefore, at design time, a system designer can consider these factors and make appropriate tradeoffs between PRR granularity, hardware overhead, and HTR execution times when partitioning the application into tasks based on the application's requirements.

6 Conclusions and Future Work

In this paper, we introduced the first, to the best of our knowledge, on-chip hardware task relocation (HTR) software for two-dimensional relocation between heterogeneous PRRs, which has no off-chip communication overhead, imposes no design/system constraints, is application/system independent, and does not require changes to the design tool flow. Our HTR maximizes shared resource utilization, performance, and throughput via task preemption and resumption between heterogeneous PRRs, which preserves the task's execution state and eliminates seconds/minutes/hours/days of re-execution time. Experimental results analyze HTR execution time, which enables system designers to guide application task granularity and partitioning decisions based on application requirements. Our future work will extend HTR's functionalities to include DSP, BRAM, and IOB resources.

Acknowledgements. This work was supported by FINCyT under contract N° 121-2009-FINCyT-BDE and in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022. The authors gratefully acknowledge the support of Universidad Nacional de Ingeniería – Lima, Perú, the Presidencia del Consejo de Ministros, through the Programa de Ciencia y Tecnología – FINCyT, and the tools provided by Xilinx.

References

1. Becker, T., Koester, M., Luk, W.: Automated placement of reconfigurable regions for relocatable modules. In: Proc. of the 2010 IEEE Intl. Symp. on Circuits and Systems (ISCAS 2010), Paris, France, pp. 3341–3344 (2010)
2. Becker, T., Luk, W., Cheung, P.Y.K.: Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. In: 15th Annual IEEE Symp. on Field-Programmable Custom Machines (FCCM 2007), Napa, California, pp. 35–44 (2007)
3. Blodget, B., James-Roxby, P., Keller, E., McMillan, S., Sundararajan, P.: A Self-Reconfiguring Platform. In: Cheung, P.Y.K., Constantinides, G.A. (eds.) FPL 2003. LNCS, vol. 2778, pp. 565–574. Springer, Heidelberg (2003)
4. Carver, J., Pittman, N., Forin, A.: Relocation and Automatic Floor-planning of FPGA Partial Configuration Bitstreams. Microsoft Research, WA. Technical Report MSR-TR-2008-111, Redmond, Washington (2008)
5. Corbetta, S., Morandi, M., Novati, M., Santambrogio, M.D., Sciuto, D., Spoletini, P.: Internal and External Bitstream Relocation for Partial Dynamic Reconfiguration. IEEE Trans. on Very Large Scale Integration (VLSI) Systems 17(11), 1650–1654 (2009)

6. Flynn, A., Gordon-Ross, A., George, A.D.: Bitstream Relocation with Local Clock Domains for Partially Reconfigurable FPGAs. In: Design, Automation & Test in Europe Conference & Exhibition 2009 (DATE 2009), Nice, France, pp. 300–303 (2009)
7. Horta, E.L., Lockwood, J.W.: PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical Report WUCS-01-13, Washington University, St. Louis, Missouri (2001)
8. Jovanovic, S., Tanougast, C., Weber, S.: A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems. In: 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), Edinburgh, United Kingdom, pp. 358–364 (2007)
9. Jozwik, K., Tomiyama, H., Honda, S., Takada, H.: A Novel Mechanism for Effective Hardware Task Preemption in Dynamically Reconfigurable Systems. In: Intl. Conf. on Field Programmable Logic and Applications (FPL 2010), Milano, Italy, pp. 352–255 (2010)
10. Kalte, H., Lee, G., Porrmann, M., Rückert, U.: REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems. In: Proc. of the 19th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS 2005), Denver, Colorado (2005)
11. Kalte, H., Porrmann, M.: Context Saving and Restoring for Multitasking in Reconfigurable Systems. In: Intl. Conf. on Field Programmable Logic and Applications (FPL 2005), Tampere, Finland, pp. 223–228 (2005)
12. Kalte, H., Porrmann, M.: REPLICA2Pro: Task Relocation by Bitstream Manipulation in Virtex-II/Pro FPGAs. In: Proc. of the 3rd Conf. on Computing Frontiers (CF 2006), pp. 403–412. ACM, New York (2006)
13. Koch, D., Haubelt, C., Teich, J.: Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation. In: Proc. of the ACM/SIGDA 15th Intl. Symp. on Field Programmable Gate Arrays (FPGA 2007), pp. 188–196. ACM, New York (2007)
14. Koester, M., Porrmann, M., Kalte, H.: Relocation and Defragmentation for Heterogeneous Reconfigurable Systems. In: Proc. of Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA 2006), pp. 70–76. Las Vegas, Nevada (2006)
15. Landaker, W.J., Wirthlin, M.J., Hutchings, B.L.: Multitasking Hardware on the SLAAC1-V Reconfigurable Computing System. In: Glesner, M., Zipf, P., Renovell, M. (eds.) FPL 2002. LNCS, vol. 2438, pp. 806–815. Springer, Heidelberg (2002)
16. Santambrogio, M.D., Cancare, F., Cattaneo, R., Bhandari, S., Sciuto, D.: An Enhanced Relocation Manager to Speedup Core Allocation in FPGA-based Reconfigurable Systems. In: 2012 IEEE 26th International Symposium on Parallel & Distributed Processing, Workshop and PhD Forum (IPDPSW 2012), Shangai, China, pp. 336–343 (2012)
17. Simmler, H., Levinson, L., Männer, R.: Multitasking on FPGA Coprocessors. In: Grünbacher, H., Hartenstein, R.W. (eds.) FPL 2000. LNCS, vol. 1896, pp. 121–130. Springer, Heidelberg (2000)
18. Sudarsanam, A., Kallam, R., Dasu, A.: PRR-PRR Dynamic Relocation. IEEE Computer Architecture Letters 8(2), 44–47 (2009)
19. Xilinx, Inc.: Virtex-5 FPGA Configuration User Guide v3.10 (UG191) (November 18, 2011)

Performance Analysis and Optimization of High Density Tree-Based 3D Multilevel FPGA

Vinod Pangracious[1], Zied Marrakchi[2], Emna Amouri[1], and Habib Mehrez[1]

[1] Laboratory d'Informatique de Paris 6
University of Pierre et Marie Curie, Paris France
[2] Flexras Technologies, Paris France

Abstract. A Tree-based 3D Multilevel FPGA architecture that unifies two unidirectional programmable interconnection network is presented in this paper. In a Tree-based FPGA architecture, the interconnects are arranged in a multilevel network with the switch blocks placed at different tree levels using Butterfly-Fat-Tree network topology. Two dimensional layout development of a Tree-based multilevel interconnect is a major challenge for Tree-based FPGA. A 3D interconnect network technology leverage on Through Silicon Via (TSVs) to re-distribute the Tree interconnects, based on network delay and thermal considerations into multiple silicon layers is discussed. The impact of of Through Silicon Vias and performance improvement of 3D Tree-based FPGA are analyzed. We present an optimized physical design technology leverage on TSV, Thermal-TSV (TTSV), and thermal analysis. Compared to 3D Mesh-based FPGA, the 3D Tree-based FPGA design reduces the number of TSVs by 29% and leads to a performance improvement of 53% based on our place and route experiments.

1 Introduction

Three Dimensional integration is a promising technology to manufacture high density and high performance Field Programmable Gate Array (FPGA). 3D integration involves stacking of multiple silicon wafers interconnected with Through Silicon Vias (TSVs) [2]. Vertical stacking of multiple chips reduces interconnect delays and increases overall integration density. Advances in 3D integration and vertical interconnect (TSVs) technologies are undoubtedly gaining momentum and have become the critical interest of the semiconductor community today. FPGA is a flexible and reusable architecture with a symmetrical array of logic blocks interconnected by routing resources. To support the growing demands, FPGAs must be built with higher logic density and interconnection networks. In such huge FPGA systems, 3D integration technology and the use of through-silicon vias (TSVs) for inter-layer communication is emerging as an effective solution to reduce the impact of increasing the interconnect delays.

For the past several years, industry and research institutions conducted major studies and research on 3D Mesh-based FPGA design and integration. A survey of existing design methods and tools for 3D integration is presented in [2] and the details of the existing 3D manufacturing technologies are presented in [3]. A 3D place and route tool (TPR) is presented in [4] to investigate the wire length and delay associated to 3D Mesh-based FPGA. In order to support the implementation of an application on such a device, VPR tools [5] are used. TPR [4] is flexible on deciding the number of vertical channels compared to horizontal channels, however it assumes all switch blocks to be 3D. This may lead to large number of unused TSV resources, which increase manufacturing cost. Furthermore, TPR also assumes the number of TSVs are electrical equivalent of the horizontal channel width. In 3D integration technology, the TSVs are much thicker than horizontal wires [9], which makes this assumption impractical.

A design framework for 3D Mesh-based FPGA architecture exploration methodology was presented in [6]. It includes an additional feature to explore the vertical interconnect distribution, however this leads to usage of 2D and 3D switch blocks intermittently, which may lead to number of design and manufacturing issues. A dynamically reconfigurable 3D Mesh-based FPGA was presented in [7], which consisted of three physical layers: logic block and local interconnect layer, routing layer, and memory layer. Recently [8] analyzed the performance benefits of a monolithically stacked 3D Mesh-based FPGA. However they used very fine TSVs 3D integration, which allowed them to stack the configuration memory on top of the of the FPGA layers.

Fundamental understanding of the electrical, mechanical, and thermal properties of vertical interconnects (TSVs) is essential in successful physical design of TSV-based 3D ICs [9]. The major challenges facing 3D integration technology today are high inter-layer temperature and limited number of TSVs. We propose architecture level solutions to optimize the number of TSV and inter-layer temperature. A detailed thermal analysis of heterogeneous Mesh-based FPGA discussed in [12], which considers functional units likes LBs, BRAM, DSP units etc. Nevertheless the programmable interconnect network of FPGA consumes a lot of power as well. In contrast, the methodology we propose, considers the power consumption of logic blocks and interconnect sections separately to investigate the temperature variations in FPGA. The rest of paper is organized as follows. Section 2 presents the Tree-based FPGA and the 3D exploration methodology elaborated in section 3. Section 4 discusses the performance analysis experiments and section 5 illustrates TSV count optimization methodology and results. Section 6 explains the thermal analysis of 3D Tree-based FPGA and section 7 conclude the paper.

2 Tree-Based Multilevel FPGA Interconnect Organization

Mesh is the most studied and used industrial topology. Considerable amount of research work [1] and industrial applications have been implemented in the case of mesh architecture. Mesh is a regular island style structure with an array of logic blocks with input pins on each side. A new re-programmable Tree-based Multilevel

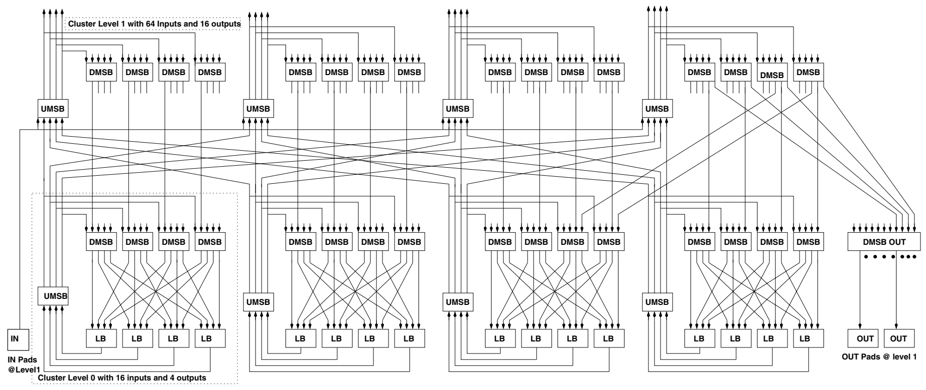

Cluster Level 1 with 64 Inputs and 16 outputs
DMSB
UMSB
DMSB OUT
LB
IN
IN Pads @Level1
Cluster Level 0 with 16 inputs and 4 outputs
OUT
OUT Pads @ level 1

Fig. 1. Tree-based Multilevel FPGA architecture with upward and downward Mini-Switch network (Rent Parameter p=1)

FPGA architecture was proposed in [10]. The main motivation for the Tree-based FPGA architecture is to achieve the best performance and density by balancing interconnect and logic block utilization, where logic blocks and routing resources are sparsely partitioned into a multilevel clustered structure [11]. In a Tree-based FPGA architecture, the LBs (Logic Blocks) are grouped into clusters located at different levels of the Tree. Each cluster contains a switch block to connect local LBs. A switch block is divided into Mini Switch Blocks (MSBs). The Tree-based FPGA architecture unifies two unidirectional upward and downward interconnection networks using a *Butterfly-Fat-Tree* topology to connect Downward MSBs (DMSBs) and Upward MSBs (UMSBs) to LBs inputs and outputs.

Figure 1 illustrates *2 level arity 4* Tree-based Multilevel FPGA architecture. The number of DMSBs of a cluster located at level ℓ is equal to the number of inputs of a cluster located at level $\ell - 1$. The upward UMSB network connects LBs outputs to the DMSBs at each level. As illustrated in Figure 1, the UMSBs are used to allow LBs outputs to reach a large number of DMSBs and to reduce fanout on feedback lines. The number of UMSBs of a cluster located at level ℓ is equal to the number of outputs of a cluster located at level $\ell - 1$. UMSBs are organized in a way allowing LBs belonging to the same "owner cluster" to reach exactly the same set of DMSBs at each level. Thus positions, inside the same cluster, are equivalent, and LBs can negotiate with their siblings about the use of a larger number of DMSBs depending on their fanout. The input and output pads are grouped into specific clusters and are connected to UMSBs and DMSBs, respectively as presented in Figure 1. Thus, all input and output pads can reach any LBs of the architecture.

3 Exploration Methodology for 3D Tree-Based FPGA

The proposed methodology for design and exploration of 3D Tree-based FPGA architecture is illustrated in figure 2. The HDL generator is designed to

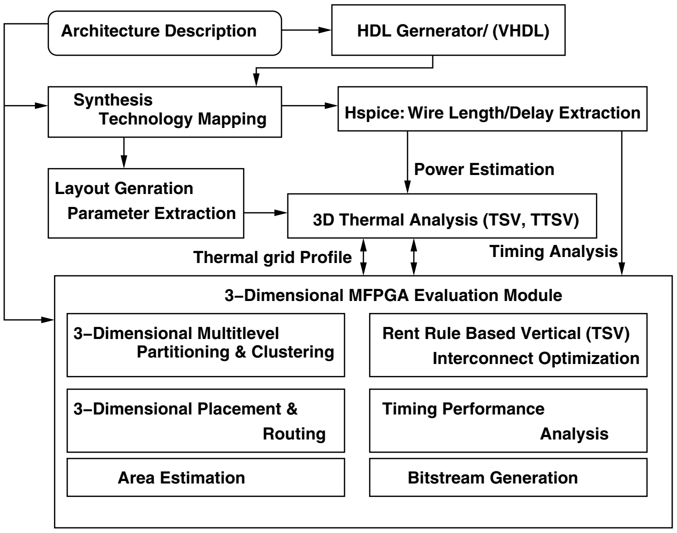

Architecture Description
HDL Gernerator/ (VHDL)
Synthesis Technology Mapping
Hspice: Wire Length/Delay Extraction
Layout Genration Parameter Extraction
Power Estimation
3D Thermal Analysis (TSV, TTSV)
Thermal grid Profile
Timing Analysis
3–Dimensional MFPGA Evaluation Module
3–Dimensional Multitlevel Partitioning & Clustering
Rent Rule Based Vertical (TSV) Interconnect Optimization
3–Dimensional Placement & Routing
Timing Performance Analysis
Area Estimation
Bitstream Generation

Fig. 2. 3D Tree-based Multilevel FPGA performance evaluation flow

generate VHDL code based on a hierarchical approach that partitions the design into smaller sections, implement them separately and assemble them together at the final design phase. The VHDL code is generated based on architecture description file, which is used directly for design evaluation and analysis. The thermal model [15] is used to extract the thermal profile of the multi-layer chip based on layout geometrical features and power consumption of the functional unites. The 3D Tree-based FPGA evaluation module includes a *top-down* recursive partitioning tool. A negotiation-based iterative *"Pathfinder"* [11], approach is used to implement the routing algorithm. The physical design experiments are performed on the layout generated using ST Micro‘s 130nm technology node. Mentor's Spice accurate circuit simulator *Eldo* is used to estimate the wire delay and power consumption of switches and interconnection networks at different tree levels. Physical design of multilevel tree interconnect is a major challenge for Tree-based FPGA. In order to maintain the hierarchy of Tree-based FPGA, a special layout methodology is used. We propose two ways to organize the Tree-based Multilevel FPGA layouts.

3.1 2-Dimensional Tile-Based Multilevel FPGA Design

The physical design experiments revealed the wire length increases exponentially as the Tree grows to higher levels, is considered as a major disadvantage of Tree-based architecture compared to Mesh, where the largest wiring distance is fixed. The layout experimentation was performed based on the layout generated using ST microelectronics 130nm technology node. The 2D Tile-based layout illustrated in Figure 3 is developed to spread the congestion and wire density over the Tree-based Multilevel FPGA surface. The switch box belonging to various

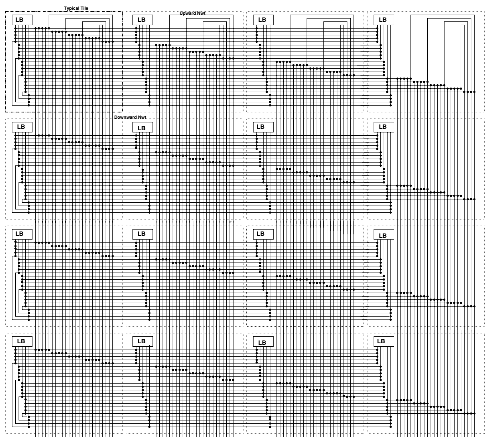


Fig. 3. 2D optimized 4×4 Tree-based Multilevel FPGA floorplan

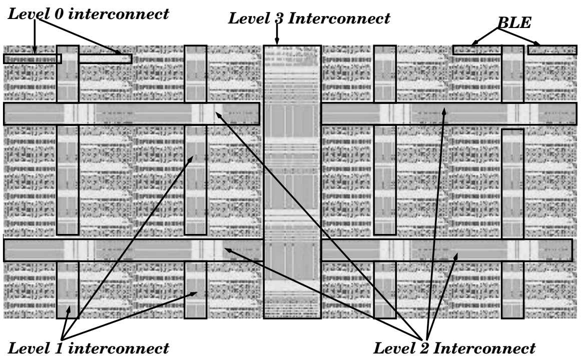


Fig. 4. Tree-based VLSI Layout design of Multilevel FPGA Tree section from *level 0 to 3*

levels may be coalesced in the same tile. However this layout is not comparable to industrial Mesh layout in terms of speed and performance due to larger wire delay at higher levels of the Tree network [11].

3.2 3-Dimensional Tree-Based Multilevel FPGA Design

To mitigate the wire length issue in 2D Tile-based layout, we designed a new Tree-based 2D layout with 3D adaptability illustrated in Figure 4. The interconnect organization in Tree-based layout is arranged in way to bring together every cluster and its corresponding interconnect in order to form level wise sections to enable the study of 2 layer 3D Tree-based FPGA. Figure 4 illustrates the custom designed VLSI layout of Tree *levels 0 to 3* [16]. This layout design offers the possibility to re-distribute the Tree interconnect at certain level called the *break point* level based on wire delay estimation from the timing characterization and

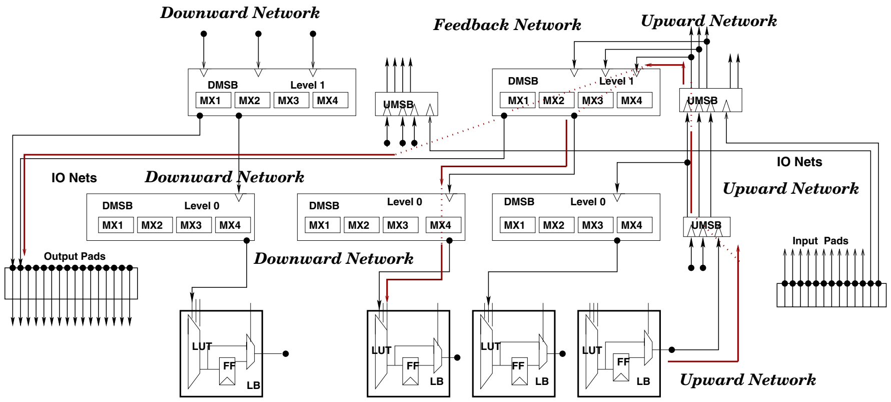


Fig. 5. Sub-path timing characterization setup shows two levels and IOs of Tree-based FPGA

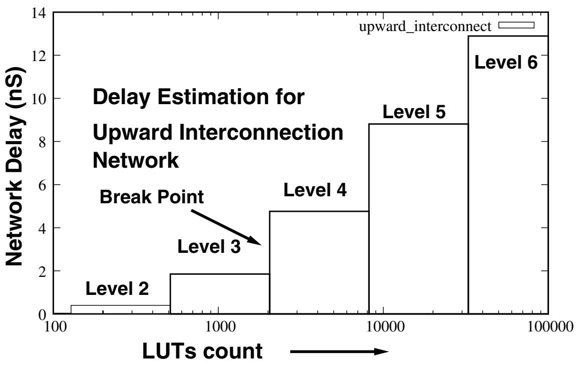


Fig. 6. Sub-paths timing characterization: Delay estimation of Upward Tree Network

thermal analysis data. However this is not possible with 2D Tile-based layout since the switches from different levels are coalesced in the same tile.

The subpath timing characterization is performed for both 2D Tile-based and Tree-based layout using the layout generated in ST Micro's 130nm Technology node. Maximum wire length at different levels are evaluated from the layout and used Mentor's spice accurate circuit simulator *Eldo* to investigate delay and power consumption. An accurate ST 130nm transistor level technology models are used to investigate switch, interconnect delay and power estimation at each level separately. A model used for timing characterization with *2 level* Tree architecture illustrated in Figure 5, in which, the upward, downward and feedback interconnect networks are marked. We performed delay estimation and power consumption analysis on all three interconnection networks. Figure 6 illustrate the upward interconnection network delay measured up to *7 levels* of the Tree-based multilevel FPGA architecture. Similar delays measured for other

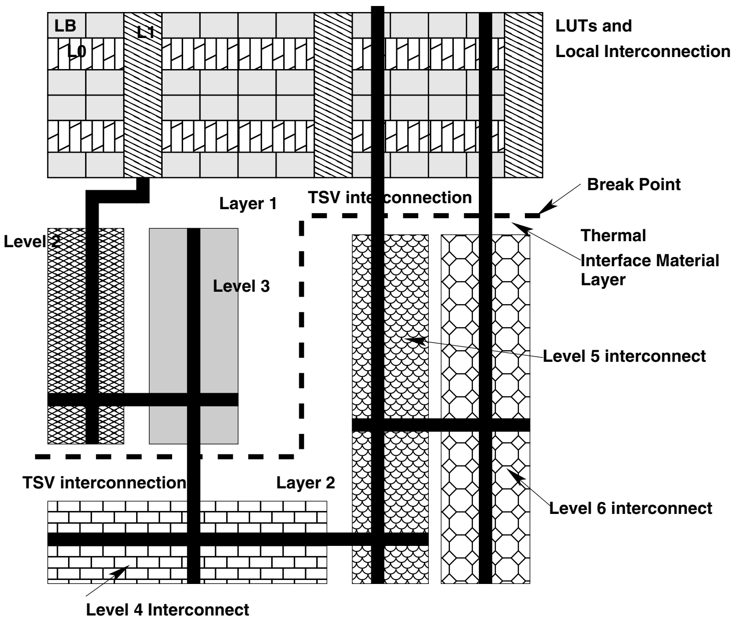


Fig. 7. 3-Dimensional 2 layer Multilevel FPGA with break point at *level 3 and 4*

interconnect networks as well. The interconnect delay investigation substantiate the exponential increase in wire delay as the tree grows to higher levels.

Based on the measured delay and thermal data, the 2D Tree-based layout design is re-distributed into 2 silicon layers at a higher interconnect Tree level called the *break point* level. The decision to choose the break point level is based on measured delay and thermal data. In this study the interconnect network is partitioned between *level 3 and 4* as the average delay is above 2ns to form a two layer 3D Tree-based Multilevel FPGA. To illustrate the design process a *7 level* Tree-based FPGA architecture is presented in Figure 7, where the break point is shown between *levels 3 and 4*. For this study the communication is realized with Through Silicon Via (TSV) and electrical characterization of TSVs was performed based on the approach from [14]. The electrical model and parasitic components for each TSV was realized using the electrical model of TSV interconnect presented in [14]. The interconnect length of levels above the break point level for 3D Tree-based FPGA timing characterization was extracted from the re-designed floorplan shown in Figure 7. The 2 layer 3D Tree-based FPGA architecture presented in Figure 7 used for experimentation and comparison. Nevertheless as Tree grows to higher levels, the multiple layer 3D Tree-based FPGA can be designed.

4 3D Tree-Based Multilevel FPGA Experimental Evaluation

To evaluate the performance of the proposed 3D architecture, we place and route the largest set of MCNC[1] benchmark circuits, and compare with the 3D Mesh-based FPGA architecture [6]. The netlist is partitioned into tree based cluster nets attributing randomly to each cluster a position inside the owner. An iterative negotiation-based *PathFinder* approach [11] is used to implement the placement and routing algorithm which is able to deal with any graph representing the interconnection routing resources. The 3D routing tool was adapted to handle the performance analysis of 2D and 3D layout with TSV interconnections, based on the 2 layer Tree-based FPGA.

Table 1. 3-Dimensional Multilevel FPGA detailed performance evaluation

		Delay($\times 10^{-9} sec$)			Performance Gain(%)		Previous Work
circuits	arch	$2D_{Tile}$	$2D_{Tree}$	$3D_{FulL}$	$2D_{Tile}$	$2D_{Tree}$	3D Mesh Gain
MCNC	Configuration	Tile-based	Tree-based	$3D_{TSV}$	Vs $3D_{TSV}$	Vs $3D_{TSV}$	Vs 2D [6]
alu4	4x4x4x4x4x4x4	53	63	24	54.7	62	37
apex2	4x4x4x4x4x4x4	48	58	19	60.4	67.2	50
apex4	4x4x4x4x4x4x4	53	65	17	68	73.8	46
bigkey	4x4x4x4x4x4x4	18	22	8	55.5	63.6	39
clma	4x4x4x4x4x4x4	175	198	43	76.3	78.2	33
des	4x4x4x4x4x4x4	34	42	16	53	62	32
diffeq	4x4x4x4x4x4x4	42	51	23	45.3	55	-1.6
disp	4x4x4x4x4x4x4	23	28	7	69.6	75	29
elliptic	4x4x4x4x4x4x4	73	90	31	57.5	65.6	6
ex1010	4x4x4x4x4x4x4	188	212	38	79.7	82	12
ex5p	4x4x4x4x4x4x4	54	66	18	66.7	72.7	55
frisc	4x4x4x4x4x4x4	89	108	40	55.1	63	-10
misex3	4x4x4x4x4x4x4	40	48	15	62.5	68.8	54
pdc	4x4x4x4x4x4x4	174	198	37	78.7	81.3	47
s298	4x4x4x4x4x4x4	97	118	34	65	71.2	50
s38417	4x4x4x4x4x4x4	94	106	26	72.3	75.5	29
s38584	4x4x4x4x4x4x4	113	129	28	75.2	78.3	38
seq	4x4x4x4x4x4x4	40	49	15	62.5	69.4	28
spla	4x4x4x4x4x4x4	53	69	18	66	74	50
tseng	4x4x4x4x4x4x4	40	49	22	45	55.1	16
ava	4x4x4x4x4x4x4	215	248	125	41.9	49.6	
average	21	81.71	96.06	28.76	62.4	68.7	32

The detailed performance analysis of 2D and 3D designs presented in Table 1. An arity 4 Tree-based FPGA architecture with tree level 0 to 6 is presented in Table 1. The critical path delay comparison between 2D and 3D layout shows that the small and big designs outperform in 3D implementation of Tree-based FPGA compared to the 2D counterpart. The place and route experiment records

[1] `http://er.cs.ucla.edu/benchmarks/ibm-place`

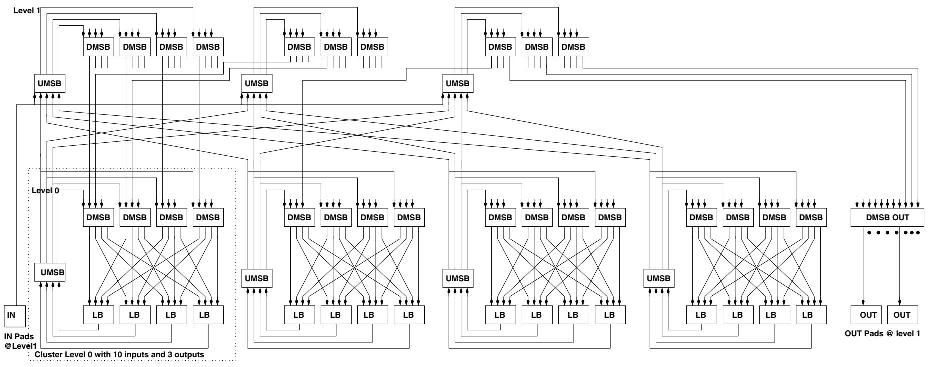


Fig. 8. Vertical interconnect (TSV) depopulation using Rent's rule(*level 1* with p=0.73)

an average speed improvement of 68.7% compared to our 2D design. The gain obtained in performance is due the optimized wire delay at higher levels of the Tree interconnect by re-arranging them in the 2 layer 3D chip with the Tree interconnection between *level 3 and 4* is realized using TSVs. Similarly the comparison with 3D Mesh-based FPGA [6] with 32% gain shows, 3D Tree-based FPGA outperform in all benchmarks and an overall performance gain of 53% recorded in the experiment.

5 Vertical Interconnect (TSV) Optimization

To make 3D Tree-based Multilevel FPGA more effective in terms of design and manufacturing, its essential to minimize the TSV count. The vertical interconnect optimization is be done using Rent's parameter "p" defined for the an architecture as follows. The Tree level is represented as ℓ and m is the cluster arity, c is the number of in/out pins of an LB and IO is the number of in/out pins of a cluster located at level ℓ.

$$IO = c.m^{\ell \cdot p} \tag{1}$$

A Rent's parameter based random level vertical interconnect minimization program developed on 3D Tree-based router is used to find the smallest number of vertical interconnects to implement MCNC netlist using a binary search methodology. The optimization program consider the same architecture level, in this case the *break point level* with different p values to estimate the minimum TSV requirement for a particular netlist. An example of two level Tree-based FPGA with p=0.73 illustrated in Figure 8, in which a 27% reduction of interconnects requirement achieved. The optimization of Tree-based interconnect network based on Rent's parameter as follows.

Table 2. 3-Dimensional Multilevel FPGA Vertical Interconnect (TSV) Optimization

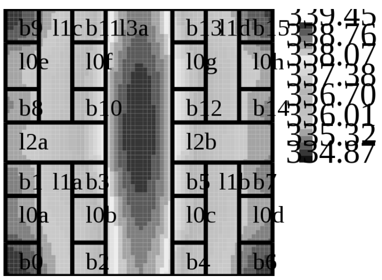
Design Names	Architecture	Optimized	$3D_{\mathrm{TSV}}Gain$	Rent="p"	Rent=1	Speed
MCNC	Configuration	Rent's "p"	(%)	speed(nS)	speed(nS)	degradation(%)
alu4	4x4x4x4x4x4x4	0.47	53	25.4	24	5.5
apex2	4x4x4x4x4x4x4	0.72	28	20.7	19	8.3
apex4	4x4x4x4x4x4x4	0.77	23	17.2	17	1.1
bigkey	4x4x4x4x4x4x4	0.61	39	8.6	8	6.9
des	4x4x4x4x4x4x4	0.77	23	17.2	16	6.9
diffeq	4x4x4x4x4x4x4	0.66	34	25.5	23	9.8
dsip	4x4x4x4x4x4x4	0.65	35	7.6	7	7.8
ex5p	4x4x4x4x4x4x4	0.76	24	19.4	18	8.7
frisc	4x4x4x4x4x4x4	0.74	26	42.7	40	3.9
misex3	4x4x4x4x4x4x4	0.64	36	16.1	15	6.8
pdc	4x4x4x4x4x4x4	0.77	23	38.5	37	3.8
s298	4x4x4x4x4x4x4	0.76	24	36.2	34	6.8
seq	4x4x4x4x4x4x4	0.76	24	16.2	15	7.4
spla	4x4x4x4x4x4x4	0.78	22	19.4	18	7.2
tseng	4x4x4x4x4x4x4	0.65	35	24.4	22	9.8
ava	4x4x4x4x4x4x4	0.79	21	128.1	125	2.4
average	16	0.704	29.6	28.95	27.4	6.4

5.1 The Tree-Based Multilevel FPGA Interconnect Network Model

In downward interconnection network, a cluster situated at level ℓ contain $N_{in}(\ell-1)$ DMSB with k outputs and $\frac{N_{in}(\ell)+kN_{out}(\ell-1)}{N_{in}(\ell-1)}$ inputs. DMSBs being full crossbar devices, total number of downward switches at level ℓ cluster is $k(N_{in}(\ell)+kN_{out}(\ell-1))$. In upward interconnection network, every cluster at level ℓ contain $N_{out}(\ell-1)$ UMSBs with k inputs and outputs. UMBSs are also full crossbar devices with $k^2 \times N_{out}(\ell-1)$ switches at a level ℓ cluster. Since we have $\frac{N}{k^\ell}$ clusters at each level ℓ, and the total number of switches in Tree network can be calculated by equation 2.

$$N_{switch}(Tree) = N \times (k^p c_{in} + 2kc_{out}) \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)} \tag{2}$$

The effectiveness of TSV optimizer was evaluated by using 16 largest MCNC benchmark circuits. During the optimization process each netlist is passed through a 3D router based TSV optimizer to find the minimum number of TSVs required to implement the function with the 2 layer 3D Tree-based multilevel FPGA. The advantage in this type of optimization is to provide a realistic count of TSVs requirement for each netlist cases whereas the 3D Mesh-based FPGA design methodology [6] assumed a random value of 30% of the actual architecture TSVs count for circuit implementation and it produced counterproductive results for few circuits like *diffeq, frisc etc* as shown in Table 1. The Tree-based FPGA experimental study revealed an average reduction of 29.6% in TSV count and corresponding speed degradation of 6.4% is presented in Table 2. With the current 3D process technologies, it is impossible to manufacture high amount of



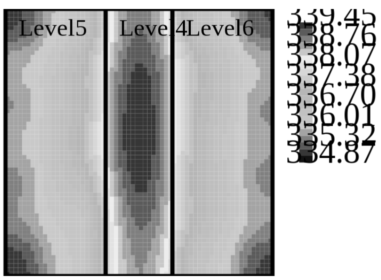


Fig. 9. Thermal profile of 3D Tree-based Multilevel FPGA, with layer 1(left) with *level 1 2 and 3* and *layer 2* (right) with *level 4, 5 and 6*

TSVs and this makes the proposed TSVs reduction methodology a more feasible technology approach. The technological approach and results confirm that 3D Tree-based FPGA is a consistent architecture to build high density and high performance FPGA, which is unlikely to be attained in Mesh-based FPGA architecture.

6 3D Tree-Based Multilevel FPGA Thermal Analysis

Thermal analysis of FPGA architecture is essential as the power dissipation and leakage expected to increase as we scale the technology below 100nm node [9,12]. The absence effective heat removal solutions may lead to performance and reliability degradation of the 3D chip and an effective thermal conduction among multiple layers of 3D chip is essential to maintain the performance of the 2 layer 3D Tree-based Multilevel FPGA. The thermal model used in this work is similar to the model presented in [15] and it considers the temperature-dependent thermal conductivity of silicon. The 3D thermal model is modified to include the impact of effective thermal conductivity of thermal interface material (TIM) through which the vertical interconnections (TSVs) pass through. The TIM layer is a thermally inactive layer and it is used to attach layer 1 and 2 on top of each other. Nevertheless the thermal conductivity of TIM increases due to the cu TSVs from layer 1 to 2. The effective thermal conductivity of TIM and the active layer 2 is calculated based on the equation 3. TSV density is computed based on number of optimized TSVs count, TSV dimensions, and pitch constraints [14] between TSVs of layer 1 and 2.

$$k_{eff} = k_{cu} \times (TSV_{Area}) + K_{th} \times (Level_{BreakPointArea} - TSV_{Area}) \quad (3)$$

Another feature included in 3D thermal model is to place additional block of thermal TSVs at a specific hotspot location to re-distribute heat from a hotspot to coldspot. The thermal profile of layer 1 and 2 is presented in Figure 9 and temperature analysis presented in Figure 10. The measured maximum temperature of 2 layer 3D Tree-based FPGA chip without thermal TSVs (TTSV) is

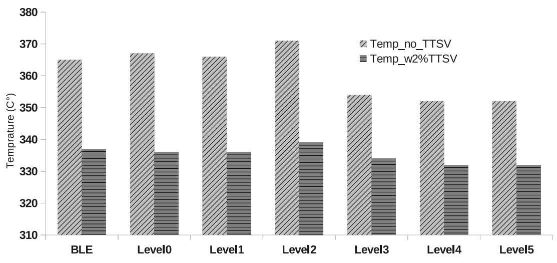


Fig. 10. Temperature extracted for different sections of the 3D Tree-based FPGA chip before and after Thermal TSV insertion

371^oC and average temperature is 361^oC. With addition of 2% TTSVs at the hotspot location resulted in balancing the temperature of the chip. The maximum temperature measured is 341^oC and average temperature is 335^oC. The 3D experimental chip had only 2 active layers and 1 TIM layer, which explains the dramatic improvement in temperature, nevertheless the improvement varies with number of layers in a 3D chip.

7 Conclusion

We have demonstrated that 3D Tree-based Multilevel FPGA provides significant advantages over 2D Mesh-based FPGA by improving the performance by 53% and a reduction of 29% in overall TSV count of 3D Tree-based FPGA. Also addressed the 2D physical design issues of Tree-based Multilevel interconnect architecture and demonstrated our alternative 3D physical design and vertical interconnect optimization solutions. However the 3D integration increases the inter-layer temperature. Our 3D experimental setup for thermal analysis indicate the the peak temperature of 2 layer 3D chip increased to 371^oC. However the heat transfer solution by placing thermal TSVs blocks at specified locations helped the 3D Tree-based Multilevel FPGA to balance the temperature uniformly across multiple layers of the 3D chip.

References

1. Betz, V., Marquardt, A., Rose, J.: A New Packing Placement and Routing Tool for FPGA Research. In: Intl. Workshop on FPGA, pp. 213–222 (1997)
2. De Micheil, G., Pavlidis, V., Atienza, D., Leblebici, Y.: Design Methods and Tools for 3D integration. In: Symposium on VLSI Tech Digest of Technical Papers, pp. 182–183 (2011)
3. Beyne, E.: 3D Interconnection and Packaging: Impending reality or still a Dream? In: Proc. of IEEE Intl. Solid-state Circuits Conf (ISSCC 2004), CA, vol. 1, pp. 138–139 (February 2004)
4. Ababei, C., Feng, Y., Goplen, B.: "Placement and routing in 3D integrated circuits". IEEE Design & Test of Computers 22(6), 520–531 (2005)

5. Betz, V., Rose, J., Marquardt, A.: Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, Dordrecht (1999)
6. Siozios, K., Bartzas, A., Soudris, D.: Architecture Level Exploration of Alternative schemes Targeting 3D FPGAs: A Software Supported Methodology. Intl. Journal of Reconfigurable Computing (2008)
7. Chiricescu, S., Leeser, M., Vai, M.M.: Design and analysis of a dynamically reconfigurable three-dimensional FPGA. IEEE Trans. Very Large Scale Integr (VLSI) Syst. 9(1), 186–196 (2001)
8. Lin, M., Gamal, A., Lu, Y., Wong, S.: Performance Benefits of Monolithically Stacked 3D-FPGA. In: Int. Symp. Field-Program. Gate Arrays, Monterey, CA (2006)
9. Lim, S.: TSV-Aware 3D Physical Design Tool Needs for Faster Mainstream Acceptance of 3D ICs. In: ACM DAC Knowledge Center (dac.com) (2010)
10. Marrakchi, Z., Mrabet, H., Amouri, E., Mehrez, H.: Efficient tree topology for FPGA interconnect network. In: ACM Great Lakes Symp. on VLSI 2008, pp. 321–326 (2008)
11. Marrakchi, Z., Mrabet, H., Farooq, U., Mehrez, H.: FPGA Interconnect Topologies Exploration. Int. J. Reconfig. Comp. (2009)
12. Gayasen, A., Narayanan, V., Kandemir, M., Rahman, A.: Designing a 3-D FPGA: Switch Box Architecture and Thermal Issues. IEEE Trans. on VLSI Syst. 16(7), 882–893 (2008)
13. Pistorius, J., Hutton, M.: Placement rent exponent calculation methods, temporal behaviour and FPGA architecture evaluation. In: Proc. of the Intl. Workshop on System Level Interconnect Prediction, Monterey, Calif, USA, pp. 31–38 (April 2003)
14. Jang, D.M., Ryu, C., Lee, K.Y., et al.: Development and evaluation of 3-D SiP with vertically interconnected Through Silicon Vias (TSV). In: Proc. of the 57th Electronic Components and Technology Conf (ECTC 2007), USA, pp. 847–852 (May-June 2007)
15. Ayala, J., Sridhar, A., Pangracious, V., Atienza, D., Leblebici, Y.: Through Silicon Via-Based Grid for Thermal Control in 3D Chips. NanoNet (2009)
16. Emna, A., Hayder, M., Zied, M., Habib, M.: Improving the Security of Dual Rail Logic in FPGA Using Controlled Placement and Routing ReConFig 2009, Mexico (2009)

Iterative Routing Algorithm of Inter-FPGA Signals for Multi-FPGA Prototyping Platform

Mariem Turki[1], Zied Marrakchi[2], Habib Mehrez[1], and Mohamed Abid[3]

[1] Laboratoire d'Informatique de Paris 6
Universite de Pierre et Marie Curie, Paris France
[2] Flexras Technologies, Paris France
[3] CES Laboratory
Sfax University, Tunisia

Abstract. Over the last few years, multi-FPGA-based prototyping becomes necessary to test System On Chip designs. However, the most important constraint of the prototyping platform is the interconnection resources limitation between FPGAs. When the number of inter-FPGA signals is greater than the number of physical connections available on the prototyping board, signals are time-multiplexed which decreases the system frequency. We propose in this paper an advanced method to route all the signals with an optimized multiplexing ratio. Signals are grouped then routed using the intra-FPGA routing algorithm: Pathfinder. This algorithm is adapted to deal with the inter-FPGA routing problem. Many scenarios are proposed to obtain the most optimized results in terms of prototyping system frequency. Using this technique, the system frequency is improved by an average of 12.8%.

1 Introduction

With the ever increasing complexity of system on chip circuits, the software and hardware developers can no longer wait for the fabrication phase to test their designs[3]. Currently, it is estimated that 60 to 80 percent of an ASIC design is spent performing verification [13]. FPGA-based prototyping is an important step in the creation of the final product and it is the key to the success of marketing in time.

Because the silicon area overhead of FPGA versus ASIC technology has been measured to be about 40x [14], FPGA programming technology requires that an ASIC logic design be partitioned across multiple FPGA devices to achieve the necessary device logic capacity. The number of FPGAs depends on the size of the prototyped system, ranging from a few [4] up to 60 FPGAs [5].

In order to map the design into a multi-FPGA board, a partitioning tool decomposes the design into pieces that will fit within the logic resources of individual FPGA devices. For some systems, partitioning must be performed so that routing restrictions in terms of available FPGA pin count and system topology are taken into account. Indeed, the number of I/Os is increasing for each new FPGA generation, but the ratio FPGA I/Os over FPGA logic capacity is decreasing. Thus, the number of signals which appear at the interface and which should be transmitted between FPGAs, is significantly higher than the number of available traces between those FPGAs.

The communication of interpartition signals between FPGAs is based on routing algorithms. In this paper, we propose a new approach to route all the inter-FPGA signals, based on signal multiplexing technique. To reach this goal, we use an iterative routing algorithm called Pathfinder [6]. This algorithm was used to route the intra-FPGA signals. We extend it for the inter-FPGA signals in order to obtain the best routing results.

The rest of the paper is organized as follows. Section 2 is dedicated to the related works which addressed this problem. In section 3 we present the iterative routing algorithm used initially to route the intra-FPGA signals. Section 4 explains the scenarios we propose to test the performance of the routing algorithm. These scenarios includes the inter-FPGA signal form and also the routing graph direction. In section 5 we describe the multiplexing IP that we use to transfer the multiplexed signals. section 6 is dedicated to the experimental results and to the evaluation of the the proposed methods. Finally, section 7 concludes the paper.

2 Related Works

To address the inter-FPGA signal routing problem, authors in [8] proposed heuristic algorithms to solve multiterminal routing signals in partial crossbar architectures. In [9], multiterminal signals are decomposed into two-terminal nets. Therefore, routing algorithm is applied to these nets.

Bab et al [1] introduced time multiplexing of I/O pins. Multiplexing means that multiple design signals are assembled and serialized through the same board connection and then de-multiplexed at the receiving FPGA. In [2], the authors proposed a new multiplexing approach based on the Integer Linear Programming. The main objective of this study is to select which signals must be multiplexed and those which must not. Using this technique, all signals are transmitted on each phase, but only those with updated values are considered. Since all the signals are transmitted in each phase, the number of slot per phase is very large, and the system frequency is decreased.

3 Inter-FPGA Signals Routing Strategy

To route inter-FPGA signals, it is necessary to find an algorithm that can assign, in an optimized manner, signals to the available resources. The technique mentioned in the section II uses constructive routing algorithm. This algorithm keeps track of the reserved and available physical connections between FPGAs. The router applies Dijkstra's shortest path algorithm [7] to determine the shortest path between the source and destination FPGAs. If the shortest path exists, the

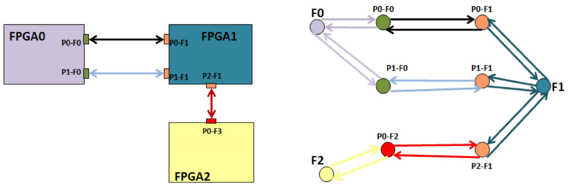


Fig. 1. Modelling routing resources as a routing graph

capacity of all used resources is decremented, then they can not be used to route the next signals. Else, router returns unsuccessfully. The main disadvantage of this method is: when a signal is already routed, it can not be rerouted to leave the routing resources currently used to another signal that has the greatest need for these resources. To avoid this problem, we route the inter-FPGA signals by an iterative routing algorithm. Among existing techniques, The Pathfinder routing algorithm seems to be best suited to our problem as it offers a compromise between performance and routability goals.

3.1 Routing Graph

Since we have chosen Pathfinder to route all inter-FPGA signals, our interest was about the modeling of the multi-FPGA board. Therefore, we chose to model all the routing resources by an oriented routing graph G(V, E). Like shown on Figure 1, the set of vertices V=$v_1, ...v_n$ in the graph represents the I/O pins of all FPGAs, but also, each FPGA is represented by a top vertex. The set of edges E=$e_1, ..., e_n$ represents all the inter-FPGA connections. An unidirectional connection is modeled by a directed edge while a bidirectional connection(for example between a vertex and a top vertex) is represented by two directed edges.

3.2 Routing Algorithm: Pathfinder

Pathfinder is used primarily for routing intra-FPGA signals. We adapt it to deal with the inter-FPGA signals. Pathfinder uses an iterative, negotiation-based approach to successfully route all the signals. During the first routing iteration, the signals are freely routed without paying attention to resource sharing. Individual signals are routed using Dijkstra's shortest path algorithm [7]. At the end of the first iteration, resources may be congested because multiple signals have used them. During subsequent iterations, the cost of using a resource is increased, based on the number of signals that share the resource, and the history of congestion on that resource. Thus, signals are forced to negotiate for routing resources. If a resource is highly congested, nets which can use lower congestion alternatives are forced to do so. On the other hand, if the alternatives are more congested than the resource, then a signal may still use that resource.

Observing the final routing results, we notice that inter-FPGA signals can be directly routed between source and destination FPGAs, or intermediate through-hops may be necessary.

4 Routing Algorithm Adaptation

Taking into account some problems to be detailed later, we adapt our routing approach to the new routing topology. In this section, we discuss the proposed solutions and the various changes we make.

4.1 Signal Direction Conflicts

The Pathfinder routing algorithm processes each signal independently. Each routing resource (node) should be shared by more than one signal. Signals that share the same resource are multiplexed together. As mentioned above, we model our architecture by a bidirectional routing graph. This causes direction conflicts since the signals sharing the same resources can have different directions.

Unidirectionnal Routing Graph. To avoid this problem, we apply the Pathfinder routing algorithm on a unidirectional graph. The idea is to assign a definite direction to all physical wires. In the routing graph, this is translated by a single edge between each pair of nodes.

Figure 2-(a) represents the routing flow on a unidirectionnal graph. The first generates the unidirectionnal graph depending on the number of inter-FPA signals between each pair. The number of physical wires that transmit direct (respectively indirect) signals between two FPGAs, is proportional to the number of direct (respectively indirect) signals between these two FPGAs. After calculating the multiplexing ratio, the capacity of all nodes is set to mux_ratio. Finally, Pathfinder routing algorithm tries to route all the signals. If a feasible solution exists, the mux_ratio parameter is decremented and the router tries to re-route the signals with the new value of mux_ratio. Otherwise, the router stops with the best solution found.

Bidirectionnal Routing Graph. The selection of the unidirectionnal wires proportionally to the number of signal between each pair of FPGA is not optimized at all. For this reason we keep the bidirectionnal graph and we combine signals into groups. Indeed, signals that have the same source and the same destinations are grouped together in "GSignals" and are considered as a single signal. Each GSignal contains a maximum of mux_ratio signals. Therefore, the capacity of all resources in the routing graph is set to 1. The bidirectional graph allows a better use for available routing wires of the multi-FPGAs prototyping board.Figure 2-(b) presents the steps to route inter-FPGA signals on a bidirectional routing graph. The first step creates the graph using two arcs of opposite direction to represent each physical wire. Next, the initial mux_ratio parameter is calculated the same way as in the unidirectional graph. This parameter determines the number of signals to be grouped together into one GSignal. The

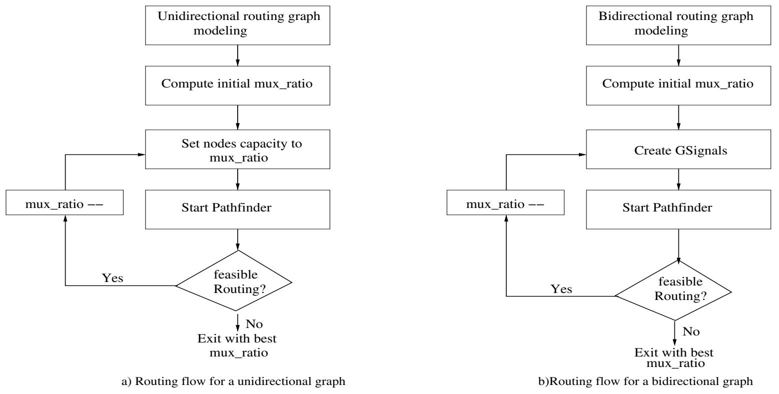

Unidirectional routing graph modeling
Compute initial mux_ratio
Set nodes capacity to mux_ratio
Start Pathfinder
mux_ratio --
feasible Routing?
Yes
No
Exit with best mux_ratio
a) Routing flow for a unidirectional graph
Bidirectional routing graph modeling
Compute initial mux_ratio
Create GSignals
Start Pathfinder
mux_ratio --
feasible Routing?
Yes
No
Exit with best mux_ratio
b)Routing flow for a bidirectional graph

Fig. 2. Routing flows

pathfinder algorithm tries to route all the GSignals. Finally, the router retains the routing solution with the best mux_ratio.

This method avoids conflict management, since the Pathfinder algorithm prevents congestion; at the end of every iteration, no node is used by more than one group of signals or GSignals, which all have the same direction.

4.2 Signal Representation

For better routing results, we notice that the choice of signal form is essential with two possibilities to consider the signal shape: a multiterminal or a two-terminal signal.

Multiterminal Signal. The Pathfinder routing algorithm can route multiterminal nets. In fact, the algorithm starts by selecting the source and the list of all destinations. After routing the first one, Pathfinder moves to the next destination and so on. Although the routing of multiterminal signals can be the optimal solution considering the number of used I/O pins, the design is considered non flexible especially when grouping those signals into GSignals. Indeed, in some cases, signals with the same source and the same destinations are not numerous so that some GSignals do not contain the max number of signals, equal to mux_ratio.

Two-Terminal Signal. In order to make the design more flexible, we decompose the multiterminal signals into branches with one source each and only one destination. The Pathfinder routing algorithm tries to find separately a routing path to each branch.

Table 1. Comparison of routing strategies effects on prototyping system performance

Benchmark	scenario1			scenario2			scenario3			scenario4		
	mux_ratio	R_hop	Freq (MHz)	mux_ratio	R_hop	Freq (MHz)	mux_ratio	R_hop	Freq (MHz)	mux_ratio	R_hop	Freq (MHz)
Circuit A	12	2	17.85	15	2	16.66	4	2	20.83	4	1	26.31
Circuit B	18	3	13.88	24	2	14.7	4	3	17.24	7	1	23.8
Circuit C	24	3	12.82	44	2	11.36	11	3	15.15	11	1	21.73
Circuit D	50	3	9.61	50	2	10.63	15	3	14.28	20	1	18.51
Circuit E	119	6	4.9	116	4	5.55	57	2	9.8	56	4	8.33
Circuit F	160	3	4.67	168	3	4.5	68	3	8.19	68	1	9.8
Circuit G	220	5	3.4	256	1	3.44	89	2	7.46	86	3	7.14

Table 2. Comparison between OAR and NCR strategies on system performance

Benchmarks	OAR			NCR			Gain
	R_hop	mux_ratio	Freq(MHz)	R_hop	mux_ratio	Freq(MHz)	
CPU50_occ30	0	9	29.41	0	9	29.41	0%
CPU125_occ50	2	16	16.66	1	16	20	20.04%
CPU150_occ30	3	24	12.82	1	29	15.62	21.84%
CPU150_occ50	2	51	10.41	1	51	11.62	11.65%
CPU375_occ80	2	51	10.41	1	51	11.62	11.65%
CPU375_occ85	2	79	8.06	2	69	8.77	8.8%
CPU700_occ80	2	134	5.61	2	109	6.49	15.68%

5 Experimental Results

We use our benchmark generator [11] to generate several synthetic designs. The targeted multi-FPGA prototyping board we use for the experiments is a DNV6F6PCIe from the DINI group [12]. The inter-FPGA clock frequency is set to 500MHz. To map the designs into this board, we use the WASGA partitioning flow provided by Flexras Technologies [10]. WASGA partitions the designs and outputs the list of inter-FPGA signals that shoud be routed. After routing these signals, WASGA generates a netlist for each FPGA. The resulting netlists are re-entered into the FPGA flow to execute the place and route and the bitstream generation individulally for each FPGA.

Table 1 shows the results for each routing scenarios described in section 4. These scenarios are defined depending on the signal shape and the routing graph.

- In the scenario 1, multiterminal signals are routed on a unidirectional routing graph.
- In scenario 2, two-terminal branchs are routed into a unidirectional routing graph.
- In scenario 3, Signals are grouped into GSignals and routed into a bidirectional graph.
- In scenario 4, Branches are combined into groups and routed into a bidirectional graph.

In this experiment, we used benchmarks where 70% of signals are multiterminal ones. Results show that routing on a bidirectional graph gives much better results since the router is more free to select the routing path. On the other hand, routing multiterminal signals is not always optimized even if the mux_ratio of scenario 3 is sometimes less than the one of scenario 4, but using more routing hop penalizes the system frequency.

Since we have demonstrated that Senario 4 gives usually the best results, we apply Pathfinder and the obstacle avoidance routing algorithms to route inter-FPGA signals, all with one source and one destination (branch) and grouped into GSignals. Table 2 shows the results of comparison. OAR means Obstacle Avoidance Routing and NCR refers to Negotiated Congestion Routing. Results show the important impact of the NCR iterative routing and its efficiency to improve system performance. The frequency is increased on average by 12.8% and the impact of NCR is important for highly congested partitioning results. In fact thanks to its iterative aspect, it avoids easily local minima and reduces the path length from a source FPGA to a destination FPGA. In addition, it leads to a good tradeoff between maximum multiplexing ratio and routing hops.

6 Conclusion

Prototyping is no longer optional due to the cost of chips and difficulty to simulate huge designs. To get a design for prototype more efficient, the highest frequency should be reached. The system frequency depends on the way the inter-FPGA signals are routed. n this paper, we presented our approach to route these inter-FPGA signals. We extend the Pathfinder routing algorithm to deal with the inter-FPGA signals. These signals are grouped into GSignals where each one has 1 source and only 1 destination. Compared to common obstacle avoidance algorithms, we obtain a significant prototyping system frequency improvement of 12.8%.

References

1. Tessier, R., et al.: The virtual Wires Emulation System: A Gate-Efficient ASIC Prototyping Environement. In: International Workshop on Field-Programmable Gate Array. ACM, Berkeley (February 1994)
2. Inagi, M., Takashima, Y., Nakamura, Y.: Globally optimal time-multiplexing in inter-FPGA connections for accelerating multi-FPGA systems. In: Proc. FPL, pp. 212–217 (2009)
3. Huang, C., Yin, Y., Hsu, C.: SoC hw/sw verification and validation. In: Proc. of the 16th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 297–300 (2011)
4. Krupnova, H.: Mapping multi-million gate SoCs on FPGAs: industrial methodology and experience. In: Proc. of Design, Automation and Test in Europe Conference and Exhibition, vol. 2, pp. 1236–1241 (2004)

5. Asaad, S., Bellofatto, R., Brezzo, B., Haymes, C., Kapur, M., Parker, B., Roewer, T., Saha, P., Takken, T., Tierno, J.: A cycle-accurate, cycle reproducible multi-FPGA system for accelerating multi-core processor simulation. In: Proc. of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 153–162 (2012)
6. McMurchie, L., Ebeling, C.: PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In: International Workshop on Field Programmable Gate Array (1995)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Cambridge, Massachusetts London, England (2001)
8. Ejnioui, A., Ranganathan, N.: Multiterminal net routing for partial crossbar-based multi-FPGA systems. IEEE Trans. Very Large Scale Integr (VLSI) Syst. 11(1), 71–78 (2003)
9. Mak, W., Wong, D.: Board-level multiterminal net routing for FPGA-based logic emulation. ACM Trans. Design Automation of Electron. Syst. 2, 151–157 (1997)
10. `http://www.flexras.com`
11. Turki, M., Marrakchi, Z., Mehrez, H., Abid, M.: Towards Synthetic Benchmarks Generator for CAD Tool Evaluation. In: 8^{th} Confrence on Ph.D. Research in Microelectronics and Electronics (PRIME) (2012)
12. `http://www.dinigroup.com/new/products.php`
13. Santarini, M.: ASIC prototyping: Make versus buy. EDN (November 21, 2005)
14. Kuon, I., Rose, J.: Measuring the gap between FPGAs and ASICs. In: International Symposium on Field-Programmable Gate Array (February 2006)

Dependability-Increasing Method of Processors under a Space Radiation Environment

Yuya Shirahashi and Minoru Watanabe

Electrical and Electronic Engineering, Shizuoka University
tmwatan@ipc.shizuoka.ac.jp

Optically reconfigurable gate arrays (ORGAs) can be reconfigured using error-inclusive configuration contexts under a radiation-rich space environment. Therefore, the ORGA presents an important benefit: the allowable amount of configuration data damage is greater than that by field programmable gate arrays (FPGAs) with error-checking and correction. However, the ORGA's programmable gate array itself is never as robust against space radiation as that of an application-specific integrated circuit (ASIC) because the programmable architecture of its gate array is the same as that of FPGAs. Therefore, to achieve a drastic increase in the robust capability of a fine-grained programmable gate array on an ORGA-VLSI, this paper presents a proposal of a novel dynamic module multiple redundancy scheme based on a mono-instruction set computer architecture exploiting high-speed dynamic reconfiguration.

An ORGA can realize nanosecond-order reconfiguration and numerous reconfiguration contexts, thereby allowing clock-by-clock dynamic reconfiguration. Such dynamic reconfiguration enables the implementation of mono-instruction set computers (MISCs) onto programmable devices. If the MISCs are used, then the number of processors for majority voting can be increased, thereby enhancing its robust capability.

Here, 11 kinds of 32-bit MISC processors of five-module redundancies were implemented onto a Cyclone IV EP4CE115 FPGA. The five-module redundancy of each MISC processor incorporates the same five 32-bit MISC processors and five majority voting circuits. Here, to compare the performance of the MISCs with that of conventional RISC processors, a triple-module redundancy of 32-bit original soft-core RISC processors including all 11 instructions was designed. Average implementation areas of MISC processors were 1,612 logic elements, although conventional triple-module redundancy RISC processor consumed 9,236 logic elements. Under MISC implementation, the implementation area of five-module redundancy was 5.7 times lower than that of the triple-module redundancy conventional soft-core RISC processor. Furthermore, the average clock frequency of MISC processors is 35 times greater than that of a conventional soft-core RISC processor. Therefore, its performance has also been improved in addition to its dependability. Consequently, using MISC processors, the dependability of a fine-grained programmable gate array can be increased.

Acknowledgment. This research is supported by the Ministry of Internal Affairs and Communications of Japan under the Strategic Information and Communications R & D Promotion Programme (SCOPE).

Ant Colony Optimization for Application Mapping in Coarse-Grained Reconfigurable Array

Li Zhou, Dongpei Liu, Botao Zhang, and Hengzhu Liu

Institute of Microelectronics & Microprocessor, School of Computer Science, National University of Defense Technology, Changsha, China
{zhouli06,liudongpei,botaozhang,hengzhuliu}@nudt.edu.cn

Coarse-grained reconfigurable array (CGRA) is an efficient architecture in digital signal processing domain. It is one of the best candidate architectures to exploit instruction level and loop level parallelism in the computation intensive applications while maintaining a certain degree of flexibility. The performance of CGRA is greatly reliant on the mapping algorithm which associate operations with PE and the time slot to execute.

The mapping problem for both data acyclic and cyclic application kernels have been proved to be NP-complete. Iterative improvement algorithms such as simulated annealing (SA) and genetic algorithm (GA) have been adopted in mapping loops onto CGRA [1]. However, little work has been done on mapping data acyclic kernels onto CGRA efficiently. In our work, the ant colony optimization (ACO) is applied to map applications kernel represented by data acyclic graph (DAG) onto CGRA.

In ACO, ants construct a solution by deciding operation i in the DAG to be executed on PE j step by step. The probability of choosing a decision $p(i, j)$ is calculated by the global pheromone $\tau(i, j)$ and the local expectation $\eta(i, j)$, the benefit of mapping i to j at current step. The $\eta(i, j)$ is related to the earliest time that operation i can be executed on PE j. We use maze route technique to find the routing path from data producer to consumer. $\tau(i, j)$ is updated at the end of exploration according to the quality of solutions. Thus, the algorithm will converged to an optimized result after serval iterations.

We also present a heuristic which confines the exploration space to reduce mapping time with little performance loss. It was showed that the affinity between 2 operations greatly influence the result of mapping. So, if a decision causes two operations which have parent-child relationship or common offsprings too far from each other, it won't be chosen. Experiments shows our method generates high quality solutions (11% improved compared to GA and SA). 70% compilation time is saved due to the heuristic we proposed.

Reference

1. B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Dresc: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. of the Intl. Conf. on Field-Programmable Technology (FPT)*, 2002, pp. 166–173.

C-Based Adaptive Stream Processing on Dynamically Reconfigurable Hardware: A Case Study on Window Join

Eric Shun Fukuda[1], Hideyuki Kawashima[2], Hiroaki Inoue[3], Taro Fujii[4], Koichiro Furuta[4], Tetsuya Asai[1], and Masato Motomura[1]

[1] Hokkaido University, Sapporo, Japan
[2] University of Tsukuba, Tsukuba, Japan
[3] NEC Corporation, Kawasaki, Japan
[4] Renesas Electronics Corporation, Kawasaki, Japan

Stream processing is becoming an important application field for reconfigurable architectures due to its two trends: 1) data rate and power consumption of a stream processing system is rapidly increasing, and 2) data streams have wide characteristics that should be dealt with different hardware architectures. However, utilizing reconfigurable processors requires hardware development skills which software engineers who develop algorithms do not have in general.

In our research, we consider window join, one of the operators of stream processing, as a case study and illustrate the current situation in two viewpoints: a) how a software engineer in this field can facilitate hardware acceleration utilizing a reconfigurable hardware with a C-level programming environment, and b) how adaptiveness is achieved in stream processing in such a solution. We use a dynamically reconfigurable architecture that features a state-of-the-art high level synthesis tool which compiles C source code into a hardware configuration.

We carried out a step-by-step optimization starting from a simple and ordinary software code. As a result, we achieved throughput improvement by two orders of magnitude compared to the first step, and roughly 50 times greater throughput/power efficiency than a pure software solution running on an Intel Core i5 CPU. We also evaluated two different architectures with each specialized in high/low match rate data streams. The architecture specialized in low match rate scored 45% higher throughput than the other when the match rate was 0.01%, while the latter scored 62% higher when the match rate was 10%.

We conclude that: i) a stream processing system with a dynamically reconfigurable hardware acceleration can be developed entirely in C, and achieve much higher throughput at a higher power efficiency, and ii) dynamically reconfigurable hardware is an efficient way to deal with streams with changing characteristics. However, there were still occasions that hardware design skills were required during the optimization steps. Therefore, we suggest that establishing a stream-oriented programming model which further hides hardware details and provides a relatively high performance without optimizing the code is necessary.

Acknowledgements. This work was partially supported by JSPS Grant-in-Aid for Challenging Exploratory Research (No. 24650033).

Automatic Design Flow for Creating a Custom Multi-FPGA Board Netlist

Qingshan Tang[1], Matthieu Tuna[2], Zied Marrakchi[2], and Habib Mehrez[1]

[1] LIP6, Université de Pierre et Marie Curie, Paris, France
{qingshan,habib.mehrez}@lip6.fr
[2] Flexras Technologies, Paris, France
{matthieu.tuna,zied.marrakchi}@flexras.com

Abstract. Field-Programmable Gate Array (FPGA) is widely used for implementing digital circuits due to its moderately high level of integration and rapid turnaround time. As the cost of masks in IC fabrication is increasing and the performance gap between FPGA and ASIC is reducing, more and more systems will be implemented with FPGAs. A multi-FPGA board, which is a collection of FPGAs, is used in case that the logic capacity of a single FPGA is insufficient.

Nowadays, creating a custom multi-FPGA board is mainly a manual process. In this paper, an automatic design flow is proposed to create a multi-FPGA board netlist which is tailored to a specific design. In order to propose an automatic design flow, we assume for this study: 1) All the FPGAs used in the board to implement logic elements of the designs have the same FPGA type (vendor, family, device and package). 2) No information about the position of the FPGAs in the board and no PCB layout generated. The proposed design flow has three steps: the design partitioning, the interconnect synthesis and the design routing. The design partitioning and the design routing can be done respectively by commercial tools provided by Flexras Technologies. An algorithm is developed to do the interconnect synthesis which distributes the inter-FPGA connections according to cut signals of the partitioned design.

For the purpose of this study, a testbench generator has been developed. In the experiments, several designs are generated by the testbench generator and several board netlists are generated for each design by varying the used FPGA type. Therefore, the automatic design flow could lower the entry barrier for new board designer. Results show that the proposed automatic design flow reduces significantly the development time, and accelerates the time-to-market of new products with the optimized first-chip cost and improved system frequency. Pareto-optimal solutions for a given design can be selected instantly among the generated board. Then, the most adapted one can be chosen according to specifications of board designers among the Pareto-optimal solutions. When comparing with an off-the-shelf board, the best case achieves a 150% increase in the system frequency with less number of FPGA in the board.

Due to that there are many different existing FPGAs in the market, the FPGA type used in the board influences the system frequency. The future work will predict the most adapted custom multi-FPGA board for a given hierarchical design.

Pipeline Optimization for Loops on Reconfigurable Platform

Qi Guo[1], Chao Wang[1], Xuehai Zhou[2], and Xi Li[2]

[1] Suzhou Institute of Advance Study, USTC, Suzhou, Jiangsu, China
{gqustc,saintwc}@mail.ustc.edu.cn

[2] University of Science and Technology of China, Hefei, Anhui, China
{xhzhou,llxx}@ustc.edu.cn

Abstract. Pipelining is an effective technique to improve the performance of a loop by overlapping the execution of several iterations, particularly on the reconfigurable platform, which is more coarse-grained. In this paper, we use reconfigurable platform to accelerate loop based applications by reconstructing the pipeline structure during the execution of application. Based on this concept, the optimized strategies such as duplexing and splitting of function unit are applied from instruction level to task level. First, a loop is abstracted as a weighted data flow graph (WDFG), where nodes represent tasks while edges stand for inter-task dependencies. The weights of nodes and edges indicate task execution times and communication overheads respectively. Based on the abstraction, we propose an algorithm which automatically maps the pipelined loops onto reconfigurable hardware and select whether the duplexing or splitting is more appropriate. The algorithm is based on profiling information of WDFG, such as execution times and communication overheads. Then several test cases from EEMBC benchmark are selected to evaluate our approach. The evaluation is demonstrated in two ways. First, we operate some software simulations to appraise the effectiveness of the algorithms. Second, a prototype system is implemented on state-of-the-art FPGA board to evaluate the practicability of our approach on reconfigurable platform. Performance indicators of pipeline such as speedup, throughput and efficiency are measured in both ways. Moreover, in software simulation, the speedup and throughput rate of optimized pipeline achieved to 2 times at least and the efficiency increased by 1.1-1.5 times, whilst in hardware platform, the speedup and efficiency increase by 1.5 times due to the communication cost and reconfiguration delay, the throughput rate also increases by 1.5 to 2 times. Experimental results demonstrate that our approach can achieve satisfactory performance both on effectiveness and practicality.

Acknowledgements. This work was supported by the National Science Foundation of China under grants (No. 61272131, No. 61202053), China Postdoctoral Science Foundation (No. BH0110000014), and Jiangsu Provincial Natural Science Foundation (No. SBK201240198).

FPGA-Based Adaptive Data Acquisition Scheduler-on-Chip (SchoC) for Heterogeneous Signals

Mohammed Abdallah

State University of New York Institute of Technology, NY, USA

Abstract. Data acquisition (DAQ) is a crucial component in instrumentation and control. It typically involves the sampling of multiple analog signals, and converting them into digital formats so that they can be processed. DAQ systems also involve microprocessors, microcontrollers, digital signal processing, and/or storage devices. Many multi-channel DAQs, which utilize some sort of processing for simultaneous input channels, are found in various applications. In this research, for heterogeneous multi-channel signals, different sampling rates are identified for each channel, and optimized for best data quality with minimal storage requirement. Accordingly, power consumption and transmission times can be reduced. The fidelity of the proposed Scheduler-on-Chip (SchoC) is increased by using reconfigurable chip technology, where flexibility, concurrency, speed and reconfiguration can be achieved in hardware. Therefore, SchoC can be utilized in various real world applications especially hazardous environments, or for remote architecture reconfiguration, while keeping the cost of the device low. Preliminary performance evaluations show improvements in the speed and memory requirements of the proposed SchoC over a comparable software-based scheduler.

The proposed SchoC can be utilized in many applications since different applications need to schedule multi-rate tasks on a single core processor. The proposed SchoC can be used in a low cost, high performance multi-channel DAQ system for custom sensing applications which have a matrix of different analog sensors. Alongside with analog sensors (voltage, current or charge output), quasi-digital sensors and transducers (frequency, period, PWM output) can use the proposed SchoC. Moreover, in the environmental measurements, the operational satellite synthetic radar was used to acquire different data at different frequencies in order to estimate the soil moisture in different sites on the earth. The system needs to acquire many signals within a short period of time due to the uncontrollable changing site conditions. In addition, it can be used in a versatile instrument that can be the base of developing a spectroscopic imaging.

High Level FPGA Modeling of an JPEG Encoder

Gabriel Nunez, Evan Tsai, Airs Lin, Aleksander Milshteyn, G. Herman,
Helen Boussallis, and Charles Liu

Department of Electrical and Computer Engineering
California State University, Los Angeles
5151 State University Drive, Los Angeles, CA 90032 USA
gnunez85@gmail.com

A high-level Field Programmable Gate Array (FPGA) prototype for a JPEG image encoder has been developed by the Structures, Pointing, and Control Engineering (SPACE) University Research Center (URC). The FPGA prototype uses MathWorks Simulink and Xilinx System Generator for deployment on its Virtex-5 FPGA board. The FPGA module serves as a co-processor of a real-time Ubiquitous Video Conferencing (UVC) software package. The project objective is to: 1) study the high-level synthesis method for FPGA design, and 2) provide the digital signal processing necessary to offload computationally-intensive video processing data from a host computer running the UVC application [1]. The project uses a Xilinx XUPV5-LV110T FPGA Development System. The image compression utilizes the JPEG standard. Each module for the JPEG encoder is designed in Simulink. To reduce computation time on the FPGA due to floating-point Discrete Cosine Transform (DCT) calculations, a table containing DCT cosine product values was computed in Excel, accurate to two decimal digits. These values are shifted 6 bits to the left to maintain accuracy and loaded onto the FPGA. The product of 8-bit YCbCr channel data and tabulated values is accumulated into a register, then scaled and shifted 6 bits to the right to complete the DCT.

Efficiency of the design is measured against an open-source encoder from OpenCores.com, which does not rely on proprietary IP cores. The use of proprietary IP cores by System Generator utilizes one tenth the Slice Registers and occupied Slices, and 1/20th the Slice LUTs as the open source encoder. Based on the specifications of the two encoders, System Generator encoder speed outperforms the open source encoder after processing 1.44 blocks of image data. Based on the complexity of the problem and the resource availability on the targeted FPGA board, this encoder can produce 1080p video at 50 frames per second in real-time. The encoder that uses the System Generator demonstrates appreciable gains in space and speed over the open-source encoder. System Generator demonstrates a significant advantage to hardware design that can be a viable alternative to traditional HDL programming. Future work may include the parallel implementation of the quantization phase.

Acknowledgments. This research was made possible by the directors at the Structures, Propulsion and Control Engineering University Research Center (SPACE URC) as part of NASA research grant NCC NNX08BA44.

References

1. "Structures, propulsion, and control engineering space center," California State University, Los Angeles, 2012. [Online]. Available: http://www.calstatela.edu/orgs/space/

Empirical Evaluation of Fixed-Point Arithmetic for Deep Belief Networks

Jingfei Jiang[1], Rongdong Hu[1], Mikel Luján[2], and Yong Dou[1]

[1] National University of Defense Technology, ChangSha Hunan 410073, China
jingfeijiang@nudt.edu.cn
[2] University of Manchester, Manchester, M13 9PL, UK

Abstract. Deep Belief Networks (DBNs) are state-of-art Machine Learning techniques and one of the most important unsupervised learning algorithms. Training DBNs is computationally intensive which naturally leads to investigate FPGA acceleration. Fixed-point arithmetic can be used when implementing DBNs in FPGAs to reduce execution time, but it is not clear the implications for accuracy. Previous studies have focused only on accelerators using some fixed bit-widths. A contribution of this paper is to demonstrate the bit-width effect on various configurations of DBNs in a comprehensive way by experimental evaluation. Our work is inspired by the original DBN built on a subset of neural networks known as Restricted Boltzmann Machine (RBM) and the idea of Stacked Denoising Auto-Encoder (SDAE). We modified the floating-point versions of the original DBN and the denoising DBN (dDN) into fixed-point versions and compared their performance. Explicit performance changing points are found using various bit-widths. The results indicate that different configurations of DBNs have different performance changing points. The performance variations of three layers DBNs are a little larger than one layer DBNs because of the better sensitivity of deeper DBN. Sigmoid function approximation methods must be used when implementing DBNs in FPGA. The impacts of Piecewise Linear Approximation of nonlinearity algorithms (PLA) with two different precisions are evaluated quantitatively in our experiments. Modern FPGAs supply built-in primitives to support matrix operations including multiplications, accumulations and additions, which are the main operations of DBNs. A solution of mixed bit-widths DBN is proposed that a narrower bitwidth can be used for neural units and a wider one can be used for weights, thus fitting the bit-widths of FPGA primitives and gaining similar performance to the software implementation. Our results provide a guide to inform the design choices on bit-widths when implementing DBNs in FPGAs documenting clearly the trade-off in accuracy.

Keywords: deep belief network, fixed-point arithmetic, bit-width, FPGA.

Deriving Resource Efficient Designs Using the REFLECT Aspect-Oriented Approach

(Extended Abstract)

José G.F. Coutinho[1], João M.P. Cardoso[2], Tiago Carvalho[2], Ricardo Nobre[2], Sujit Bhattacharya[3], Pedro C. Diniz[4], Liam Fitzpatrick[5], and Razvan Nane[6]

[1] Department of Computing, Imperial College London, UK
[2] Faculdade de Engenharia da Univ, do Porto, Dep. Eng. Informática, Portugal
[3] Department of Electrical and Electronic Eng., Imperial College London, UK
[4] INESC-Investigação e Desenvolvimento, Lisboa, Portugal
[5] ACE Associated Compiler Experts bv, The Netherlands
[6] Technical University of Delft, The Netherlands
`www.reflect-project.eu`

In the context of the REFLECT project[1] we have developed an aspect-oriented compilation and synthesis toolchain that aims at facilitating the mapping of applications described in high-level imperative programming languages, such as C, to heterogeneous and configurable computing systems. More specifically, we have designed an aspect-oriented domain-specific language, called LARA[2], that allows programmers to convey application-specific and domain-specific knowledge as a way to capture non-functional concerns. The LARA specifications and the subsequent control of the tools via a code weaver allows a seamless exploration of alternative designs and run-time adaptive strategies, in effect enabling design-space exploration (DSE).

Figure 1 depicts a specific instantiation of the REFLECT aspect-oriented design-flow, which generates resource-efficient Xilinx designs from C kernels and LARA descriptions. This design-flow operates as follows. There are 3 main inputs: (1) C application sources, (2) input parameters that control which and how kernels are synthesized to hardware, and (3) LARA aspects that capture the DSE strategy that derives the final designs. The DSE weaver invokes the Harmonic weaver to compute the word-lengths of variables based on user-provided parameters, such as input ranges and required output accuracy. The results of the word-length analysis are captured as a LARA aspect and passed down to the Reflectc weaver, which controls the CoSy [3] engines. The word-length information contains the precision of each variable, including the minimum number of bits for the integer, fraction and signal that satisfy the target output accuracy. The Reflectc weaver receives the LARA aspect with the word-length information and the C kernel, and uses the weaveshrink CoSy engine [3] to change data-types from floating-point to fixed-point using the specified word-length information. Next, the weaver invokes the DWARV code generator to derive the optimized design in VHDL. Finally, the DSE weaver invokes the Xilinx ISE tools to generate the corresponding hardware design.

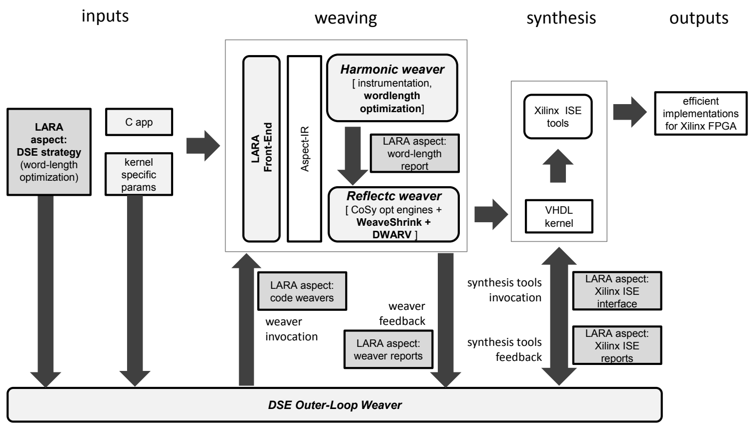

inputs
weaving
synthesis
outputs
LARA aspect: DSE strategy (word-length optimization)
C app
kernel specific params
LARA Front-End
Aspect-IR
Harmonic weaver [instrumentation, wordlength optimization]
LARA aspect: word-length report
Reflectc weaver [CoSy opt engines + WeaveShrink + DWARV]
Xilinx ISE tools
VHDL kernel
efficient implementations for Xilinx FPGA
LARA aspect: code weavers
weaver invocation
weaver feedback
LARA aspect: weaver reports
synthesis tools invocation
synthesis tools feedback
LARA aspect: Xilinx ISE interface
LARA aspect: Xilinx ISE reports
DSE Outer-Loop Weaver

Fig. 1. The REFLECT Aspect-Oriented Design-Flow

Next we present a LARA aspect description that captures a word-length strategy that operates on the design-flow shown in Fig. 1. This aspect receives a list of target accuracies (line 6), and generates a set of FPGA implementations that satisfy the computation requirements of each element of the list, given a set of C sources and a function name (line 4):

```
 1 import xilinx_opts, store_results;
 2 aspectdef wlot_strategy
 3    input // kernel params
 4       csource, cflags, kernel_name,
 5       input_ranges, // e.g. { x: { min: −1, max: 10} }
 6       targetAccs, // e.g. [3e−1, 9e−4, 9e−9],
 7       targetVar, maxFreq,
 8    end
 9    for (t in targetAccs) {
10       var targetAcc = targetAccs[t];
11       run(tool:'harmonic', args:['-aspLARA=wlot_harmonic.lara', ...]);
12       run(tool:'reflectc', args:['wlot.lara','kernel.c','-gen=vhdl']);
13       call xilinx_opts(folder:"VHDL", opt:4, ctrMaxFreq: maxFreq);
14       println("MaxFreq : "+ @design.CCU.maxFreq + " (MHz)");
15       dir = 'wcode' + t;
16       call store_results(dir);
17    }
18 end
```

Table 1 presents the results obtained with the REFLECT design-flow using Xilinx ISE tools [4] on a 3D path planning kernel [1], and targeting a Virtex-5 FPGA-based platform. The results compare the original single precision version of the kernel against 3 automatically derived fixed-point designs using the above aspect strategy with target accuracies of 7.0e-9, 6.0e-4 and 1.0e-1. These target accuracies generated the Q3.29, Q3.13 and Q3.5 fixed-point designs respectively. Qx.y represents a fixed-point representation with x integer bits and y fractional bits. Considering the single floating-point precision design as the reference, we achieve area savings from 27.54% to 34.09% by dropping the computation accuracy from 5.58e-9 to 9.4e-2. Regarding the power savings, we achieve dynamic power savings from 21.14% to 35.02% by dropping the computation accuracy from 5.58e-9 to 9.4e-2, when running the design at 300 MHz.

Table 1. FPGA resources reported for various data type representations

	accuracy		area		power	
	target accuracy	computation accuracy	slices	reduction slices (%)	dynamic power @300Mhz (mW)	reduction dynamic power @300Mhz (%)
single precision	-	4.2e-7	748	-	142.93	-
Q3.29 (unsigned)	7.0e-9	5.58e-9	542	27.54%	112.72	21.14%
Q3.13 (unsigned)	6.0e-4	3.6e-4	498	33.42%	98.9	30.81%
Q3.5 (unsigned)	1.0e-1	9.4e-2	493	34.09%	92.87	35.02%

These results show a clear trade-off between accuracy and resource utilization, and between accuracy and dynamic power. By decoupling non-functional concerns (expressed in LARA) with functional concerns (implemented in C), we can easily revise the computational requirements and even the strategy itself to automatically derive new solutions. Furthermore, this approach allows the development of compilation strategies that can be re-used and applied to different applications and possibly different target architectures, thus increasing design productivity for the same target as well as code portability across multiple and very distinct target architectures.

Acknowledgements. This work was partially supported by the European Community's Framework Programme 7 (FP7) under contract No. 248976, 257906 and 287804, and UK EPSRC. The authors are grateful to the members of the REFLECT project for their support.

References

1. REFLECT Project (2013), http://www.reflect-project.eu
2. Cardoso, J.M.P., Carvalho, T., Coutinho, J.G.F., Luk, W., Nobre, R., Diniz, P.C., Petrov, Z.: LARA: An Aspect-Oriented Programming Language for Embedded Systems. In: Proc. of the ACM Intl Conf. on Aspect-Oriented Software Development (AOSD 2012) (March 2012)
3. ACE CoSy® Compiler Development System (2012), http://www.ace.nl/compiler/cosy.html
4. All Programmable Technologies from Xilinx Inc. (2011), http://www.xilinx.com

Embedded Reconfigurable Architectures

Stephan Wong

Computer Engineering Laboratory,
Delft University of Technology,
J.S.S.M.Wong@tudelft.nl
http://www.era-project.eu/

Abstract. ERA (Embedded Reconfigurable Architectures) is a 3-year project funded by the EU under the FP7 framework programme (starting January 2010)[1]. The ERA project addresses issues rising from a scenario in which the complexity and diversity of embedded systems is rising and causing extra pressure in the demand for performance at the lowest possible power budget and in which designers face the challenges brought by the power and memory walls in the production of embedded platforms. The focus of the ERA project is to investigate and propose new methodologies in both tools and hardware design to break through these walls and help design the next-generation embedded systems platforms. The proposed strategy is to utilize adaptive hardware to provide the highest possible performance with limited power budgets. The envisioned adaptive platform employs a structured design approach that allows integration of varying computing elements, networking elements, and memory elements. More precisely, we focused on the dynamic adaptation of our platform (of the computing, networking and memory elements) to the software requirements and the operating environment targeting performance, power/energy, and resource availability.

1 The Technical Approach

In detail, the ERA project focuses on three hardware components (**processor core**, **memory hierarchy**, and **network-on-chip**) in the design of an embedded system platform without losing sight of the requirements posed by the **(embedded) applications**. The adaptability of the envisioned platform opens up new **compilation strategies** and **scheduling techniques** (either in hardware or by an operating system) to deal with this increased freedom. In the following, we summarize the major results within the ERA project:

– the development of a new dynamically reconfigurable parameterized processor core that can adapt itself to the application needs or the power budget at hand through parameters such as issue width, register file size, type and

[1] The partners are: (1) Delft Univ. of Technology (NL) - coord., (2) Industrial Systems Institute (GR), (3) Universita' degli Studi di Siena (IT), (4) Chalmers Univ. (SE), (5) Univ. of Edinburgh (UK), (6) Evidence (IT), (7) ST Micro (IT), (8) IBM (IL), (9) Universidade do Rio Grande do Sul (BR), and (10) Uppsala University (SE).

number of functional units, and memory bandwidth. For example, when an application (thread) exhibits a high level of parallelism, more issue slots, a larger register file, and more functional units are instantiated to better execute the application (thread).

– the definition of a dynamically adjustable memory hierarchy that can be tailored to the application behavior in order to improve the utilization of the available memory bandwidth and perform these adjustments through smooth transitions. Example parameters include: cache size, cache line size, and set associativity.
– the development of a dynamically reconfigurable network-on-chip (NoC) that can adjust itself to the traffic that is flowing through its nodes. In more detail, the NoC allocates more resources to the nodes when it determines that certain paths in the network are more frequently taken.
– the characterization of embedded applications (working on top of an OS) and mapping of these characteristics to the parameters of the hardware components. This work is performed by identifying phases within applications that will benefit from different hardware configurations and therefore, require the need for dynamic reconfiguration. In this characterization, we used additional tools such as COTSon, McPAT, Paraver, and QEMU.
– the development of new compilation techniques to deal with the dynamic behavior of the hardware components in order to better utilize the available resources without losing generality in generating application software.
– the development of an OS environment to work in conjunction with a hardware scheduler to manage the hardware resources and schedule a given set of applications that need to be executed.

2 Results and Conclusions

These results have been integrated into two fully functional toolchains (for the computing elements) and three independent hardware platforms. The first toolchain is based on the VEX compiler from HP - as the ρ-VEX follows the VEX ISA - that contains both a compiler (based on Multiflow) and a simulator. We developed an assembler and linker that can convert the output of this compiler to actual binaries to be used by the actual softcore or by the xSTsim simulator (developed within the ERA project). The second toolchain is based on GCC and using the same assembler and linker we can generate executables. Moreover, the xSTsim simulator also simulates the memories and network-on-chip employing the new algorithms we developed within ERA. The first platform comprises a ρ-VEX processor with its own memory hierarchy. The second platform comprises several ρ-VEX processors connected together using an NoC. The third platform comprises a MicroBlaze running a Linux distribution controlling several ρ-VEX processors acting as accelerators. We have demonstrated in our ERA project that with with new techniques in dynamic adaptation, a more refined and gradual trade-off can be achieved between performance, power, and resource utilization.

Algorithm Design Methodology for Embedded Architectures*

Kiran Kumar Matam[1] and Viktor K. Prasanna[2]

[1] Computer Science Department
[2] Ming Hsieh Department of Electrical Engineering
University of Southern California

Abstract. Power efficiency is a critical constraint for embedded systems. To address this many technological innovations are being proposed by the community. However, leveraging such advances also requires algorithmic solutions to handle the potential for run-time errors due to near threshold computing. In this work we plan to develop algorithmic innovations and optimizations for power efficiency in the emerging landscape of embedding computing platforms. We also plan to develop resilient run-time systems and incorporate resiliency in algorithmic solutions. In this paper we briefly describe our design methodology employed in the TAPAS (Tunable Algorithms for PERFECT Architectures) project. The TAPAS project is funded under the DARPA PERFECT (Power Efficiency Revolution for Embedded Computing Technologies) program.

1 Introduction

Many technological advances are being proposed by the community, which must be effectively exploited at the software and application layers to achieve and sustain the power envelope of a given application. Optimization at the algorithmic level has a much higher impact on total energy dissipation than microarchitecture or circuit level. Some recent studies have shown that the impact ratio is 20:2.5:1 for algorithmic, register, and circuit level energy optimizations. In this work we plan to develop algorithmic innovations and optimizations for power efficiency in the emerging landscape of embedding computing platforms. We plan to achieve the following objectives :

– Develop algorithmic optimizations for latency, throughput and energy performance,
– Identify opportunities ("knobs") to integrate these into the compilation capability and also enable overall application composition, and
– Demonstrate improved energy and resilience performance for signal processing kernels on next generation embedded computing platforms.

* This work has been funded by DARPA under grant number HR0011-12-2-0023.

2 Technical Approach

We plan to develop model-based algorithmic techniques [1] in which the design space can be explored for a given target embedded platform by considering various novel design time optimizations, coarse performance modeling and hierarchical design space exploration.

- **Tunability:** Our parallel algorithms for kernels will be specified using a small number of parameters including latency, energy, resiliency, input (problem) size, and number of processors. The methodology will permit the algorithm space to be explored by the designer by varying the parameters. Thus, we obtain tunable performance by varying these parameters and the algorithm execution specifies a surface in the three dimensional space of execution latency, energy consumed and achieved resiliency.
- **High level coarse performance modeling:** The space of embedded platforms is rather large. We do not expect one model to capture the key features of all target platforms or the features of a specific target platform to sufficient accuracy to explore the performance tradeoffs when mapping to that platform. Rather, we define a high level performance model called Integrated Computational Model (ICOM) which will be customized for each architecture-algorithm pair and a target platform. The model is static in the sense that the parameters are known and are estimated at design time. Also, it is a high level (coarse grained) model, by which we mean few parameters are used to get a coarse estimate of performance.
- **Exploring algorithm design space:** For a given kernel, many parallel techniques can be explored. Several parallel implementations for Discrete Fourier Transform (DFT) such as radix-2, radix-4, decimation in time or frequency can be specified as an architecture-algorithm pair. In our design methodology, each pair is explored for possible mappings onto the target platform by using ICOM. Thus, given a problem size and target platform resources, we explore the design space and generate a set of design choices for each value of latency, energy and resiliency. Detailed implementation or simulation is performed for these designs to choose an energy optimal one. We generate a surface of feasible solutions by varying performance parameters: latency, energy and resiliency.
- **Exploring various target architectures:** Our methodology is intended to model various target architecture features and explore them at the software layer. The features to be modeled can also depend on the characteristics of the architecture-algorithm pair. Thus, we will not develop a single model to capture all target embedded platforms or space of parallelizations, but rather use a customized model specific to a target embedded platform with parameters identified for that platform and the architecture-algorithm pair.

Reference

1. V. K. Prasanna, "Energy-Efficient Computations on FPGAs," *J. Supercomput.*, vol. 32, no. 2, pp. 139–162, May 2005.

Efficient Hardware Based Security Algorithm Implementation for WSN Medical Applications: The ARMOR Perspective

Christos Antonopoulos[1], George Krikis[2], and Nikolaos Voros[1]

[1] Technological Educational Institute of Mesolonghi, Greece
{cantonopoulos,voros}@teimes.gr
[2] Noesis Technologies, L.P., Patras, Greece
gkrikis@noesis-tech.com

Abstract. Utilization of emerging WSN technologies in the field of demanding medical applications comprise one of the most critical and challenging objectives of the ARMOR project. However, contemporary WSN node implementations are notorious for the resource limitations e.g. in terms of processing and memory capabilities. Therefore, significant effort is devoted in hardware based implementations of critical components with respect to the project objectives that can alleviate respective re-source limitations drawback.

1 Introduction

WSN communication paradigm offers high value features making respective platforms very appealing for a wide range of applications. Additionally, over the last few years platforms and technologies have emerged offering enhanced performance making them adequate candidates for demanding applications [1]. In the ARMOR project such an application is the primary focus in terms of both communication requirements as well as security [2]. The main goal of the ARMOR project is to enable accurate, reliable and non-intrusive monitoring and analysis of epilepsy-relevant multi-parametric data including EEG and ECG signals. Consequently on one hand aiming to study epilepsy requires a high number of sensors being able to acquire and transfer very high amount of data (especially concerning EEG measurements) and one the other hand, it is of paramount importance to offer high security services. The latter aspect requires the efficient execution of state of the art cipher algorithms. Considering the resource limitation exhibited by today's prominent platforms effort has been devoted in designing and implementing an ultra-low power AES hardware based cipher implementation in order to enhance the functionality of existing WSN platforms while not compromising required performance. Furthermore, in the context of the ARMOR project all aspects of an end-to-end EHR system (as depicted in Fig. 1) will be addressed including real time analysis at local site as well as off-line analysis through an EHR system offering advanced services.

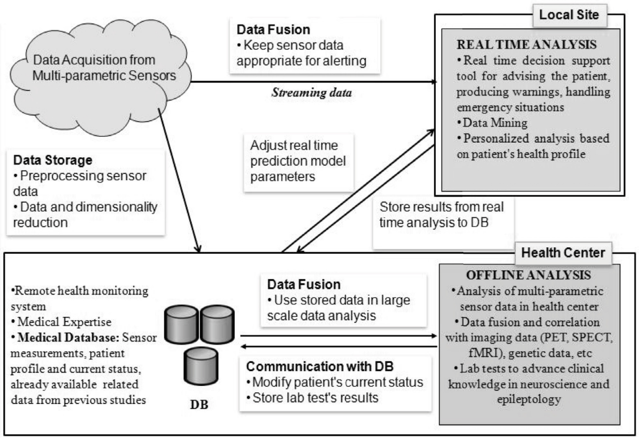

Data Acquisition from Multi-parametric Sensors
Data Fusion
• Keep sensor data appropriate for alerting
Streaming data
Local Site
REAL TIME ANALYSIS
• Real time decision support tool for advising the patient, producing warnings, handling emergency situations
• Data Mining
• Personalized analysis based on patient's health profile
Data Storage
• Preprocessing sensor data
• Data and dimensionality reduction
Adjust real time prediction model parameters
Store results from real time analysis to DB
Health Center
•Remote health monitoring system
• Medical Expertise
• Medical Database: Sensor measurements, patient profile and current status, already available related data from previous studies
DB
Data Fusion
• Use stored data in large scale data analysis
Communication with DB
• Modify patient's current status
• Store lab test's results
OFFLINE ANALYSIS
• Analysis of multi-parametric sensor data in health center
• Data fusion and correlation with imaging data (PET, SPECT, fMRI), genetic data, etc
• Lab tests to advance clinical knowledge in neuroscience and epileptology

Fig. 1. ARMOR Toolset Overview

2 Proposed Hardware Encryption Module

The hardware encryption module IP Core implements the 128-bit block size NIST FIPS-197 Advanced Encryption Standard (AES) algorithm [3]. High configurability is exhibited enabling the use of a 128/192/256 bit key, various operational modes (ECB, CBC, CFB, OFB and CTR) and optional key expansion. Effort has been devoted in minimum logic resources requirements rendering it as adequate solution for low power applications. It offers AES IP core solution exhibiting enhanced performance-silicon area ratio compared to relative to available industry implementation as well as efficient algorithm mapping techniques.

3 On-going and Future Hardware Based Implementations

Focusing on hardware based advancements on-going efforts are devoted to security aspects (i.e. decryption module) as well as the performance domain through compression algorithm design and implementation.

Acknowledgements. This study is partially funded by the European Commission under the Seventh Framework Programme (FP7/2007-2013) with grant ARMOR, Agreement Number 287720.

References

1. Tsou, T.-L.: Design, development and testing of a wireless sensor network for medical application. In: Proc. of the 7th Intl. Wireless Comm. and Mobile Computing Conf. (IWCMC) (July 2011)
2. `http://armor.tesyd.teimes.gr`
3. Conrad, E.: Advanced encryption standard. White Paper

Coarse Grained Parallelism Optimization for Multicore Architectures: The ALMA Project Approach

George Goulas[1], Christos Gogos[2], Christos Valouxis[1], Panayiotis Alefragis[1], and Nikolaos Voros[1]

[1] Technological Educational Institute of Mesolonghi, Greece
`{ggoulas,alefrag,voros}@teimes.gr, cvalouxis@gmail.com`
[2] Technological Educational Institute of Epirus, Greece
`cgogos@teiep.gr`

Abstract. In this paper, the coarse grained parallelism optimization step of the ALMA EU FP7 project is discussed. The current results look promising, as the possibility to use Integer Programming and provide optimal results to the problem model seems feasible and efficient.

Keywords: Coarse grain parallelization, Scliab, Integer Programming.

1 The ALMA Toolset

The ALMA toolset, major product of the ALMA EU FP7 project [1], as presented in Figure 1, provides an end-to-end solution from Scilab to embedded *Multi Processor System-on-Chip* (MPSoC) code. The ALMA *Architecture Description Language* (ADL) provides tools and description for the MPSoC platform and allows the toolset to be agnostic of specific platforms. The ALMA project driver platforms are based on Recore's reconfigurable DSP cores or KIT's Kahrisma cores. The ALMA-specific Scilab dialect source code is converted to C and then is loaded into the GeCoS open source compiler framework. The *fine-grain parallelism extraction* step targets the exploitation of the *Single Instruction, Multiple Data* (SIMD) instruction set of the underlying MPSoC architectures, addressing the data type selection and memory access aware vectorization problems. The *coarse-grain parallelism optimization* step analyzes and modifies the *Control and Data Flow Graph* (CDFG) in order to expose task level parallelism using the platform ADL description as well as feedback from the *ALMA Multi-Core Simulator* interface.

2 Coarse Grain Parallelism Optimization

The coarse-grain parallelism optimization step attempts to expose task level parallelism with respect to a global view of the program. A combination of loop fusion and tiling are employed to expose available tasks. The process that follows is iterative and uses feedback from the platform simulator. Clustering combines basic blocks into composite structures, named *hyperblocks*, in order to control search space size. These hyperblocks are further combined to produce *tasks*, with virtual input and output blocks that aggregate data and control dependences of the internal blocks. The tasks are then mapped and scheduled to processing cores.

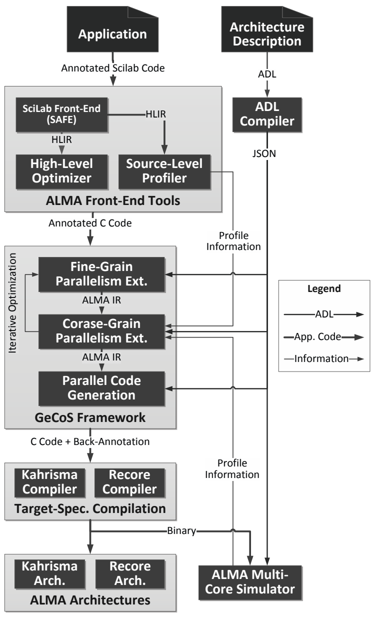


Fig. 1. ALMA Toolset Overview

The CDFG is transformed to a *Hierarchical Task Graph* (HTG) *Intermediate Representation* (IR). Each layer of this graph is a set of *Directed Acyclic Graphs* (DAGs). Each DAG is modeled as an *Integer Programming* (IP) problem, optimizing the critical path execution time. The IP model inputs are the set of processors, the set of tasks, the estimated running times for each task on each processor and the communication costs between every pair of dependent tasks. The IP model has been used to find optimal solutions for small problems available in the relevant literature. It should be mentioned that optimal solution to the HTG does not necessarily imply optimal solution of the original program.

The planned approach is to produce a sequence of IP problem solutions for each level of the HTG for all cardinalities of available processors. Lower-level solutions are used as alternative options for the higher-level HTG IP models. As we move higher in the HTG hierarchy, the problems grow bigger, so in order to control solution time and complexity, for bigger programs, lower levels of the solution may use heuristic algorithms.

3 Conclusions

In this paper, the coarse grained parallelism optimization step of the ALMA EU FP7 project is discussed. The current results look promising, as the possibility to use Integer Programming and provide optimal results to the problem model seems feasible and efficient.

Acknowledgements. This work is co-funded by the European Union under the 7th Framework Programme under grant agreement ICT-287733.

Reference

1. Architecture oriented paraLlelization for high performance embedded Multicore systems using scilAb, ALMA (2013), `http://www.alma-project.eu`

Author Index