Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

INSTRUCTION FUSION AND VECTOR PROCESSOR VIRTUALIZATION FOR HIGHER THROUGHPUT SIMULTANEOUS MULTITHREADED PROCESSORS

by
Yaojie Lu

The utilization wall, caused by the breakdown of threshold voltage scaling, hinders performance gains for new generation microprocessors. To alleviate its impact, an instruction fusion technique is first proposed for multiscalar and many-core processors. With instruction fusion, similar copies of an instruction to be run on multiple pipelines or cores are merged into a single copy for simultaneous execution. Instruction fusion applied to vector code enables the processor to idle early pipeline stages and instruction caches at various times during program implementation with minimum performance degradation, while reducing the program size and the required instruction memory bandwidth. Instruction fusion is applied to a MIPS-based dual-core that resembles an ideal multiscalar of degree two. Benchmarking using an FPGA prototype shows a 6-11% reduction in dynamic power dissipation as well as a 17-45% decrease in code size with frequent performance improvements due to higher instruction cache hit rates.

The second part of this dissertation deals with vector processors (VPs) which are commonly assigned exclusively to a single thread/core, and are not often performance and energy efficient due to mismatches with the vector needs of individual applications. An easy-to-implement VP virtualization technology is presented to improve the VP in terms of utilization and energy efficiency. The proposed VP virtualization technology, when applied, improves aggregate VP utilization by enabling simultaneous execution of multiple threads of similar or disparate vector lengths on a multithreaded VP. With a vector register

file (VRF) virtualization technique invented to dynamically allocate physical vector registers to threads, the virtualization approach improves programmer productivity by providing at run time a distinct physical register name space to each competing thread, thus eliminating the need to solve register name conflicts statically. The virtualization technique is applied to a multithreaded VP prototyped on an FPGA; it supports VP sharing as well as power gating for better energy efficiency. A throughput-driven scheduler is proposed to optimize the virtualized VP's utilization in dynamic environments where diverse threads are created randomly. Simulations of various low utilization benchmarks show that, with the proposed scheduler and power gating, the virtualized VP yields a larger than 3-fold speedup while the reduction in the total energy consumption approaches 40% compared to the same VP running in the single-threaded mode.

The third part of this dissertation focuses on combining the two aforementioned technologies to create an improved VP prototype that is fully virtualized to support thread fusion and dynamic lane-based power-gating (PG). The VP is capable of dynamically triggering thread fusion according to the availability of similar threads in the task queue. Once thread fusion is triggered, every vector instruction issued to the virtualized VP is interpreted as two similar instructions working in two independent virtual spaces, thus doubling the vector instruction issue rate. Based on an accurate power model of the VP prototype, two different policies are proposed to dynamically choose the optimal number of active VP lanes. With the combined effort of VP lane-based PG and thread fusion, compared to a conventional VP without the two proposed capabilities, benchmarking shows that the new prototype yields up to 33.8% energy reduction in addition to 40% runtime improvement, or up to 62.7% reduction in the product of energy and runtime.

INSTRUCTION FUSION AND VECTOR PROCESSOR VIRTUALIZATION FOR HIGHER THROUGHPUT SIMULTANEOUS MULTITHREADED PROCESSORS

by
Yaojie Lu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Electrical Engineering

Helen and John C. Hartmann Department of Electrical and Computer Engineering

May 2016

Copyright © 2016 by Yaojie Lu

ALL RIGHTS RESERVED

APPROVAL PAGE

INSTRUCTION FUSION AND VECTOR PROCESSOR VIRTUALIZATION FOR HIGHER THROUGHPUT SIMULTANEOUS MULTITHREADED PROCESSORS

Yaojie Lu

Dr. Sotirios G. Ziavras, Dissertation Advisor — Date
Professor of Electrical and Computer Engineering,
Associate Provost for Graduate Studies and Dean of the Graduate Faculty, NJIT

Dr. Durgamadhab Misra, Committee Member — Date
Professor of Electrical and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member — Date
Professor of Electrical and Computer Engineering, NJIT

Dr. Roberto Rojas-Cessa, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Alexandros V. Gerbessiotis, Committee Member — Date
Associate Professor of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author: Yaojie Lu

Degree: Doctor of Philosophy

Date: May 2016

Undergraduate and Graduate Education:

- Doctor of Philosophy in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2016

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2012

- Bachelor of Science in Physics,
  Fudan University, Shanghai, P. R. China, 2010

Major: Electrical Engineering

Presentations and Publications:

Y. Lu, S. Rooholamin and S.G. Ziavras, "Power-Performance Optimization of a Virtualized SMT Vector Processor via Thread Fusion and Lane Configuration," submitted to IEEE Computer Society Annual Symposium on VLSI, 2016.

Y. Lu, S. Rooholamin and S.G. Ziavras, "Vector Coprocessor Virtualization for Simultaneous Multithreading," accepted for publication, ACM Transactions on Embedded Computing Systems, March 2016.

Y. Lu and S.G. Ziavras, "Instruction Fusion for Multiscalar and Many-Core Processors," International Journal of Parallel Programming, Springer, 2015. DOI: 10.1007/s10766-015-0386-1.

Y. Lu and S.G. Ziavras, "Instruction Fusion for Multiscalar and Many-Core Processors," 12th IFIP International Conference on Network and Parallel Computing (IFIP and ACM SIGMICRO), New York, NY, September 17-19, 2015.

To my Family, with Love and Gratitude

在此，我希望用中文，把最特别最诚挚的谢意献给我的家人。感谢我的父母，陆禹先生以及王瑛女士,这些年来对我无条件的爱护和支持。是你们含辛茹苦养育了我，并用自己多年的积蓄帮我完成学业和梦想。你们是这个世界上最伟大的父亲和母亲。我也要感谢我亲爱的太太，戴旭斐女士。是你这些年来不离不弃的陪伴，让我有勇气和信心一直走到了今天。

ACKNOWLEDGMENT

In the first place, I would like to express my deepest gratitude to my adviser, Dr. Sotirios Ziavras, for being my mentor throughout my PhD research. As my advisor, he not only gave me great research related advice, but also inspired me to think independently and gave me the opportunity and confidence to pursue my own innovations. His extraordinary skills in academic writing and in depth knowledge in various fields really made the research process a wonderful learning experience. I wouldn't have learned and achieved this much without his time and patience.

I would like to thank Dr. Durga Misra for serving on my dissertation committee and also for his guidance and support during my master program. His VLSI courses have been of great value to me. I would also like to extend my sincere appreciation to Dr. Edwin Hou, Dr. Roberto Rojas-Cessa and Dr. Alexandros V. Gerbessiotis for serving as members of my dissertation committee. They were always willing to spend the time whenever I needed their help and advice.

Moreover, I am truly indebted and thankful to the ECE Department at NJIT. My work as PhD student would not have been possible without the teaching assistant award granted by the ECE Department.

Further thanks go to the staff of the office of global initiatives, the staff of graduate studies, and the staff of the ECE Department for their advice, help and support with administrative matters during my PhD studies and work as teaching assistant.

Additionally, I would like to thank my friends and colleagues, Seyedamin, Tim, Shiyang, Zhihao and Xiaolu for all the great and unforgettable moments we shared together during these years.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

1.1 The Utilization Wall 1

1.2 Inefficient VP Usage 2

1.3 Motivation and Objectives 5

2 RELATED WORK 9

2.1 Work Related to Instruction Fusion 9

2.2 Work Related to VP Virtualization 11

3 IMPLEMENTATION AND EVALUATION OF INSTRUCTION FUSION 15

3.1 How Instruction Fusion Works 15

3.1.1 Instruction Similarity 15

3.1.2 Instruction Fusion 17

3.2 Processor Implementation for Fused Mode Execution 19

3.2.1 MIPS-I Like Multiscalar Architecture 19

3.2.2 FPGA Implementation 22

3.3 Benchmarking for Instruction Fusion 22

3.3.1 Evaluation Procedure 22

3.3.2 Analysis of Results 23

4 VP VIRTUALIZATION 29

4.1 Virtual Register Name Space for Multithreading 29

4.2 The VRF Structure 30

TABLE OF CONTENTS
(Continued)

Chapter **Page**

4.3 The Vector Register Management Module (RMM) and Algorithm 31

4.4 Assigning/Releasing VRF Resources 34

4.5 Fragmentation Analysis 35

4.6 Performance of the VRF Management Algorithm 36

5 VIRTUALIZED VP ARCHITECTURE AND FPGA IMPLEMENTATION 39

5.1 The Host Subsystem Architecture 39

5.2 The VP Architecture 42

5.2.1 VP ISA and Pipeline 42

5.2.2 VP-MB Interface 45

5.2.3 Hazard Detection Unit (HDU) 46

5.2.4 Vector Lane Structure 47

5.3 FPGA Implementation 48

6 BENCHMARKING AND HOMOGENEOUS SMT 51

6.1 Benchmark Details 51

6.2 Homogeneous SMT Results 53

6.3 Comparison with Prior Works 58

7 SCHEDULING VECTOR THREADS 59

7.1 The Scheduling Algorithm 59

7.2 Queues of Fixed Length 61

7.3 Open System with Randomly Arriving Threads 63

TABLE OF CONTENTS
(Continued)

Chapter **Page**

8 VP ENERGY CONSUMPTION 67

8.1 VP Dynamic Power 67

8.2 Total Energy Consumption 70

9 VIRTUALIZED VP FOR THREAD FUSION AND DYNAMIC LANE CONFIGURATION 72

9.1 Virtualized VM Address Space 72

9.2 SMT VP and Thread Fusion 75

9.3 System Architecture and FPGA Implementation 79

9.4 Benchmarking 81

9.5 The Power Model 84

9.6 The Scheduling Policy 87

10 A PIPELINED INTER-LANE NETWORK FOR EFFICIENT DATA SHUFFLE 91

10.1 The Benefits of the Network 91

10.2 Structure of the Shuffle Network 92

10.3 The Decoder Virtualization Technique 96

11 CONCLUSIONS AND FUTURE WORK 99

11.1 Conclusions 99

11.2 Future Work 102

REFERENCES 104

LIST OF TABLES

Table		Page
3.1	Performance/Power/Energy Benchmarking (Normal and Fused Execution)	24
5.1	ISA of the VP	45
5.2	Resource Consumption of the VP Prototype	50
6.1	Matrix Multiplication Performance (Input Matrix Size: VL*VL, 1 Iteration per Core)	54
6.2	FIR Performance (Input Vector Size: VL, 1 Iteration per Core)	54
6.3	VDP Performance (Input Vector Size: VL, 1 Iteration per Core)	55
6.4	DCT Performance (Input: VL/8 Blocks of Size 8*8, 1 Iteration per Core)	55
6.5	RGB2YIQ Performance (Input: 1024 Pixels, 1 Iteration per Core)	55
6.6	Speedup Comparison with Prior Works	58
7.1	Detailed Results for a Schedule with Pending Thread Queue Length of 8	62
7.2	Detailed Results for a Schedule with Pending Thread Queue Length of 16	63
7.3	Characteristics of Chosen Tasks for an Open System	64
7.4	Detailed Task Arrivals and Execution Time for λ=0.5	65
7.5	Detailed Task Arrivals and Execution Time for λ=0.75	65
7.6	Detailed Task Arrivals and Execution Time for λ=1	66
8.1	Power and Energy Consumption for Benchmarks.	69
9.1	Resource Consumption and Utilization Percentage of the New VP Prototype	81
9.2	Performance Profile Data for Unfused VP with Four Active Lanes.	82
9.3	Performance Profile Data for Fused VP with Four Active Lanes.	82
9.4	Performance Profile Data for Unfused VP with Two Active Lanes.	82

LIST OF FIGURES

Figure **Page**

3.1 Example of MIPS-like code for loop unrolling and instruction fusion. a. Original loop. b. Loop after unrolling it twice. c. The unrolled loop of part b after pair-wise instruction fusion 17

3.2 Pipeline stage and Icache usage for a MIPS-like multiscalar processor with two complete pipelines or a dual-core processor. a. Normal execution mode. b. Fused execution mode 21

3.3 Dynamic power/energy reduction 24

3.4 Performance loss in the fused mode for various vector lengths. 25

3.5 Dynamic power reduction in the fused mode for various vector lengths. 25

3.6 Projection for FPGA and ASIC energy savings. Fused mode. Various multiscalar degrees. 27

3.7 Numbers of instructions in Icache 28

4.1 VRF structure. 31

4.2 RMM and its TLT interface. 32

4.3 Data structures used to manage the VRF. 33

4.4 Duration of fragmented registers for VL=32 and 64. 36

4.5 Thread request and release functions runtimes under various VLs and number of registers. 37

5.1 Multicore architecture for VP sharing (Instr Arb: vector instruction arbitrator). .. 40

5.2 Detailed architecture of the four-lane VP (FP: Floating-point). 43

5.3 Pipeline structure in the LDST and ALU data paths. 44

5.4 Vector lane architecture. 48

6.1 Macros to define vector instructions. 52

6.2 Utilization of the LDST or ALU units for various benchmarks, VLs, and number of simultaneous threads. 57

7.1 Software flowchart of the throughput-driven scheduler. 60

LIST OF FIGURES
(Continued)

Figure **Page**

7.2 Average execution time per schedule for pending thread queues of length; (a) 8, (b) 16. 62

7.3 The average of the total execution time for all threads scheduled in a time slice, with and without VP sharing, for λ= 0.5, 0.75 and 1. (Time slice: 10ms.)..... 64

8.1 Average total dynamic energy consumption per time slice for λ=0.5, 0.75 and 1. 70

8.2 Total energy consumption with (w/) and without (w/o) VP sharing, and with power gating, for λ=0.5, 0.75 and 1. 71

9.1 Mapping of VL, host-to-VM address and VP-to-VM address via virtualization. . 74

9.2 System architecture of a fusion capable VP of degree two. 77

9.3 Fusion of two DCT operations. 79

9.4 Dynamic power vs. utilization for both ALU and LDST data paths. 84

9.5 Optimal utilization boundaries for a. minimum energy b. minimum energy-execution time product. 88

9.6 Comparison of the E_{min}, ET_{min} policies against a VP w/o fusion and lane configuration over the average of 1000 time slices. a. energy b. runtime c. energy-runtime product. 90

10.1 Execution order of elements within an array of VL=16. 93

10.2 The ring-based structure of the data shuffle network. 94

10.3 Execution order and element destinations of a 4*4 matrix transpose. a. the original order b. the RLT setup c. the new order after RLT order translation..... 97

11.1 An example of multiple SMT VPs shared across processor groups. 103

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit
AC	Application Core
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CTS	Coarse-grain Temporal Sharing
DCT	Discrete Cosine Transform
DLP	Data Level Parallelism
DMA	Direct Memory Access
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FTS	Fine-grain Temporal Sharing
GPU	Graphical Processing Unit
HDU	Hazard Detection Unit
LDST	Load Store
LM	Local Memory
LMB	Local Memory Bus
MIMD	Multiple Instruction Multiple Data
MM	Matrix Multiplication
RLT	Reorder Lookup Table
RMM	Register Management Module
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multithreading
SPS	Scalar Processor Subsystem
TLT	Translation Lookup Table
VC	Vector Controller
VDP	Vector Dot Product
VL	Vector Length

VLS	Vector Lane Sharing
VM	Vector Memory
VP	Vector Processor
VRF	Vector Register File
VT	Vector Thread
WB	Write Back

CHAPTER 1

INTRODUCTION

This dissertation consists of three major parts: the instruction fusion technique, the vector processor (VP) virtualization technique, and the method to combine the two techniques. In this chapter, the two core technologies proposed by this dissertation are briefly introduced. The problems to be solved are defined first, and then the solutions to the problems are briefly introduced. At the end of this chapter, potential benefits of combining the two techniques are discussed.

1.1 The Utilization Wall

Increased requirements for energy consumption, memory size and system bandwidth have always been major concerns in the design of System-on-Chip (SoC) embedded systems. The situation became really insurmountable around 2005 when we entered the era of post-Dennardian scaling [Taylor, 2012]. The utilization wall [Goulding-Hotta et al., 2012], also known as the dark silicon problem, states the following: "With a constant power budget, the percentage of transistors on a chip that can switch at full frequency decreases exponentially with each process generation." This implies that an increasing portion of on-chip transistors will have to stay idle or even be power gated (i.e., to get in the sleep mode) with each new generation, thus giving rise to the name "dark silicon".

The utilization wall is caused by the breakdown of Dennard's MOSFET scaling law that was proposed in 1974 [Dennard et al., 1974]. When the CMOS process scales by a factor of k in the Dennard regime, it increases by k^3 the computational capabilities of a chip of fixed size due to increased operating frequency (f) and transistor density. On the other

hand, the accompanied decreases in the capacitance (*C*) and supply voltage (*Vdd*) lead to a decrease of k^3 in the energy consumption of each computation, therefore keeping the total power consumption almost constant.

Unfortunately, Dennard scaling has stopped since 2005 due to the breakdown of *Vdd* scaling caused by current leakage associated with threshold voltage scaling [Goulding-Hotta et al., 2012]. Without the downward scaling of *Vdd* in this post-Dennardian CMOS era, the performance of a fixed size chip still grows as k^3 in theory but the energy per computation only drops by *k*, thus limiting the on-chip resource usage when power consumption is constrained.

Experiments as well show that CMOS may have already hit the utilization wall; with a 45nm TSMC process, only 7% of a 300mm^2 die can switch at full frequency to stay under an 80W power budget [Venkatesh et al., 2010]. As CMOS technology continues to scale in the post-Dennardian era, this problem's impact worsens exponentially. This challenge requires microprocessor architects to focus on energy efficiency while achieving required performance levels. In fact, improved energy efficiency may facilitate higher performance by enabling more transistors to switch.

1.2 Inefficient VP Usage

Single instruction multiple data (SIMD) architectures are highly efficient in exploiting data level parallelism (DLP) in applications due to their specialization. A VP, also known as array processor, employs an SIMD architecture capable of processing an array of data elements simultaneously by executing a single vector instruction. As an accelerator, a VP can offload the DLP workload from general-purpose processors, thus enhancing the overall performance and energy efficiency. The VIRAM's multi-lane architecture is the basis of

several VP designs [Kozyrakis and Patterson, 2003]. VIRAM has separate pipeline structures for load-store (LDST) units and arithmetic logic units (ALUs). Vector registers are distributed evenly across the vector lanes. Each lane carries out ALU array operations on data within its local VRF. Vector elements in a lane are processed sequentially due to the ALU's pipelined architecture while all lanes work in parallel on different array parts. SODA [Lin et al., 2006] is a fully programmable VP that realizes the W-CDMA and IEEE802.11a protocols. [Lee et al., 2013] compared accelerators having MIMD (Multiple-Instruction Multiple-Data), vector SIMD and vector thread (VT) architectures in reference to programmability and implementation efficiency; it confirmed that SIMD vector architectures exploit DLP more efficiently than MIMD even for irregular data pattern accesses.

Multicores with embedded VPs have been implemented on FPGAs. They are often referred to as soft vector processors (SVPs). [Cho et al., 2006] introduced an SVP that eliminates memory bank conflicts by using address generation and rearrangement units in the vector memory. VESPA [Yiannacouras et al., 2008] allows the addition of vector lanes with minimum hardware modifications. Benchmarking shows a speedup of 6.3 under saturation with 16 lanes compared to a single lane. VIPERS achieves a speedup of 44 compared to the Nios II scalar processor [Yu et al., 2008]. Its number of functional units and register file bandwidth are configured by software; it occupies 25 times more area compared to its host scalar core. VEGAS [Chou et al., 2011] improves VIPERS's area-delay product. Using scratchpad memory instead of a VRF, it achieves a speedup of up to 208 compared to Nios II. Further improvements to reduce ALU bottlenecks produced VENICE [Severance and Lemieux, 2012] that doubles the performance-per-logic block

compared to VEGAS. With the integration of a streaming pipeline in the data path, a speedup of 7000 over a scalar processor was reported for the N-body problem [Severance et al., 2014].

Application specific VPs are optimized for certain applications. Multimedia applications containing video processing kernels deal with massive DLP. SIMD vector architectures are the best candidates to exploit the parallelism in video frames. Many researchers have tried to optimize codecs for the implementation of new video coding standards such as H.264 or MPEG4. [Iranpour and Kuchcinski, 2004; Lee et al., 2004; Kim et al., 2005; Shengfa et al., 2006; Lee et al., 2008] propose SIMD-based video codecs. A major challenge in video applications is irregular data accesses for video compression. [Lo et al., 2011] overcome this issue by inserting a crossbar between the ALUs and the VRF. [Yang et al., 2005] propose an application-specific VP prototyped on an FPGA for sparse matrix multiplication. IBM's PowerEN processor integrates five hardwired application-specific accelerators in a heterogeneous architecture for key functions such as compression, encryption, authentication, intrusion detection and XML processing. This approach facilitates energy-proportional performance scaling [Heil et al., 2014].

Unfortunately, single-thread dedicated VPs are often not efficiently utilized for the following reasons. First, every application contains some serial code for flow control or other system management, thus vector instructions may not be issued at a rate sufficient to keep the VP highly utilized. Second, data dependencies within some applications' vector instruction flows can cause frequent stalls, wasting precious clock cycles in the VP's super-pipelined floating-point units (FPUs). Finally, it may be preferable that applications with small vectorizable code be executed on the scalar host in order to give another

highly-vectorized application exclusive VP access. However, the former applications as well could benefit from using simultaneously the VP. Benchmarking shows that applications with VP utilization as low as 8.5% can yield a speedup of 84 by executing on a VP compared to a scalar processor with the same clock frequency: an 8*8 Matrix Multiplication performed by a MicroBlaze (MB) scalar processor takes 20.5ms to complete, while the VP used in this dissertation only requires 241μs as shown in Section 6.2. Given the 100MHz clock rate of its implementation, the 4-lane VP's average ALU utilization is calculated to be 8.5%.

The issue of inefficient usage worsens as VPs scale in the number of lanes: Traditional VPs designed to service exclusively one host scalar processor are normally optimized for applications of a certain level of DLP, and scale easily so that more vector lanes can be added to exploit other applications of higher DLP [Kozyrakis and Patterson, 2003; Yiannacouras et al., 2008; Yu et al., 2008]. However, an increased number of lanes will further reduce the already low VP utilization for the lower-DLP applications.

1.3 Motivation and Objectives

To alleviate the impact of the utilization wall as mentioned in Section 1.1, an instruction fusion technique is proposed in this dissertation. The technique can be applied to multiscalar and many-core processors to fuse similar instructions within vector code, and therefore reduce processor dynamic power by idling early processor stages. When running in the proposed fused mode, the multicore processor only fetches one instruction per clock cycle while executing multiple copies of the fetched instruction, thus saving energy in the fused early pipeline stages as well as in the instruction cache.

To address the challenges imposed by inefficient VP utilization as mentioned in Section 1.2, virtualization for VP sharing with simultaneous multithreading (SMT) is introduced in this dissertation. The SMT approach is similar to Intel's Hyper-Threading Technology (HTT) for general-purpose processors that "makes a single physical processor appear as multiple logic processors" [Marr et al., 2002]. However, in this dissertation, SMT is applied to vector code. This approach achieves high aggregate VP utilization independent of individual vector thread DLP rates. VP virtualization solves register name conflicts among threads using a novel VRF virtualization algorithm that can dynamically allocate physical registers of varying lengths to threads. With easy-to-use VRF management kernel functions, programmers are provided with a fixed register name space and VRF management becomes transparent. To prove its viability, VP virtualization is realized on a multi-lane VP [Rooholamin and Ziavras, 2015], and the VP is interfaced with a multicore processor system to benchmark its performance and energy consumption.

Related work will be discussed in Chapter 2. All the details of the proposed instruction fusion technique will be discussed in Chapter 3, followed by the discussion of the VP virtualization technology in Chapter 4 to Chapter 8. Chapter 3 covers the approach that applies fusion (Section 3.1), the architecture of the FPGA prototype (Section 3.2), and the simulation results of eight loop intensive benchmarks (Section 3.3). Topics related to the VP virtualization technique will be covered in Chapter 4 to Chapter 8. Details of the innovative VP virtualization technique are covered in Chapter 4. The architecture and FPGA implementation of the prototype system, including the host subsystem (that consists of five scalar processors) and the VP subsystem, are described in Chapter 5. The benchmarks and VP performance of homogeneous multithreading are discussed in Chapter

6. The proposed throughput-driven scheduling algorithm designed for both static and dynamic heterogeneous multithreading is covered in detail in Chapter 7. Power analysis of the VP is given in Chapter 8.

An improved VP prototype that incorporates both the virtualization and the instruction fusion techniques is presented in Chapter 9. The improved prototype supports scheduler-triggered thread level vector instruction fusion. Due to the complete virtualization of the VP, fused threads can be easily interpreted by the hardware as multiple independent threads working in separate virtual spaces. The new prototype also supports dynamic VP lane power gating. To optimize VP efficiency for various applications, an accurate VP power model is derived based on the VP's ALU and LDST utilization. Two optimization policies are proposed to achieve minimum energy consumption or minimum execution time and energy product. The optimal number of active VP lanes can be dynamically chosen by the scheduler based on the optimization policy and applications' ALU and LDST utilization.

All the VP prototypes introduced in this work incorporate a private memory architecture where each VP lane has its dedicated memory bank and cannot access data elements in other banks. Inter-lane data exchanges are performed by the host scalar processor and this approach adversely affects performance. To overcome this issue, a high throughput pipelined data shuffle network with an innovative reordering algorithm is proposed. The network allows high throughput pipelined inter-lane data shuffle and is presented in Chapter 10.

Part of this work that primarily relates to VP virtualization is a collaborative research project with another CAPPL (Computer Architecture & Parallel Processing

Laboratory) PhD student, Seyedamin Rooholamin. The main contributions of the author include the introduction of the basic instruction fusion technique, the design of the multicore host architecture for evaluating the SMT VP, the invention of the VRF virtualization technique and the throughput-driven scheduler, the proposal of the improved VP prototype which supports thread fusion and dynamic lane configuration, and the invention of the data shuffle network.

CHAPTER 2

RELATED WORK

In this chapter, related works are compared against the techniques proposed in this dissertation. Instruction fusion related work is covered in Section 2.1, and work related to VP virtualization and SMT is covered in Section 2.2.

2.1 Work Related to Instruction Fusion

Thread fusion [Rakvic et al., 2010] dynamically fuses (i.e., combines) instructions by exploiting any parallelism between two parallel threads in simultaneous multi-threaded (SMT) processors. When a synchronous point is reached by the two threads at run time, similar instructions from the two threads are merged to save energy in the front end of the pipeline. Dynamic vectorization [Pajuelo et al., 2002] attempts to exploit SIMD parallelism in superscalar processors by using historical information to predict if certain scalar operations are likely to be repeated and may be, therefore, vectorizable. Once predicted to be vectorizable, multiple instances of these scalar instructions will be combined for simultaneous execution in the vector mode, which has some similarities with the fused mode execution technique introduced later in this dissertation. VPs [Beldianu and Ziavras, 2015] employ specialized hardware to execute vector instructions which are identified at static time either by programmers or optimizing compilers. Better energy efficiency and performance are achieved due to hardware customization.

The proposed instruction fusion technique looks at assembly-language code produced by loop unrolling to identify similar RISC instructions that can be fused at static time for simultaneous SIMD execution at run time. Loops involving vector code are very

common due to existing applications (e.g., matrix operations, solving dense systems of linear equations [Wang et al., 2013], power flow analysis [Wang et al., 2007]) and emerging data streaming applications (e.g., MPEG and JPEG encoding, cryptography [Ansari and Hasan, 2008]) that often require high-performance implementations with reduced energy budgets. Compared to the earlier work on dynamic techniques mentioned above, which incorporate dedicated hardware that constantly consumes energy throughout the entire code execution in order to make dynamic decisions, the proposed static approach is more energy efficient. It consumes extra energy only when the processor switches execution mode.

The instruction fusion technique enables SIMD execution with minimum modifications to multiscalar or many-core processors. It actually transforms the target hardware into a novel hybrid of a VP and a general-purpose processor. Compared to a standard VP, the proposed processor is capable of executing branch instructions even in the SIMD mode. Plus, with the employment of suitable compiler techniques, SIMD execution using the instruction fusion technique can support memory access strides that are non-constant, thus competing with advanced VPs. Since the proposed design also consumes a drastically reduced number of logic resources compared to VPs with controlling general-purpose processors, it is a better candidate for FPGA-based realizations (FPGAs have constrained resource counts).

In relation to the work in [Rakvic et al., 2010] and [Pajuelo et al., 2002], the main difference is that the proposed approach makes decisions about instruction fusion at static time instead of run time; run time approaches are more difficult to implement and require additional logic to constantly monitor program execution, thus they consume extra energy

even when fusion is not applied. The proposed approach, on the other hand, reduces power consumption in early pipeline stages when the processors are executing loop code in the fused mode. The scheme is designed specifically to optimize the execution of loop code in the fused mode without introducing any time or energy overheads for unfused mode execution. One further advantage of static fusion is that cache utilization is reduced by 50% for fused code, thus making more cache space available for code that results in increased cache hit rates and improved performance.

2.2 Work Related to VP Virtualization

Multicores with embedded VPs do not normally support sharing [Yang and Ziavras, 2005; Yiannacouras et al., 2008; Yu et al., 2008; Chou et al., 2011; Severance and Lemieux, 2012]. VP sharing for multiple threads or cores was first proposed in [Beldianu and Ziavras, 2013]. Three sharing policies were introduced for a multi-lane VP, namely coarse-grain temporal sharing (CTS), fine-grain temporal sharing (FTS) and vector lane sharing (VLS). Under CTS, a core reserves the entire VP exclusively until its current vector thread stalls or completes execution, and then hands over VP access to another core. FTS provides finer-grain time sharing of the VP, under which vector instructions from different threads compete for per clock cycle VP access. CTS and FTS support sharing for threads of similar VL (vector length that represents the number of elements in a vector). VLS is the only mode allowing threads of different VLs to coexist in the VP which is split into distinct sets of vector lanes, one set per core; VLS uses multiple vector controllers (VCs) to control the sets. FTS achieves the best VP utilization and may double the speedup compared to CTS while reducing the dynamic energy by 50% for a dual core [Beldianu and Ziavras, 2015].

The VP sharing proposed in this dissertation is similar to FTS where all cores access the entire VP concurrently and resource conflicts are resolved by a single arbitrator.

To improve the performance of the VP introduced in [Beldianu and Ziavras, 2013], [Rooholamin and Ziavras, 2015] introduced a multi-lane VP with separate pipelines for the ALU and LDST units. A lane's LDST unit exclusively accesses a port of a distinct vector memory (VM) bank, thus eliminating contentions that can lead to delays or stalls for sequential read/write operations. All non-sequential memory accesses (e.g., data shuffling and index addressing) are handled by per-lane dedicated shuffle engines that utilize a second port of each VM bank. This VP is highly flexible for applications with varying VL, thus allowing the VL value to be specified by each individual vector instruction; the instruction decoder in each lane is then in charge of vector instruction synchronization. Threads of disparate VLs running on the same scalar processor can exploit the VP in a CTS-like fashion as long as they do not result in vector register name conflicts. Benchmarking showed speedups of up to 1500 compared to running vector code on a scalar processor with the same clock frequency.

In this dissertation, the proposed VP virtualization technique was applied to a multithreaded VP similar to that of [Rooholamin and Ziavras, 2015] with minor hardware modifications. Whereas the latter prototype employed two cores without SMT, the new VP prototype interfaces five cores (without loss of generality), supports SMT and power gating, and carries out high throughput runtime scheduling of vector threads. Four cores can share the VP simultaneously while running vector codes of different VLs. The fifth core does VP management and vector thread scheduling. Any vector register name conflicts between threads are resolved via an innovative VRF virtualization technique.

Virtualization involves an effective register management algorithm run on the control core and a hardwired translation look-up table (TLT) for fast virtual-to-physical register name (i.e., ID) translation. With VRF virtualization, the management of physical vector register names becomes transparent to application programmers who assume a virtual register space.

The proposed VP sharing also differs from [Beldianu and Ziavras, 2013] in four major aspects. Most importantly, in this dissertation, VRF virtualization is introduced for VP sharing to improve FTS. Second, their work does not support FTS for threads of different VLs. In contrast, the proposed sharing technique maximized VP utilization by allowing multiple threads of different VL to run simultaneously that yields substantial throughput increases. Third, contrary to their work where all cores directly interfaced the VP, in this dissertation a distinct FIFO is added between the VP and each core to eliminate frequent core stalls due to vector instruction arbitration. Under low VP utilization, an application's speed is bounded by its host core. The distinct FIFOs allow a core to keep sending vector instructions until its FIFO becomes full. Finally, in this implementation the crossbar between the vector lanes and VP memory banks (VM) is removed by connecting a bank's dedicated port to the attached lane's LDST unit. This modification eliminates arbitrator delays in the crossbar and improves VP throughput for sequential memory accesses that are omnipresent due to VP pipelined units that target array operations. Inter-lane data exchange is supported by the scalar cores, which have access to all VM banks in a low-order interleaved fashion. Removing the crossbar also improves VP scalability. With both the VM and VRF distributed across the VP lanes, scalability is achieved since the individual lane complexity is independent of the number of lanes.

Although a general-purpose GPU (GPGPU) can run hundreds of vector threads simultaneously using streaming multiprocessors (SMs), all threads must be homogeneous and invoked by the same host. In contrast, the proposed virtualized VP can execute simultaneously heterogeneous host threads. VPs also consume drastically reduced resources and energy compared to GPGPUs [Beldianu and Ziavras, 2015]; e.g., Nvidia's Maxwell GPU GTX 980 consists of 16 SMs, each having 128 CUDA cores, and 5.2 billion transistors [Nvidia Corp., 2014]. Without highly sustained DLP and a much needed fine-grain power management mechanism, a lot of CUDA cores in each SM may be frequently idle while consuming prohibitively high static energy.

CHAPTER 3

IMPLEMENTATION AND EVALUATION OF INSTRUCTION FUSION

In this chapter, the details of the instruction fusion technique are discussed. The first part (Section 3.1) explains how instruction similarity can be utilized to facilitate instruction fusion. A processor with a special execution mode known as the fused mode is prototyped to evaluate the benefit of the fusion technique. The processor architecture and its FPGA implementation details are covered in the second part (Section 3.2). The third part (Section 3.3) covers the benchmarking and simulation results of the prototyped processor.

3.1 How Instruction Fusion Works

3.1.1 Instruction Similarity

After a loop is unrolled twice, there exist two very similar instruction blocks in the code. Without any further code manipulation, pairs of respective instructions from the two blocks differ only in their use of operand registers, but the patterns of data flow are identical in both blocks. Such two instruction blocks are said to be fusible if there are not any interdependencies between them. Sometimes one or more sub-blocks within the blocks, or individual instruction pairs, may be fusible depending on existing interdependencies.

Figure 3.1 gives a very simple example of a loop that is unrolled to generate two fusible instruction blocks. The loop in Figure 3.1a simply adds two integer arrays, ***a*** and ***b***, of length 100 each to generate a third integer array ***c***. In the unrolled loop of Figure 3.1b, respective pairs from instruction sequences 1-7 and 8-14 can be fused, which means that these 14 instructions can be represented using only seven instructions. This example can be

easily extrapolated for loops unrolled more than twice. Such fused code, when executed on a multiscalar processor, enables the pipeline to fetch and decode only one instruction each time and then issue it to multiple ALUs, therefore saving energy in the instruction fetch and decode stages of the disabled units as well as in their respective instruction caches. In addition, instruction cache space requirements can be reduced by avoiding code replication for such loop iterations. Of course, it is assumed that a register renaming technique for loop unrolling is applied at static time and the register file is modular to allow different ALUs to simultaneously operate on different register banks

For the sake of simplicity and without loss of generality, it is demonstrated in this dissertation the case where the processor can fetch and execute two instructions in each clock cycle. This concept can be easily extended to a multiscalar processor that can execute four or even eight instructions per clock cycle, or to a many-core processor of comparable hardware complexity.

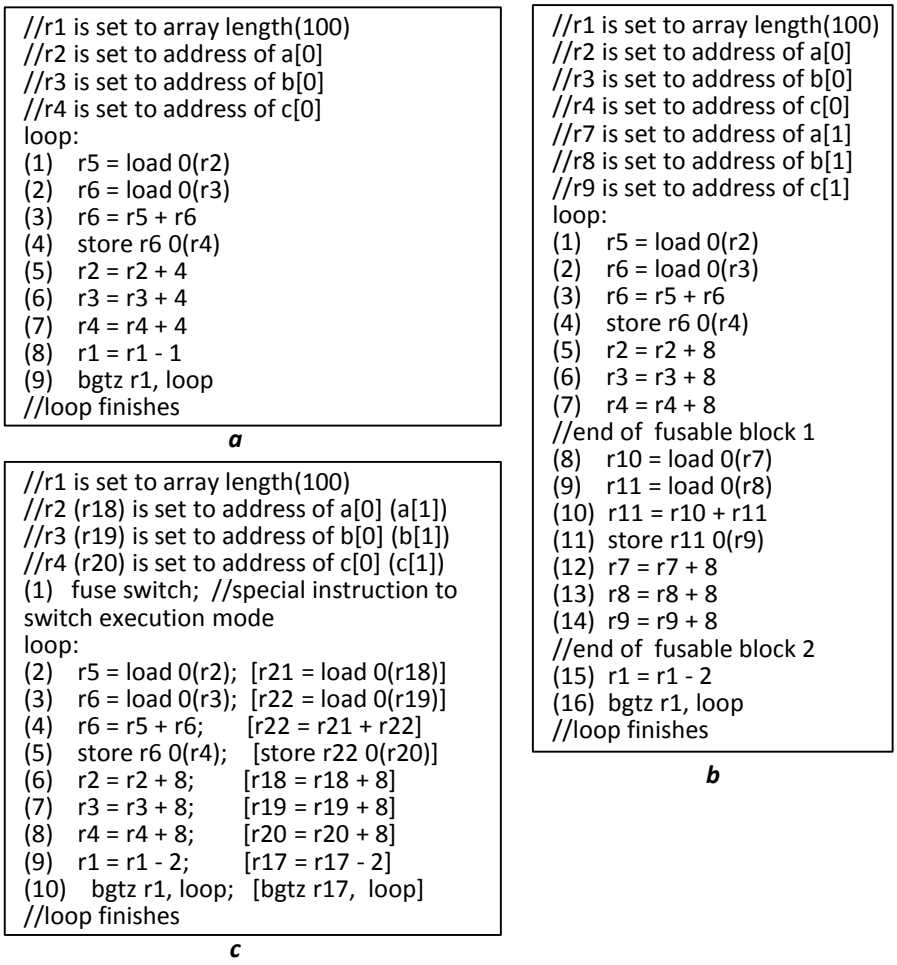


Figure 3.1 Example of MIPS-like code for loop unrolling and instruction fusion. **a.** Original loop. **b.** Loop after unrolling it twice. **c.** The unrolled loop of part b after pair-wise instruction fusion.

3.1.2 Instruction Fusion

When executing a fused instruction that represents a pair of similar machine-language instructions, the ALU registers employed by the two original instructions must be the ones used in the actual operations. To achieve this without extending the length of the fused instructions, a simple method of register renaming was introduced, along with a special execution mode for the processor, called the fused mode. In the fused mode, the processor will fetch only one instruction per clock cycle, then decode it and do the register renaming

to generate two slightly different copies of control signals for the two pipelines running in parallel. These two copies of signals will be treated by the execution units as two regularly decoded instructions. An extra one-bit signal, called the fuse state, is used to represent the pipelines' execution mode; '0' is used for normal execution and '1' for the fused mode. The switching between the normal and fused modes is implemented by a compiler-inserted special instruction called "fuse switch".

Register renaming in the fused mode is done in the way shown in Figure 3.1c, which illustrates how the doubly unrolled loop in Figure 3.1b can be modified to run in the fused mode. In Figure 3.1b, r7 is preset to hold the address of the second element in array a, and its role in block 2 is identical to r2's role in block 1 with the only difference being that they access consecutive odd- and even-addressed elements, respectively, of array a. This represents the output of the compiler. After fusing this pair of instructions, the processor will only fetch the instruction which operates on r2. To produce the second copy of signals that operate on r7, hardwire-driven runtime register renaming is needed.

For the purpose of simple hardware implementation, register r18 is used instead of r7 when applying the fusion technique, as shown in Figure 3.1c, and the reason follows. Assuming a processor with 32 general-purpose registers, the addresses of r18 and r2 only differ in the most significant bit, so producing at run time r18's ID from that of r2 can be easily done by changing to 1 the most significant bit of the operating register's ID. Applying this register renaming scheme to all the registers in the fused mode, the processor can recover instruction block 2 from block 1. The assumption is that the compiler does not assign the upper register bank for other programming purposes, or code is embedded to move around register values. As shown in Figure 3.1c, registers are configured to hold the

desired values before the “fuse switch” instruction is executed. After “fuse switch” has been executed, the processor enters the fused mode where it begins to fetch one instruction per clock cycle which is followed by the execution of two copies of it, one copy being the fetched instruction itself and the other being the register-renamed copy.

In Figure 3.1c, the fused code only contains nine instructions excluding the “fuse switch” instruction. However, there are actually 18 executable instructions in the code because all the register-renamed instructions displayed in brackets will be executed in parallel with the ones outside the brackets. It is worth mentioning that there are 16 instructions in the unrolled loop under the normal mode, but only 14 of them are fusible. After the instruction fusion, the 14 fusible instructions are merged into seven, but the two loop control instructions remain unchanged, resulting in a total of nine instructions (18 executable instructions) under the fused mode. The fused mode yields two obvious disadvantages: the first one is that the number of registers available to the programmer is effectively reduced by half. The second problem is that the loop control instructions, such as instructions nine and ten in Figure 3.1c, have to be executed twice, and one of the results is redundant.

3.2 Processor Implementation for Fused Mode Execution

3.2.1 MIPS-I Like Multiscalar Architecture

The processor architecture was prototyped using two copies of an in-order scalar processor based on the MIPS-I instruction set. The two scalars share a common register file and are merged to form a very simple multiscalar processor, on which the proposed fusion

technique is applied. The processor contains two pipelines, namely pipeline 0 and pipeline 1, each with the classic five stages known as per the following:

- Fetch: Instructions are fetched from the instruction cache and stored in a buffer, waiting to be sent to the next stage. Fetch unit 1 will be put to idle when the processor is executing instructions in the fused mode.

- Decode: Instructions are decoded and operands are read from the register file or the bypass unit. In fused mode, decode unit 1 is in the idle state; decode unit 0 will decode the instruction, read the operands for both the original instruction and the register-renamed instruction, and finally send them to the two units in the next pipeline stage.

- Execution: Decoded instructions are executed. Starting from the execution stage, the pipelines are always in the on state and no longer affected by the fused state.

- Memory: Memory accesses to the data cache are performed in this stage.

- Write Back: ALU results and data from memory are written to the register files.

Figure 3.2 illustrates pipeline stage usage for normal and fused execution. In Figure 3.2a, all functional units are working while fetching in parallel two instructions from the instruction caches in each clock cycle. In Figure 3.2b, the processor runs in the fused mode so everything before the execution stage is idling in pipeline 1. Only one instruction fetch and decode is performed per clock cycle but actually two instructions are then executed.

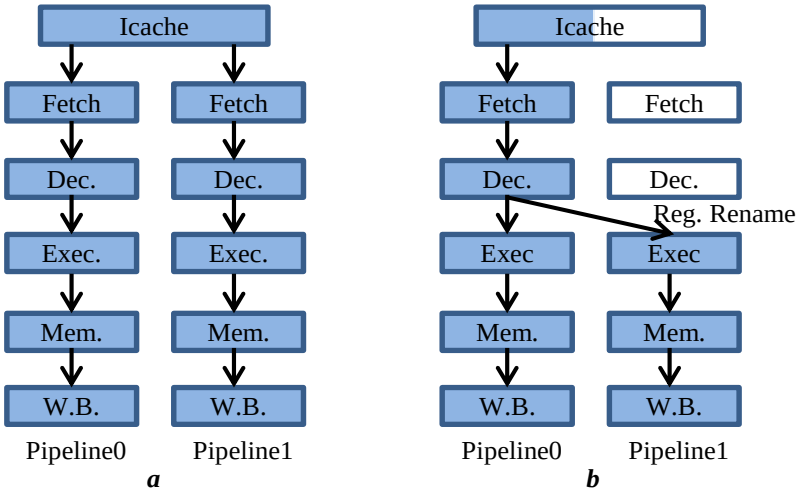


Figure 3.2 Pipeline stage and Icache usage for a MIPS-like multiscalar processor with two complete pipelines or a dual-core processor. **a.** Normal execution mode. **b.** Fused execution mode.

Many-core or massively multi-core processors of the future will contain hundreds of basic cores. Within a quad-core or octa-core cluster of such a many-core processor, the proposed instruction fusion technique will be applied using a similar to the presented dual-core approach with simple modifications to the extensions shown in Figure 3.2b. The various cores will execute their own instructions independently before they reach a synchronization point designating the start of fused-mode execution for the involved cluster cores. Once synchronous execution begins, the core in charge of loop control will be fetching and decoding sequences of fused instructions and then broadcasting control signals for the fused instructions to all other cores. All the cores will be executing fused instructions synchronously until the end of the fused loop is reached. For a multi-core implementation, register renaming can be omitted since each core is expected to own an independent register space. However, future many-core processors may support the fusion of register files across involved cores, thus implying the need for register renaming.

3.2.2 FPGA Implementation

The design was implemented on a Xilinx Virtex-7 xc7vx330t FPGA device, which is built with the 28nm process technology. The multiscalar processor is written in VHDL, and two 16KB block memories generated using the Xilinx Core Generator are used as the instruction and data caches. The system is fully synthesized, translated and routed to operate at 100MHz. The entire design flow is performed using the Xilinx ISE Design Suite 14.7. Due to the low clock frequency of the processor, which is expected for FPGA realizations, all memory accesses can be completed within one clock cycle. Therefore, the performance numbers given in Section 3.3 reflect the situation where all instruction and data accesses result in a cache hit rate of 100%. This assumption is reasonable since when executing loop code the cache is kept hot.

3.3 Benchmarking for Instruction Fusion

3.3.1 Evaluation Procedure

Since the proposed instruction fusion technique aims to optimize the power consumption of loop code, eight loop-intensive benchmark applications were created to evaluate the proposed processor design that supports the execution of instruction pairs in the fused mode. They are: 8x8 matrix multiplication (MM8), 16-point discrete Fourier transform (DFT16), 32-tap finite impulse response filter (FIR32), 32x32 matrix transpose (MT32), RGB to YIQ conversion (RYC), vector dot product with a vector length of 32 (VDP32), 2D discrete cosine transform (DCT) shuffling with a vector length of 64 (DCT64), and fast Fourier transform (FFT) reordering with a vector length of 128 (FRO128). For the sake of simplicity, only the transformation of the real-number component was performed in the

implementation of DFT16 since operations on the imaginary component are almost identical; they involve a mere change of the coefficients.

All eight applications are executed with loops unrolled twice, assuming identical input data sets under both the normal and fused modes. Without loss of generality, all assembly-language codes were hand written and manually optimized. To obtain reliable results, application execution is simulated in the Xilinx ISE Simulator (ISim) using the fully placed and routed model. A Switching Activity Interchange Format (SAIF) file is obtained during simulation to record all the switching activities of signals in the design. The SAIF file is then used by the Xilinx XPower analyzer to calculate the precise power consumption under each execution scenario.

3.3.2 Analysis of Results

The detailed performance and dynamic power measurements are shown in Table 3.1. As expected, all the scenarios have the same leakage power, 180mW, which is determined by the FPGA model. For MM8, DFT16, FIR32, MT32 and VDP32, the execution times are measured based on the average time needed to produce each element in the output. For RYC, the execution time per pixel is given. For DCT64 and FRO128, the execution time for an entire output vector is measured. The dynamic power and energy reduction of each application under the fused mode are calculated and shown in Figure 3.3. According to the figure, the dynamic power of applications under the fused mode is reduced by up to 11.9% compared to the normal mode. Although under the fused mode there is a performance loss of less than 1% for most applications, it is very minor compared to the power reduction, and the energy for each application is still reduced by more than 6%.

Table 3.1 Performance/Power/Energy Benchmarking (Normal and Fused Execution)

Scenario	Fused			Normal		
	Power (mW)	***Time (ns)***	***Energy (nJ)***	***Power (mW)***	***Time (ns)***	***Energy (nJ***
MM8	272	720.9	196.1	303	715.3	216.7
DFT16	294	1336.3	392.9	320	1328.1	425.0
FIR32	316	1230.6	388.9	338	1225.0	414.0
MT32	271	61.0	16.5	289	60.8	17.6
RYC	302	332.5	100.4	343	333.8	114.5
VDP32	266	1389.4	369.6	294	1383.8	406.8
DCT64	285	2791.3	795.5	310	2775.0	860.3
FRO128	243	2311.3	561.6	263	2295.0	603.6

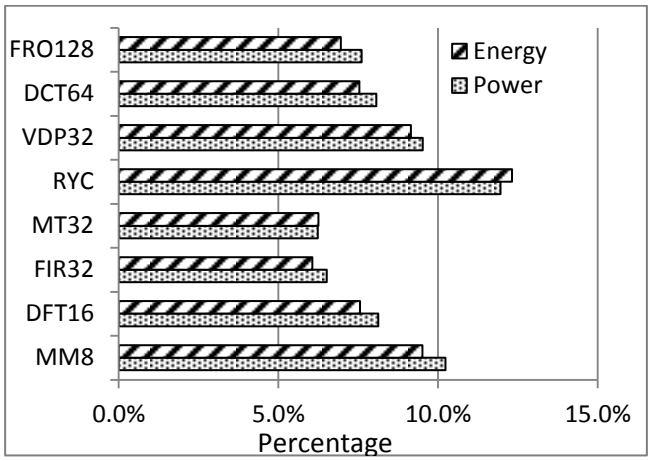


Figure 3.3 Dynamic power/energy reduction.

As mentioned at the end of Section 3.1.2, loop control instructions introduce limited redundant execution in the fused mode. Therefore, higher control instruction occurrence in the execution flow will lead to higher performance loss and less energy reduction under the fused mode. To study the impact of this effect as a function of the vector length, in depth experiments was performed on the vector dot product (VDP) and matrix transpose (MT) benchmarks with a vector length of 4, 8, 16 and 32. The performance loss and power reduction in the fused mode under various vector sizes are shown in Figure 3.4 and 3.5. It can be seen in Figure 3.4 that the performance loss increases with the decrease in the vector length. This agrees with the previous prediction since a

lower vector length causes the execution flow to execute loop control instructions more frequently. As shown in Figure 3.5, changes in the vector length do not affect the power reduction significantly; therefore, changes in the overall energy reduction in the fused mode, as a function of the vector length, will mostly depend on the performance differences.

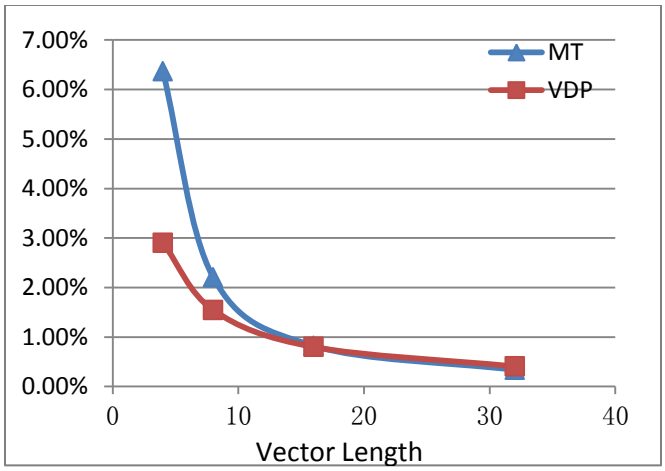


Figure 3.4 Performance loss in the fused mode for various vector lengths.

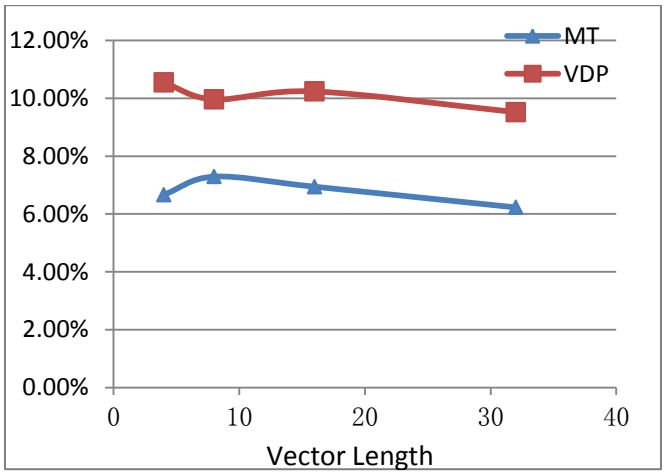


Figure 3.5 Dynamic power reduction in the fused mode for various vector lengths.

The proposed system was implemented on an FPGA platform for fast prototyping. However, compared to an application specific integrated circuit (ASIC) implementation, an FPGA sacrifices both performance and energy efficiency, and more importantly in this case, the percentage of energy that can be saved by using the fused mode. In this FPGA implementation, a block RAM (Xilinx BRAM) was used to emulate the function of the instruction cache, and the simulation showed that under the normal mode it only consumed approximately 20% of the total dynamic power of the processor. Even though the instruction cache dynamic power was reduced by nearly 40% under the fused mode in the dual-port cache, the benchmarks show that it contributed to less than 10% in total dynamic power savings. This is due to at least two facts: (a) the logic circuits needed to realize processor components on FPGAs are mainly implemented using LUT techniques; therefore, they consume much more power than their ASIC counterparts; (b) the proposed design uses a dual-port cache implemented in a BRAM but the savings can improve further with two separate caches that can reduce not only the dynamic but also the static power via power gating.

It is worth mentioning that L1 instruction caches in high speed processors consume a much larger portion of the total power compared to the one in this implementation. Instruction caches have the highest utilization among all the components in processors, and are extremely power hungry due to the requirement for high performance; for example, fetching each instruction requires accessing all N ways in the N-way set-associative cache and reading N tags. In an embedded processor, supplying instructions may dissipate up to 42% of the total dynamic energy [Dally et al., 2008]. With this assumption, one could expect that the energy saved (as a percentage of the total energy) by introducing the fused

instruction mode will be dramatically increased for an ASIC implementation of the proposed fused processor. The proposed technique's impact can become more dramatic by involving four or eight pipelines.

Given the average energy savings of 8% in the benchmarks and that the major energy savings in the fused mode are due to reduced memory accesses, it can be projected the total energy savings under the fused mode for a multiscalar processor. Furthermore, it can be safely assumed, based on the assumption discussed previously, that the percentage of energy savings under the fused mode will almost double for an ASIC implementation. Figure 3.6 shows the projected energy savings in the fused mode for multiscalars of various degrees realized with FPGA and ASIC technologies.

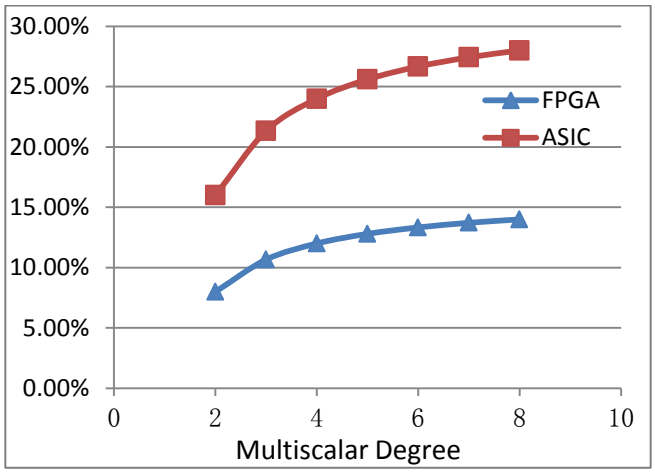


Figure 3.6 Projection for FPGA and ASIC energy savings. Fused mode. Various multiscalar degrees.

In addition to the reduction in energy, the instruction fusion technique also reduces the code size for unrolled loops. Figure 3.7 compares the number of instructions stored in the cache memory for each application under the two execution modes. The

assembly-language codes of an application for the fused and normal modes take about the same number of instructions to pre-configure the register file before actually executing the application. Once the pre-configuration is completed, the number of non-loop-control instructions in the fused mode is reduced by 50%. For the benchmarking in this chapter, where all the codes were manually optimized, the fused mode yields 17% to 45% in code size reduction, as shown in Figure 3.7. If the design was implemented in ASIC, the reduced code size will have further benefit other than reduced memory requirements: higher cache hit rates may be achieved due to the reduced code size under the fused mode, and, therefore, the fused mode will have both better performance and lower power compared to the normal mode.

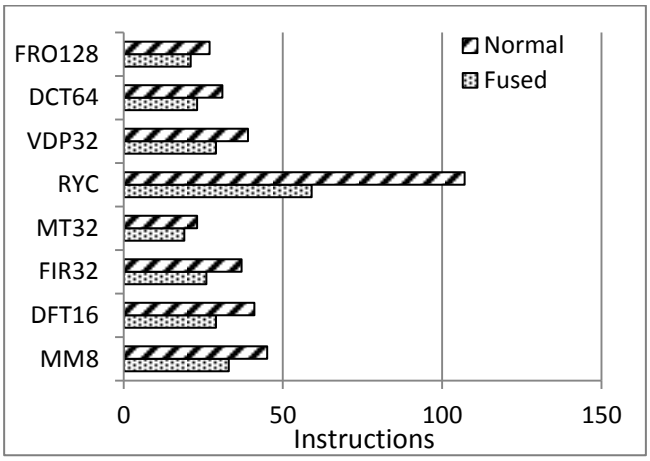


Figure 3.7 Numbers of instructions in Icache.

CHAPTER 4

VP VIRTUALIZATION

The VRF virtualization technique is the core to efficient VP multithreading. Without the dynamic VRF resource allocation support, it is hardly possible for programmers to decide in advance the vector register usage of various combinations of simultaneous threads. This chapter introduces the details of the scalable VRF virtualization technique and its application on the chosen target VP.

4.1 Virtual Register Name Space for Multithreading

The proposed prototype supports simultaneous VP sharing for four threads with a thread's VL being 16, 32 or 64. Virtualization resolves register conflicts among active threads using a software algorithm accelerated by minor hardware modifications. Each vector thread is programmed with its own virtual register name space that is mapped at runtime to physical VRF registers based on their availability. The virtualization technique involves two components: (1) a register management algorithm run by a scalar processor that determines virtual to physical vector register mappings; and (2) a hardwired TLT that facilitates the fast translation of IDs between virtual and physical registers after the former algorithm completes a mapping. TLT name translation uses one pipeline stage in the VP, and it is the only VP hardware modification needed. With the programming interface for the proposed prototype, applications have access to virtual vector registers 0-31 for VL=16 or 32, and 0-15 for VL=64.

4.2 The VRF Structure

The physical VRF consists of 16 vector registers where each register can store 64 (i.e., VL=64) 32-bit elements. If needed, each register of VL=64 can be split into two registers of VL=32, and each register of VL=32 can be further split into two registers of VL=16. The notation **reg_64(n-1)** is used to represent the n-th physical vector register for VL=64, where n=1, 2, …, 16. As illustrated in Figure 4.1, **reg_64(0)** can be split into **reg_32(0)** and **reg_32(1)**, or further to become **reg_16(0)**, **reg_16(1)**, **reg_16(2)** and **reg_16(3)**. The vector instruction decoder needs both a register's physical name and an instruction's VL to physically locate a register in VRF. In the proposed VP prototype, each instruction contains a 2-bit thread ID, the 5-bit IDs of involved virtual registers, and the instruction's VL encoded in a 2-bit field. The thread ID and virtual register IDs are used to obtain physical register IDs from TLT. The VRF can be easily expanded since it is distributed across multiple lanes. The proposed VRF management algorithm also scales well to manage any VRF with a power of two register size. More simultaneous threads can be supported by linearly expanding the number of entries in the TLT and the instruction arbitrator's state machine. It is only demonstrated in this chapter the case where up to four threads share simultaneously the VP, and the VRF has the structure of Figure 4.1

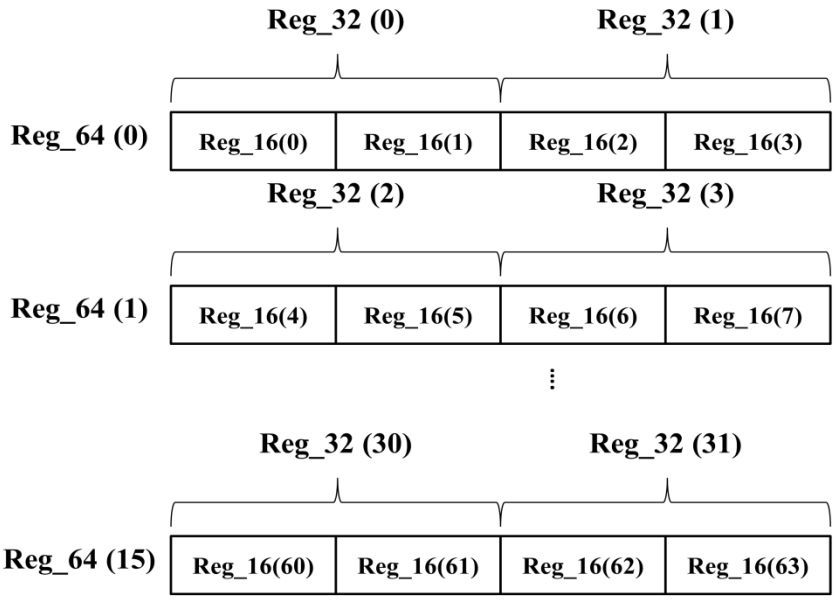


Figure 4.1 VRF structure.

4.3 The Vector Register Management Module (RMM) and Algorithm

The functional blocks of the RMM and its TLT interface are shown in Figure 4.2. The register management algorithm supports a virtual space of 32 vector registers for each thread. RMM receives as input a request to either allocate or release a number of registers of certain VL; for a release, it just receives the ID of a retiring vector thread since RMM maintains detailed lists of assigned resources. After processing the request and updating TLT, RMM assigns a vector thread ID to the new allocation and sends it to the requesting core. To minimize vector register fragmentation, the register access queues as well as the register split, allocation, release and merge/recovery mechanisms give priority to the preservation of registers with larger VL. For current benchmarking in this dissertation, the functionality of RMM is realized in software by MB0. A hardwired RMM is a future

objective towards even higher performance and lower energy consumption. The register management algorithm is written in C.

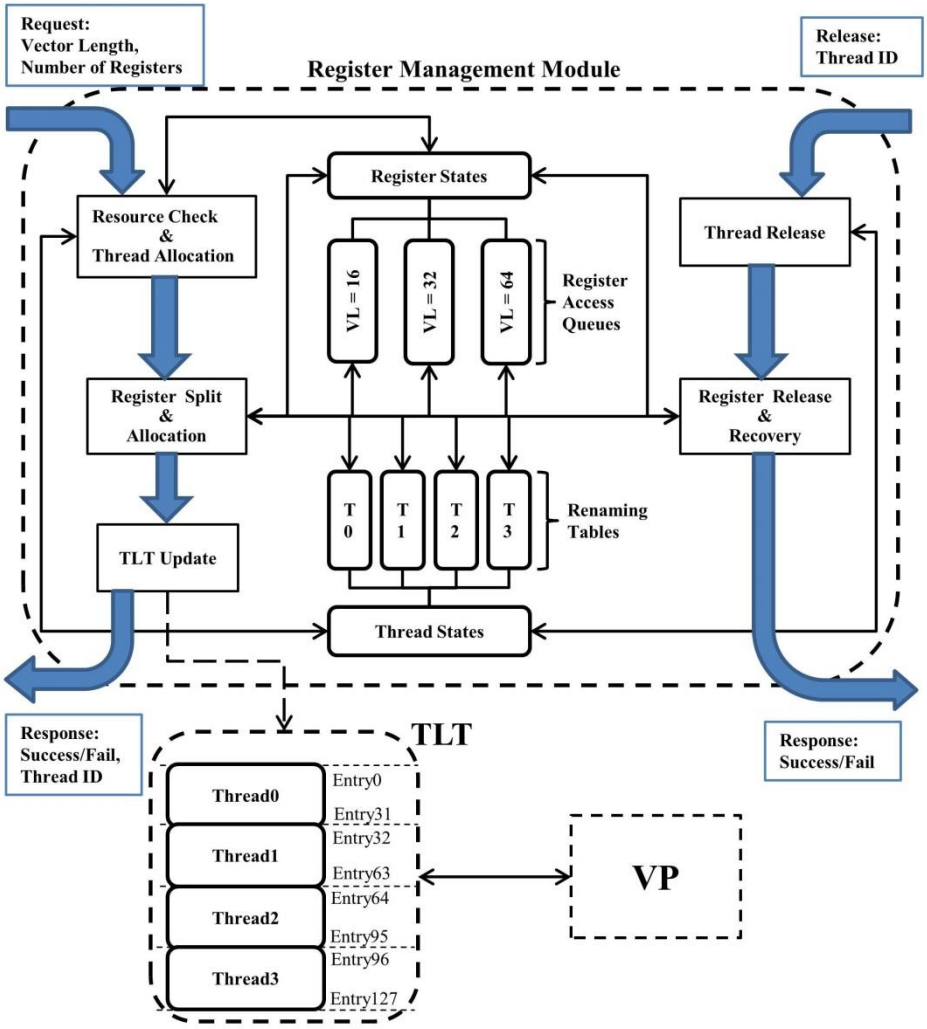


Figure 4.2 RMM and its TLT interface.

Figure 4.3 shows two data structures for VRF management. *Struct* **vp_control** contains data for VRF management. Each register is an instance of *struct* **vp_reg**; there are three **vp_reg** arrays in **vp_control** for VL=16, 32 and 64, respectively. A register's **vp_reg** record is located by using its physical ID as the index into one of the three arrays. If the

register is available, **vp_reg** can also be accessed using the quick access queue. Inside **vp_reg**, **rname** is the physical name of the register; it initializes to the index in the array. **in_queue** is set to '1' when a register is put into the fast access queue; it can become available to a thread or to be split for a smaller VL. After a register is assigned or split, **in_queue** is set to '0' and **used** is set to '1'. Fields **prev** and **next** are for the fast access queue (a doubly linked list). The latter is accessed to identify an available register for allocation or splitting. Using one of the **head_16**, **head_32** and **head_64** pointers in **vp_control**, the **vp_reg** record of the first available register in a queue is found and its fields are modified accordingly. Before any thread accesses the VP, **vp_control** is initialized. No register is used initially, therefore the fields representing the number of registers available for VL=16, 32 or 64 are 64, 32 and 16, respectively. Initially, all 16 registers with VL=64 are ready to be accessed or split; they are arranged into the fast access queue pointed to by **head_64**. The other two access queues for VL=32 and 16 are initially empty. **in_que_64**, **in_que_32** and **in_que_16** are initialized to 16, 0 and 0, respectively.

```
struct vp_reg
{
 int rname; //Register's physical name
 int in_que, used; //Register's status
 vp_reg *prev, *next; //Pointers for implementing the access queue
};

struct vp_control
{
 vp_reg reg_16[64], reg_32[32], reg_64[16]; //Array of all the registers
 vp_reg *head_16, *head_32, *head_64; //Head of access queue for each VL
 int avail_16, avail_32, avail_64; //Number of registers available for each VL
 int in_que_16, in_que_32, in_que_64; //Number of registers in the fast access queue
 int thread_len[4]; //VL for each thread
 int thread_num[4]; //Number of registers used by each thread
 int tlt_table[32][4]; //Mapping of virtual name to physical name
};
```

Figure 4.3 Data structures used to manage the VRF.

4.4 Assigning/Releasing VRF Resources

When a thread requests VP access, its VL and needed number of registers are provided. Based on VL's value, **avail_16**, **avail_32** or **avail_64** within **vp_control** is compared with the latter number. If the remaining number of available registers is not enough for the thread, VP access is denied. Otherwise, the thread is assigned an ID (0 to 3) for unique identification while using the VP, and register allocation begins. **thread_len[ID]** and **thread_num[ID]** in **vp_control** are modified to record the thread's VL and number of registers. Only vector registers in the fast access queue are allocated. When registers of VL=16 are needed, their available number in the queue is checked; if the number is not sufficient, registers in the queue of VL=32 are split. If registers in the queue of VL=32 are not sufficient, registers in the queue of VL=64 are split. Whenever a register of VL=N is split, for N=64 or 32, the respective number of VL=N registers in the queue and the potentially available number of registers are decremented by one. However, for registers of VL=N/2, their number in the queue is incremented by two while their number of potentially available remain unchanged until the register is actually allocated.

After register splitting, there are sufficient registers in the fast access queue representing the VL of the assigned thread. Chosen registers are removed from the queue for allocation. The physical IDs of the registers are stored into TLT and **tlt_table** in **vp_control**. The physical names in **tlt_table** are used later to release VP registers. TLT has three read ports and contains the same information with array **tlt_table**; it supports three VP register name readings per clock cycle. VP uses the 2-bit thread ID concatenated with the 5-bit register ID to form an index into the 128-entry TLT for locating the physical register ID used by a vector instruction. When a thread finishes execution, the **tlt_table**

entries assigned to the thread are identified for releasing its registers. Instead of putting it back into the fast access queue, a released register may be combined with its "sister" register to form a register of higher VL depending on the current status of VRF. For example, **reg_16(15)** is checked when **reg_16(14)** is released. If **reg_16(15)** is not in the access queue, **reg_16(14)** is returned to the queue. Otherwise, the two registers are combined into **reg_32(7)**; it may trigger the recovery of **reg_64(4)** based on the status of **reg_32(6)**.

4.5 Fragmentation Analysis

The proposed VRF management algorithm is designed to minimize register fragmentation by forming registers of larger VL upon releasing VP threads. However, if the VP threads do not complete execution in the reverse order of their VP instantiation, fragmentation can still occur. To evaluate the efficiency of the proposed algorithm, an experiment involving random VP request/release calls was performed. After each request/release call, the number of fragmented **reg_32** and **reg_64** are counted. The number of request failures due to register fragmentation is also counted. Random calls are generated using the **rand()** C function for random integer number generation. When the VP is not occupied by any thread, the call is a request; when the VP is fully occupied by four threads, it is a release; otherwise, release and request have equal probability. For a VP request, all three VLs have the same probability; once the VL is set, all possible numbers of registers for that VL are chosen with equal probability. For a VP release, all the current VP threads have the same probability of being released. Such random calls were repeated 10^9 times. The numbers of fragmented **reg_32** and **reg_64** and their duration (measured in number of calls) are plotted in logarithmic scale in Figure 4.4. In the worst case, two out of the thirty-two **reg_32** and

three out of the sixteen **reg_64** are fragmented. However, fragmented registers are not present more than 98% of the time. 591,441,754 of the 10^9 random calls are for VP requests, and 408,558,246 of them succeed. Among the request failures, only 155,865 are due to fragmentation, thus fragmentation may impact a request only with a 0.026% probability.

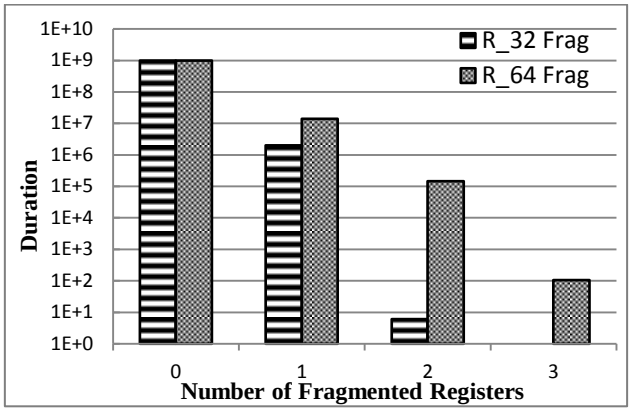


Figure 4.4 Duration of fragmented registers for VL=32 and 64.

4.6 Performance of the VRF Management Algorithm

The VRF management algorithm is designed with performance in mind to minimize the per thread overhead. To measure the runtimes of the VRF management kernels, a MB processor of 100MHz clock rate is attached to a 32-bit timer with accuracy of 10ns for time stamping, and thread request and release functions of various VLs and number of registers are performed by the MB. The measured runtimes of request and release functions for various conditions are shown in Figure 4.5. All measurements are performed on a real system implemented on a ZedBoard all programmable SoC, with VRF kernels and data

structures stored in the fast local memory of the MB, and therefore, the cache hit rates for the kernels are 100%.

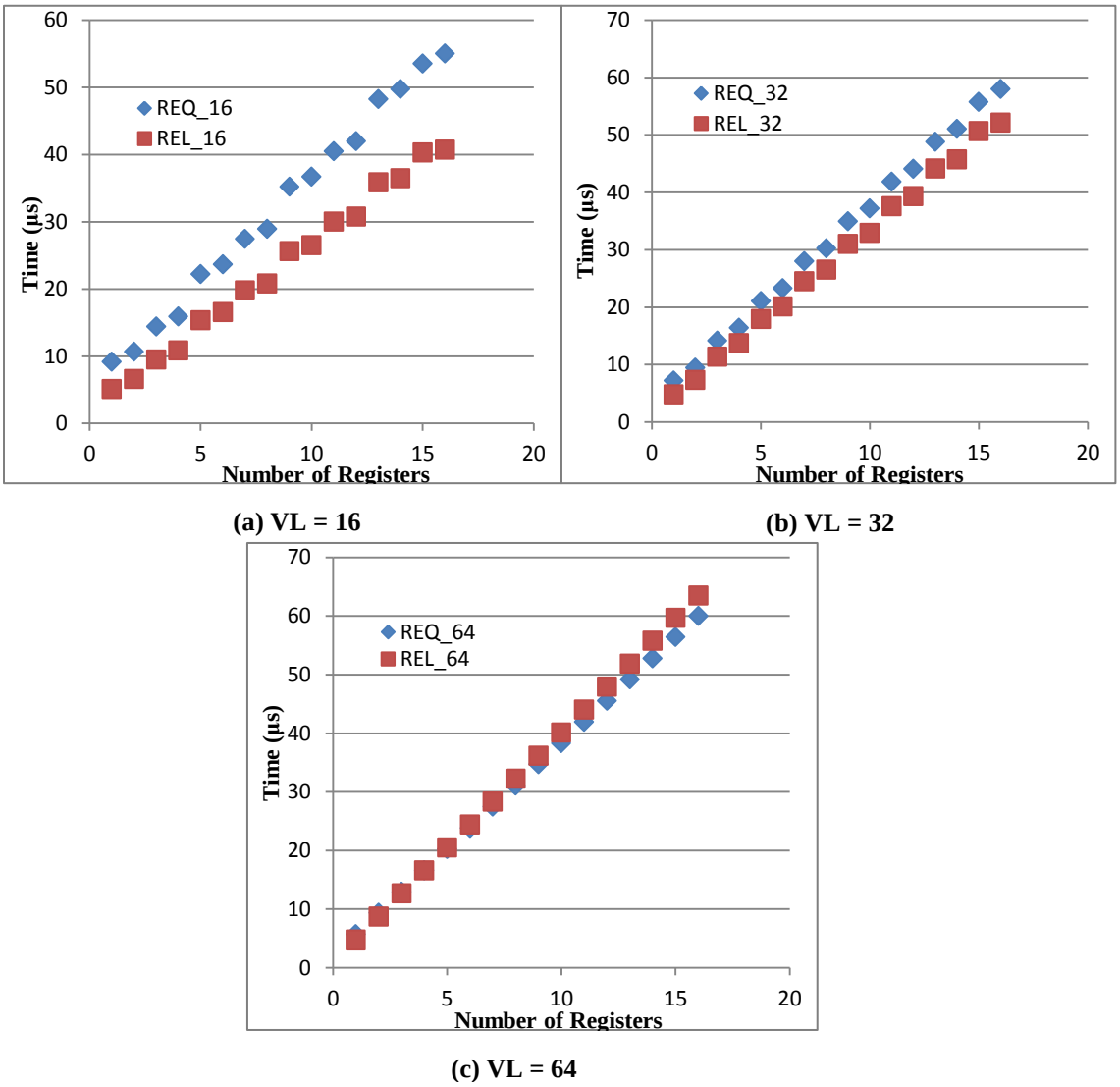


Figure 4.5 Thread request and release functions runtimes under various VLs and number of registers.

As shown in Figure 4.5, the kernels runtimes are linearly related to the number of registers required by the thread. The results also show very impressive performance. For a typical vector application like the 64*64 matrix multiplication in the benchmarking

(Section 6.1), the overall VRF kernel overhead is about 18μs for requesting and releasing two registers of VL = 64. Compared to the application runtime of 3819μs for one matrix multiplication (shown in Table 6.1), the overhead is almost negligible (less than 0.5%).

CHAPTER 5

VIRTUALIZED VP ARCHITECTURE AND FPGA IMPLEMENTATION

The prototype in Figure 5.1 contains two sub-systems: the scalar processors sub-system (SPS) with five cores and VP. SPS does system management, runs the control flow in applications and issues VP instructions. TLT has hardware support for run time VP register renaming, and it is managed by SPS.

5.1 The Host Subsystem Architecture

AXI4 (Advanced eXtensible Interface 4.0) interconnects SPS components. Two AXI4 types, AXI4-Stream (AXI4-S) and AXI4, are also present. AXI4-S pairs realize bidirectional handshaking [Xilinx Inc., 2011]. The interface between SPS and VP is pipelined, and VP can read up to one 32-bit instruction/datum and three 6-bit physical register names from SPS per clock cycle.

MicroBlaze, a Xilinx 32-bit RISC soft processor [Xilinx Inc., 2010], forms SPS cores MB0-MB4. In Figure 5.1, its Harvard architecture interfaces a fast local memory (LM) via a local memory bus (LMB); LM contains frequently used library functions. LM blocks are initialized from the FPGA's flash memory upon power up; these connections are omitted. The libraries can be modified at runtime by MBs. In addition to regular load/store instructions that access memory and I/O devices mapped within the 4GB address space, MB also supports AXI4-S. AXI4-S is used with put/get instructions; its interface consists of one input and one output port, providing a low latency dedicated link to the processor's pipeline. AXI4-S is used for inter-core and core to VP connections. The put/get instructions each has two versions: blocking and non-blocking. Blocking stalls MB if the

receiver/sender is not ready. With non-blocking, MB keeps executing instructions even without needing acknowledgment.

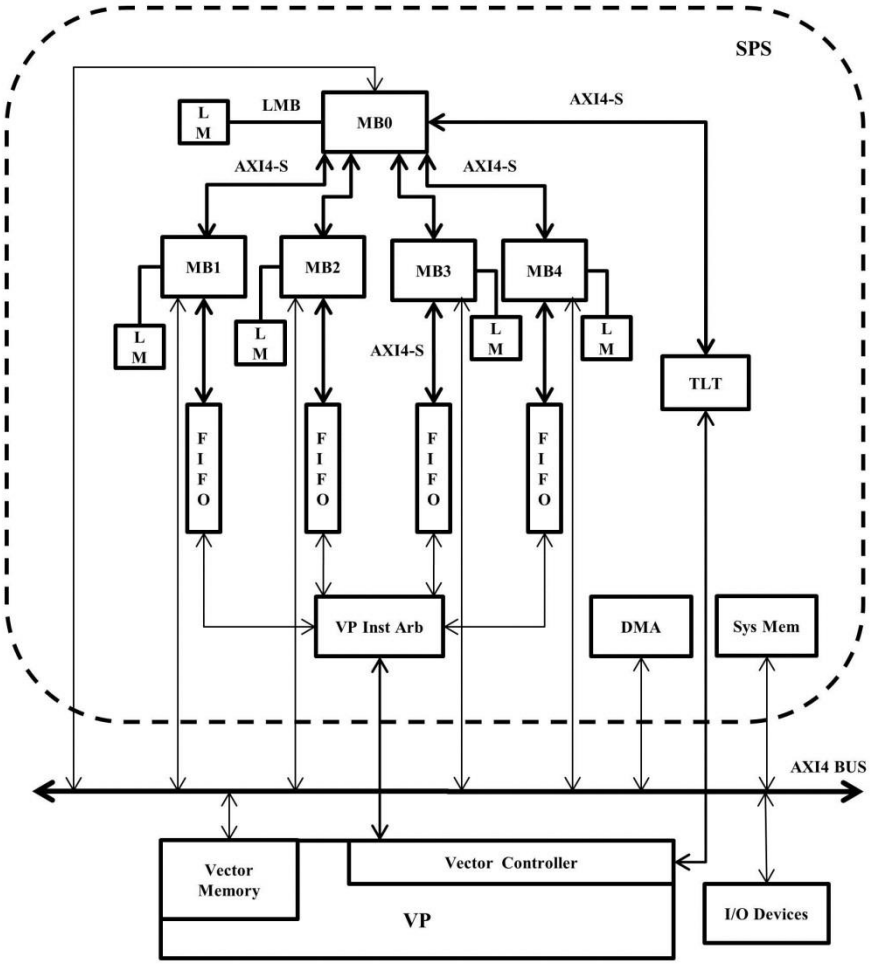


Figure 5.1 Multicore architecture for VP sharing (Instr Arb: vector instruction arbitrator).

MB0 is connected to four MBs and TLT using AXI4-S. MB0 performs these tasks: i) It runs the register management algorithm for VRF virtualization. ii) It updates TLT based on the former algorithm's mapping of a thread's virtual vector registers to physical VRF registers. iii) It estimates VP utilization using information for active vector threads before scheduling new threads. iv) To simplify benchmarking on the proposed prototype,

MB0 notifies application cores (ACs) MB1-MB4 about new tasks assigned to them. v) And, it polls MB1-MB4 for task completion before releasing VP resources.

MB0 is connected to TLT using the output port of its AXI4-S, and uses a non-blocking put since it knows if TLT is ready. Connections between MB0 and slave cores are bi-directional. MB0 assigns tasks to idle cores. With a non-blocking get, MB0 polls each slave for task completion, which is denoted by a task completion flag, for avoiding premature release of VP resources. MB0 is attached to a fast 32KB LM that contains the register management and thread scheduler codes.

MB1-MB4 serve as ACs running applications that may contain function calls to vector kernels. These vector kernels are stored in a library in the attached 16KB LM. For benchmarking simplicity, ACs receive commands from MB0 to execute vector kernels and acknowledge to MB0 their successful completion. ACs apply blocking put/get to communicate with MB0. Another AXI4-S interface connects an AC to its dedicated vector instruction FIFO (see Figure 5.1). An AC generating vector instructions (covered in Section 5.2) forwards them to this FIFO. Each vector instruction goes through the VP instruction arbitrator before reaching the VP.

Each First Word Fall Through (FWFT) vector instruction FIFO contains 16 32-bit words. An AC sends vector instructions or relevant data using blocking puts. An AC keeps issuing vector instructions until its FIFO is full. The latter condition implies VP saturation due to a round-robin arbitrator that gives equitable access to all ACs; it polls non-empty FIFOs. The pipelined arbitrator has two stages for arbitration and handshaking with the VP, respectively. FIFO and arbitrator interconnects accommodate 32-bit transfers per clock cycle.

An AXI4 connects all MBs to the vector and 128 KB system memories with separate read and write channels, and supports incremental bursts for up to 256 32-bit transfers. The vector memory is also accessible by VP. Vector data initially stored in the system memory are moved to VM for processing. A direct memory access (DMA) engine expedites transfers. Each VM bank has two ports; one port directly connects to a lane's LDST unit. With four direct connections between VP lanes and VM banks, a four-fold bandwidth increase is achieved between VP and VM compared to a system with a crossbar [Beldianu and Ziavras, 2013]. The other port of each bank is connected to the system bus in low-order interleaved fashion; sequential data communicated by a MB or the DMA engine are low-order interleaved among the four banks to support fast pipelined access. I/O devices on the system bus support debugging, display and I/O.

5.2 The VP Architecture

VP consists of a VC, data hazard detection unit (HDU), VRF of 1024 32-bit elements, 64KB VM, and four vector lanes; each lane has a LDST unit and a FPU. VM is divided into four low-order interleaved banks; each bank is a true dual-port RAM with one port connected to a distinct vector lane and the other port to the system bus. Each vector lane can only access its own dedicated VM bank; all cores and the DMA controller can access all four VM banks. Application data are initially stored in the system memory, and are transferred for VP processing to VM using either the DMA engine or an AC.

5.2.1 VP ISA and Pipeline

Figure 5.2 shows the architecture of the VP in the proposed prototype. Two types of vector instructions are used. The first type is for vector-vector ALU operations and the instruction

is 32 bits; it does not contain data. The second type contains a 32-bit operand in addition to the 32-bit instruction; e.g., vector-scalar ALU instructions and vector LDST instructions are of this type. Vector instructions are generated by ACs using macro definitions in C and are sent to the VP via the arbitrator interface.

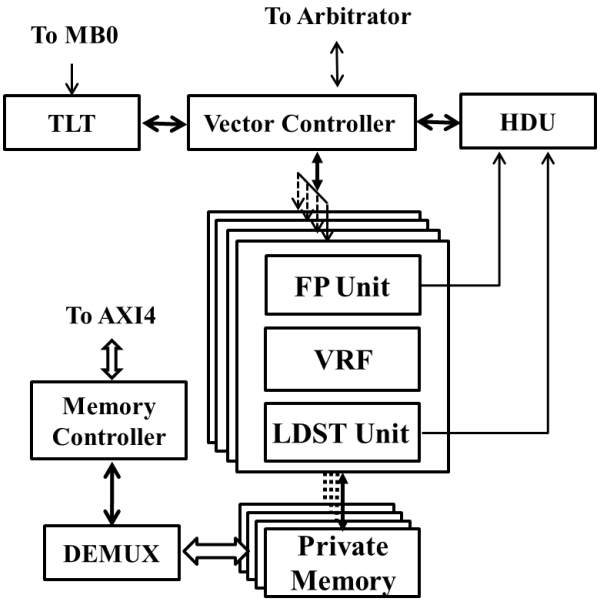


Figure 5.2 Detailed architecture of the four-lane VP (FP: Floating-point).

The first three pipeline stages in the proposed VP's data path are inside VC, which handles register renaming, hazard detection, and assignment of ALU and LDST instructions into separate data paths. The ALU and LDST pipeline stages are shown in Figure 5.3. Two clock cycles are consumed in the ALU or LDST FIFO to pass an instruction and its data to VP. The ALU decode unit consumes four clock cycles for decoding, fetching operands and feeding them to the execution unit. The FPU takes six clock cycles and an extra cycle is needed by write back (WB). The total latency to fill up

the pipeline with ALU instructions is 16 clock cycles (considering both the lane and VC delays).

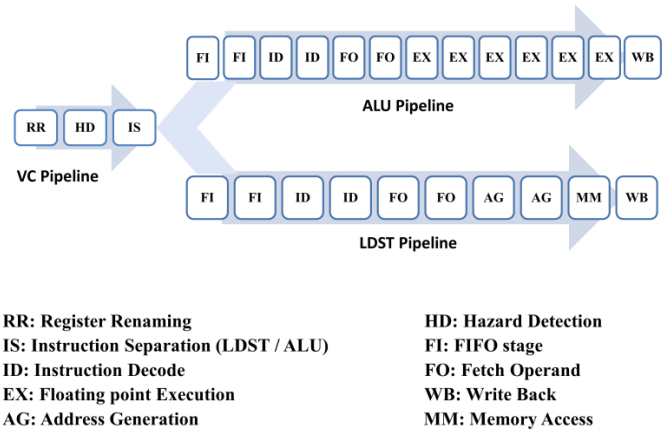


Figure 5.3 Pipeline structure in the LDST and ALU data paths.

Memory access instructions are decoded by the LDST decode unit, which uses six stages with store instructions for data fetching and address generation. For a load from VM, two more clock cycles are added for memory access and WB data latching. Fetching two consecutive vector instructions from a FIFO produces an idle clock cycle between them to ease functional verification and instruction tracking in behavioral simulation. To fill up the pipeline, 11 and 13 clock cycles are needed for a store and a load, respectively. ALU and LDST instructions share the first three stages in VC. The VP's complete ISA for vector as well as control instructions for VP virtualization are listed in Table 5.1. The control instruction __VP_REQ is implemented as a C function which takes an application's VL and the number of registers as input. Upon a successful VP request, the thread ID is returned. The __VP_REL function takes as parameter the thread ID and releases all vector registers occupied by the corresponding thread. Vector application development for the

virtualized VP is almost identical to that for a single-threaded VP. Programmers only have to use the __VP_REQ function to obtain a thread ID and then use it as the ID field for every VP instruction. When an application completes, VP resources must be released using a __VP_REL call.

Table 5.1 ISA of the VP

Target	Instruction	Description
MB	__VP_REQ	Requesting VP resources
	__VP_REL	Releasing VP resources
ALU	__VADD	Vector_vector addition
	__VADD_S	Vector_scalar addition
	__VSUB	Vector_vector subtraction
	__VSUB_S	Vector_scalar subtraction
	__VMUL	Vector_vector multiplication
	__VMUL_S	Vector_scalar multiplication
LDST	__VLD	Vector load (unit stride addressing)
	__VLD_S	Vector load (stride addressing)
	__VST	Vector store (unit stride addressing)
	__VST_S	Vector store (stride addressing)

5.2.2 VP-MB Interface

The arbitrator in the SPS interfaces the VP via the VC. The latter has a pipelined architecture that consists of three stages for register renaming, hazard detection and data path separation, respectively. The VC always gives transaction permission to the arbitrator unless VP resources are not available (i.e., the lane FIFO is full) or a previous instruction has been stalled due to a data dependency. Register renaming is performed by reading physical register names/IDs from the TLT, which is managed and updated by MB0 in the SPS. Each vector instruction uses at most three vector registers, and therefore the TLT is triple-ported. Each vector instruction contains up to three register name fields, which represent the virtual names of the source and destination registers. In the first stage of the

VC (the renaming stage), these virtual names are replaced by their corresponding physical names, which are mapped using the VRF virtualization technique introduced in Chapter 4.

5.2.3 Hazard Detection Unit (HDU)

After updating the register name fields, instructions enter HDU. RAW (Read-After-Write), WAW (Write-After-Write) and WAR (Write-After-Read) data hazards are detected HUD. Hazard information is forwarded to VC that may stall instructions. It can be assumed that there is no dependency across threads. Each HDU module has two separate slots that buffer the previous ALU and LDST instructions of a thread that entered the vector lanes, and two counters that count the number of remaining same-thread ALU and LDST instructions in the lanes. Each buffered instruction is a potential cause of hazard since an incoming instruction may depend on it. The counter of the corresponding instruction type is incremented by one upon issuing a new instruction from the same thread; it is decreased by one when an instruction of its corresponding type completes execution. The ALU and LDST units broadcast an acknowledgment with the thread ID to HDU modules when an instruction completes; the module with the matching thread ID then updates its counters. A zero count implies no pending instruction of its corresponding type in the lane for this thread; thus, there is no need to check the buffered instruction for hazards. When an instruction enters HDU, the HDU module that corresponds to the instruction's thread-ID performs hazard detection. The instruction is compared against both buffered instructions in the module; upon data hazard detection, the instruction is stalled from entering the lanes until any related counter is reduced to zero.

This mechanism adds only one extra pipeline stage and does not decrease the throughput without hazards. With a data hazard, the instruction in the HDU stage stalls

until its dependent has gone through the safe point; by the time the former starts fetching its first operand, the latter will have written its first result. For longer VL instructions, the pipeline will still be fully filled even with a hazard. With VL=16, at most three bubbles will be injected into the pipeline due to a stall. The stall cannot be avoided with in-order execution. However, since the proposed design targets SMT assuming no dependencies among threads, the HDU's performance impact is almost negligible.

5.2.4 Vector Lane Structure

The lane architecture is depicted in Figure 5.4. To reduce the complexity in order to track the progress of instructions through pipelines, simple execution units are chosen. Once a vector instruction passes hazard checking, it is broadcasted to all vector lanes. A lane's VRF consists of 256 32-bit (single-precision FP) elements. It is accessed using three read and two write ports since the ALU and load units need two and one read port, and the WB and store units require one write port each. The design needs one clock cycle to send an output. All read ports are configured with an "enable" for power efficiency. The ALU decode unit requires two read ports when reading a pair of operands for vector-vector operations. A lane's ALU execution unit contains a floating-point adder/subtractor and a multiplier derived from open source code. Compared to a FP multiplier, an IEEE-754 adder/subtractor includes two to three extra stages for exponent comparison and mantissa alignment [Ehliar, 2014]. Hence, it has six pipeline stages for addition and subtraction, and four for multiplication. The results are sent to the WB block, which is connected to a write port of VRF for writing one element per clock cycle in a pipelined fashion.

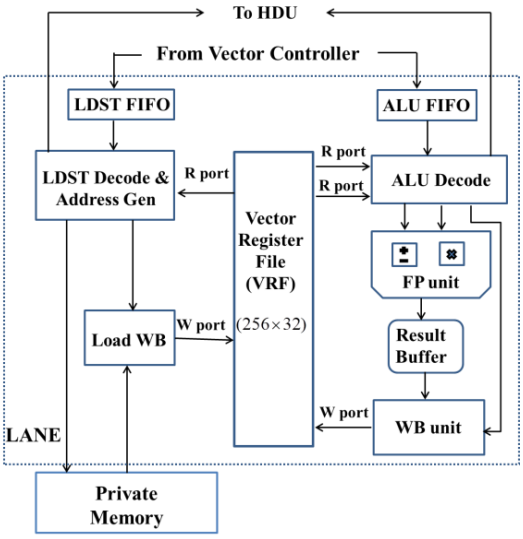


Figure 5.4 Vector lane architecture.

LDST instructions use absolute memory addressing with a unit or non-unit stride. Each lane is connected to a private VM bank, and therefore memory accesses are never stalled. Arbitration deteriorates performance if all memory banks are accessible to all lanes [Beldianu and Ziavras, 2013]. The ALU and LDST decode blocks in each lane include counters for synchronization across lanes; counts are initialized based on the VL value in instructions. Vector instructions with different VLs may coexist in VP.

5.3 FPGA Implementation

The proposed prototype uses a Xilinx Virtex6 xc6vlx240t FPGA device. The entire VP, arbitrator and TLT are custom designed in VHDL. The rest of the system uses IP cores in Xilinx ISE. The system is fully synthesized and routed. The chosen clock frequency of 100MHz is the result of the open source FPU codes. Critical path delay analysis shows that the VP's clock cycle could be as low as 7.01 ns (i.e., representing 142.65 MHz)

corresponding to the adder. This delay is due to 32 levels of logic. The earliest and latest signal arrival times are 1.897 ns and 2.126 ns, respectively.

Table 5.2 shows the resource consumption. This FPGA contains 37,680 slices; each slice has eight registers and four 6-input lookup tables (LUTs). Each register is implemented with flip-flops or latches, and each LUT may be composed of a pair of 5-input LUTs. Some LUTs are implemented as small RAM blocks which are known as distributed RAMs. Large RAM memory can be realized using 36Kbit BRAM blocks (RAMB36E1). Embedded digital signal processor (DSP) slices (DSP48E1) contain a hardwired 25x18 two's complement multiplier/accumulator. The proposed FPUs are designed with custom ASIC logic without DSP slices. Only four DSP48E1s are used, one for each lane's address calculator in the LDST unit. The VP subsystem and its SPS interface (including the vector instruction FIFOs, arbitrator and TLT) consume 13.9% and 45.8% of the total registers and LUTs. The resource consumption of FPGA-based designs is also affected by the randomness of the routing process. Some registers and LUTs are used as wires and buffers to reduce critical path delays. In this dissertation, benchmarking relies on cycle accurate behavioral system simulation. For highly accurate power measurements, post place-and-route simulation is performed at a fine detail, down to the switching of individual LUTs. The binaries for each benchmark are generated and used as testbenches to obtain Switching Activity Interchange Format (SAIF) files, which are then used by the Xpower Analyzer to derive accurate power consumption.

Table 5.2 Resource Consumption of the VP Prototype

Entity	Slice Registers (% Utilization)	Slice LUTs (% Utilization)	RAMB36E1s (% Utilization)	DSP48E1s (% Utilization)
A Vector Lane	10247 (3.4%)	17035 (11.3%)	0 (0%)	1 (<1%)
VM (4 Banks)	16 (<1%)	272 (<1%)	16 (3.8%)	0 (0%)
VC (Including HDU)	358 (<1%)	305 (<1%)	0 (0%)	0 (0%)
VP (VC+4 Lanes+VM)	41378 (13.7%)	68717 (45.6%)	16 (3.8%)	4 (<1%)
VP/SPS Interface	388 (<1%)	283 (<1%)	0 (0%)	0 (0%)
VP + VP/SPS Interface	**41766 (13.9%)**	**69000 (45.8%)**	**16 (3.8%)**	**4 (<1%)**
SPS	9962 (3.3%)	15268 (10.1%)	73 (17.5%)	23 (3%)

CHAPTER 6

BENCHMARKING AND HOMOGENEOUS SMT

This chapter covers all the benchmarks used throughout the dissertation for evaluating the proposed VP virtualization technique. Homogeneous multithreading runs of multiple copies of each benchmark are also performed to illustrate the effect of VP utilization saturation.

6.1 Benchmark Details

As shown in Figure 6.1, two basic types of vector instructions are sent to VP: without (**type V_instr_a**) and with a scalar operand (**type V_instr_b**). Macro definitions ease programming by providing an assembly-like VP programming interface. As an example, Figure 6.1 shows the macro definition for the 32-bit *__ADD (vector-vector add)* type a instruction, and the *__VLD (unit-stride load)* and *__VST (unit-stride store)* type b instructions that hold an extra 32-bit scalar operand as address. The main function in Figure 6.1 loads two 16-element vectors from VM and stores the summation result back into VM. To compile benchmarks written in C that also contain macros and assembly code for vector instructions, the MB GNU mb-gcc tool without optimization (i.e., option o0) is applied.

The first benchmark is *matrix multiplication (MM)* for square matrices of size 16*16, 32*32 and 64*64. All elements in a row of the result matrix are calculated in one loop iteration to maximize the vectorization ratio (i.e., ratio of vector to scalar code). It multiplies a single element of the first matrix with all elements on a row of the second matrix to produce partial products. To calculate row i in the result, each element on row i of the first matrix is multiplied with the respective row in the second matrix and appropriate

partial products are summed up. All multiplications are performed using scalar-vector multiplications, and additions are of the vector-vector type. Using an optimal approach, only two vector registers of size VL are needed. The execution time measured for MM is based on the time needed to generate the entire result matrix. Through some simple calculations of the results shown in Table 6.1, it can be seen by increasing the dimensionality of the matrix and consequently VL, the time needed to generate each element in the result decreases slightly (due to a higher vectorization ratio).

```
// Functions defining two types of vector instructions, with and w/o data
#define V_instr_a(instr)  asm volatile("put\t%0,rfsl1\t\n" ::"d"(instr))
#define V_instr_b(instr,data)  asm volatile ("cput\t%0,rfsl1\t\n" \
"put\t%1,rfsl1\t\n"  ::"d"(instr),"d"(data))

// Based on the above, define vector instructions as macros
// Constant X_SHIFT determines the location of field X within 32-bit instruction
#define __VADD(VDst,VSrc_1,VSrc_2,VL,Id)\
V_instr_a((OP_VADD<<OP_SHIFT)|(VDst<<DST_SHIFT)|(VSrc_1<<SRC1_SHIFT)|\
(VSrc_2<<SRC2_SHIFT)|(VL<<VL_SHIFT)|(Id<<THREAD_ID_SHIFT))

#define __VLD(VDst,BaseAddr,VL,Id)   V_instr_b((OP_VLD<<OP_SHIFT)|(VDst<<DST_SHIFT|\
(VL<<VL_SHIFT)|( Id<<THREAD_ID_SHIFT), BaseAddr)

#define __VST(VSrc,BaseAddr,VL,Id)   V_instr_b((OP_VST<<OP_SHIFT)|(VSrc<<SRC_SHIFT|\
(VL<<VL_SHIFT)|(Id<<THREAD_ID_SHIFT), BaseAddr)

int main(){              // For VL=16 & thread=0
__VLD(0,adr1,16,0);    // Load from location adr1 to r0
__VLD(1,adr2,16,0);    // Load from location adr2 to r1
__VADD(2,0,1,16,0);    // r2 ← r0+r1
__VST(2,adr3,16,0);    // Store r2 into location adr3
};
```

Figure 6.1 Macros to define vector instructions.

The second benchmark is *Finite Impulse Response (FIR) digital filter* that uses the outer product [Sung and Mitra, 1987]. 16, 32 and 64 tap FIR filters are implemented with the input sequence having the same size as the filter; the resulting sequence has twice the input's length. A loop unrolling technique expands the kernel four times and increases the vectorization ratio. Two vector registers of size VL were used for this benchmark. The

execution time for FIR is measured based on the time needed to generate an entire result vector which has twice the size of the filter.

The third benchmark is *vector-dot product (VDP)* with VL= 16, 32 and 64. A vector-vector multiplication is followed by two vector-vector additions. Four VL-sized vector registers are used. The execution time of VDP is measured for an input of one pair of array having VL elements per thread.

The fourth benchmark is the *discrete cosine transform (DCT)* which is common in video processing. Since DCT is usually applied on fixed-sized pixel blocks, like 8*8 or 4*4, one-dimensional 8-point DCT on blocks of size 8*8 were performed in the benchmark. 2, 4 and 8 adjacent blocks are used as input with VL=16, 32 and 64, respectively. Three vector registers of size VL are used. The execution time measured for DCT is based on the time to produce 2, 4, and 8 blocks of 8-point DCT for VL 16, 32, and 64, respectively.

The last benchmark is *RGB to YIQ color space mapping (RGB2YIQ)*. It has the highest portion of vector code among all benchmarks and uses seven vector registers. Configurations of VL=16, 32 and 64 were used to perform the calculation on a 1024-pixel block. The execution time for RGB2YIQ is measured based on the time needed to handle a block of 1024 pixels. Since the input size is independent of VL, higher VL leads to fewer loop iterations, and therefore shorter execution times.

6.2 Homogeneous SMT Results

In this chapter only, it is assumed each time simultaneous VP runs of up to four threads from the same benchmark. The only exception is RGB2YIQ with VL=64 since it requires seven registers per thread while the proposed VP has 16 registers of VL=64; up to two threads of SMT was performed for RGB2YIQ. 58 simulations are done for various VLs

and degrees of multithreading. For clarity, the times for task request and register management are excluded from the obtained measurements. Since the threads start execution at the same time and the SPS's VP interface involves a round-robin arbitrator, all threads finish execution at the same time. Tables 6.1 to 6.5 show the execution times and VP utilization of these benchmarks for various numbers of VL and active cores (i.e., threads). The execution times are for the input size described above.

Table 6.1 Matrix Multiplication Performance (Input Matrix Size: VL*VL, 1 Iteration per Core)

VL	# of cores	LDST NWT	ALU FLOP	Execution Time (μs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	4608	8192	241	53.11	4.78	8.49	84.97
	2	9216	16384	241	106.22	9.56	16.99	169.95
	3	13824	24576	241	159.33	14.34	25.49	254.93
	4	18432	32768	241	212.44	19.12	33.99	339.91
32	1	34816	65536	942	106.53	9.23	17.39	173.38
	2	69632	131072	942	213.06	18.47	34.78	346.76
	3	104448	196608	942	319.59	27.72	52.17	520.19
	4	139264	262144	942	426.12	36.96	69.57	693.53
64	1	270336	524288	3819	208.07	17.69	34.32	337.8
	2	530672	1048576	3819	416.14	35.39	68.64	675.69
	3	811008	1572864	4221	564.76	48.03	93.15	917.01
	4	1081344	2097152	5625	565.06	48.05	93.20	917.5
NWT: Number of Word Transactions								

Table 6.2 FIR Performance (Input Vector Size: VL, 1 Iteration per Core)

VL	# of cores	LDST NWT	ALU FLOP	Execution Time (μs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	576	1024	27	59.25	5.3	9.4	78.8
	2	1152	2048	27	118.51	10.6	18.9	157.4
	3	1728	3072	27	177.77	16	28.4	236.1
	4	2304	4096	27	237.04	21.3	37.9	314.8
32	1	2176	4096	51	122.98	10.6	20	153.07
	2	4352	8192	51	245.96	21.3	40	306.15
	3	6528	12288	51	368.94	32	60	459.23
	4	8704	16384	51	491.92	42.6	80	612.31
64	1	8448	16384	97	256	21.77	42.22	354.13
	2	16896	32768	97	512	43.54	84.44	708.26
	3	25344	48152	133	552.6	47.63	90.0	774.83
	4	33792	65536	177	561.17	47.72	92.56	776.29

Table 6.3 VDP Performance (Input Vector Size: VL, 1 Iteration per Core)

VL	# of cores	LDST NWT	ALU FLOP	Execution Time (µs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	112	64	2.4	73.33	11.6	6.6	4.88
	2	224	128	2.4	146.66	23.2	13.3	9.77
	3	336	192	2.4	220	34.8	20	14.65
	4	448	256	2.4	293.33	46.4	26.6	19.54
32	1	288	160	3	149.33	24	13.33	8.1
	2	576	320	3	298.66	48	26.6	16.2
	3	864	480	3	448	72	40	24.3
	4	1152	640	3.4	527.05	84.7	47.05	28.58
64	1	704	448	3.6	320	48.8	31.1	13.05
	2	1408	896	4	576	88	56	23.5
	3	2112	1344	6	576	88	56	23.5
	4	2816	1792	8	576	88	56	23.5

Table 6.4 DCT Performance (Input: VL/8 Blocks of Size 8*8, 1 Iteration per Core)

VL	# of cores	LDST NWT	ALU FLOP	Execution Time (µs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	4224	2048	87	72.09	12.13	5.96	7.98
	2	8448	4096	87	144.18	24.27	11.92	15.97
	3	12672	6144	87	216.27	36.41	17.89	23.96
	4	16896	8192	87	23.85	48.55	23.85	31.95
32	1	8448	4096	87	144.18	24.24	11.57	19.2
	2	16896	8192	87	288.36	48.55	23.51	38.4
	3	25344	12288	87	432.55	72.82	32.25	57.65
	4	33792	16384	94	533.78	89	43.15	71.14
64	1	16896	8192	87	288.36	48.55	23.53	48.55
	2	33792	16384	109	460.33	77.5	37.57	77.50
	3	50688	24576	132	557.51	93.86	45.51	93.86
	4	67584	32768	176	557.51	93.86	45.51	93.86

Table 6.5 RGB2YIQ Performance (Input: 1024 Pixels, 1 Iteration per Core)

VL	# of cores	LDST NWT	ALU FLOP	Execution Time (µs)	Million FLOP/S	% LDST Utilization	% ALU Utilization	Speedup
16	1	6144	15360	244.2	88.05	6.29	15.72	358.13
	2	12288	30720	244.2	176.11	12.58	31.45	716.26
	3	18432	46080	244.2	264.17	18.87	41.74	1074.39
	4	24576	61440	244.2	352.23	25.16	62.9	1432.53
32	1	6144	15360	123.6	173.98	12.43	31.06	707.57
	2	12288	30720	123.7	347.68	24.83	62.08	1415.14
	3	18432	46080	155.8	414.06	29.57	73.49	1690.51
	4	24576	61440	204.1	421.44	30.10	75.25	1713.98
64	1	6144	15360	63.74	337.37	24.09	60.24	1372.07
	2	12288	30720	96.7	444.57	31.76	79.43	1808.8
	3	18432	46080	NA	NA	NA	NA	NA
	4	24576	61440	NA	NA	NA	NA	NA

In this chapter, simultaneously active threads from an application are homogeneous but independent, and their control flows are executed on different MBs. Threads operate on their own input data for higher throughput. Chapter 7 deals with the simultaneous execution of heterogeneous threads with different VLs arriving from different MBs. VP utilization with a single thread is very low for all benchmarks when VL=16; as more threads/cores are involved, the utilization improves substantially. As VL increases, the utilization of a thread increases up to a saturation point. As explained earlier, an idle clock cycle between issuing successive instructions decreases the maximum utilization but eases the verification of functional behavior. Due to this effect, the nominal maximum utilization that can be achieved for VL=16, 32 and 64, is calculated as 80%, 88.88% and 94.11%, respectively. For a low VP-utilization benchmark, the total execution time of multiple threads may be almost the same as the benchmark's *native duration* (i.e., a thread's execution time with exclusive VP usage). When the total VP utilization with many simultaneous threads exceeds the VP's nominal maximum, all threads' execution are slowed down proportionally due to resource competition. When either the ALU or LDST unit saturates, the other unit's utilization may not increase further since ALU and LDST operations may depend on each other. Among the five basic benchmarks, MM, FIR and RGB2YIQ have higher ALU utilization that leads to VP saturation. VDP and DCT have higher LDST utilization that may lead to LDST saturation. Upon VP saturation, the slowdown amount is determined by the higher of the ALU and LDST utilizations. Figure 6.2 shows the larger of ALU and LDST utilizations for various benchmarks, VLs and core numbers. The performance of RGB2YIQ with VL=64 saturates for two cores although the ALU utilization is not close to the nominal maximum of 94%. It happens when threads

produce high VP utilization and many data hazards, causing frequent VC stalls. For each benchmark, sequential C code with identical functionality and behavior was also run on a 100 MHz MB. The last column in the tables is the speedup of VP versus scalar core runs.

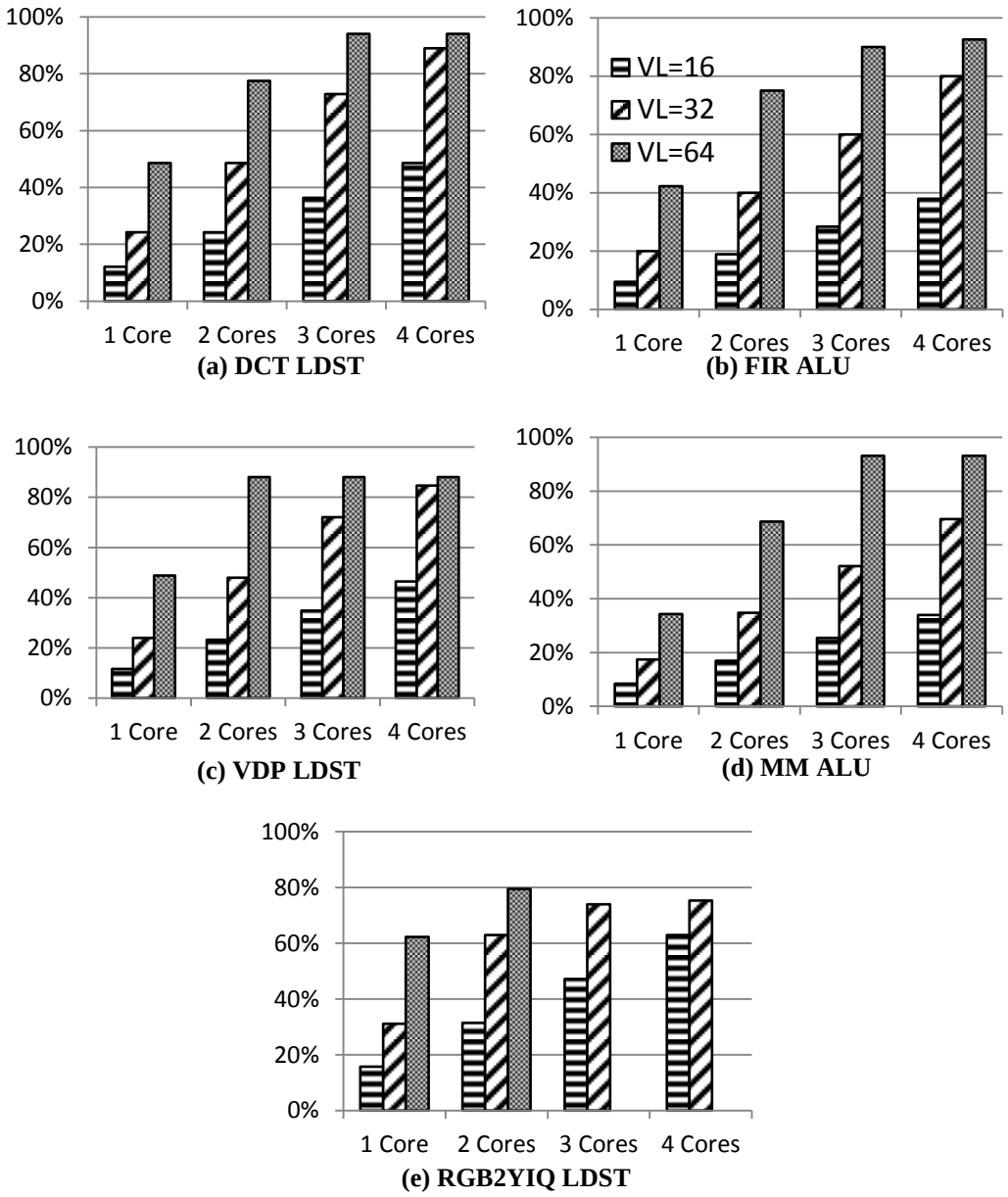


Figure 6.2 Utilization of the LDST or ALU units for various benchmarks, VLs, and number of simultaneous threads.

6.3 Comparison with Prior Works

To perform a fair performance comparison with prior works that focused on VP sharing for multicores, a common reference point is chosen. Since VP speedups against host processors were listed in these works, the same is done in this dissertation. Moreover, the chosen benchmark scenarios are similar (including identical VLs). Table 6.6 shows comparisons with [Beldianu and Ziavras, 2013] that implemented an 8-lane shared VP for two cores using the CTS, FTS and VLS policies. FTS has the best performance among these policies. As per Section 2.2, FTS is similar to the proposed VP sharing technique. [Rooholamin and Ziavras, 2015] used a VP with many similarities to ours. It utilized a hardware scheduler and a register renaming block to support VP sharing for two threads with identical VL. It relied on compiler optimizations to increase the instruction issue rate. The proposed virtualization yields better speedup than other techniques even with half the lanes.

Table 6.6 Speedup Comparison with Prior Works

SYSTEM \ BENCHMARK	MM	FIR	RGB2YIQ	VL
[Rooholamin and Ziavras 2015], 4 lanes,1 core	92.66	73.32	383.32	16
The proposed VP, 4 lanes, 4 cores	**339.91**	**314.8**	**1432.53**	
[Beldianu and Ziavras 2013], CTS, 8 lanes, 1 core	12.97	10.93	NA	32
[Beldianu and Ziavras 2013], FTS, 8 lanes, 2 cores	25.89	21.83	NA	
[Rooholamin and Ziavras 2015], 4 lanes,1 core	193.06	150.94	762.22	
The proposed VP, 4 lanes, 4 cores	**693.53**	**612.31**	**1713.98**	
[Rooholamin and Ziavras 2015], 4 lanes,1 core	403.50	360.12	1512.44	64
The proposedVP, 4 lanes, 4 cores	**917.50**	**776.29**	**1808.80**	

CHAPTER 7

SCHEDULING VECTOR THREADS

The focus of this chapter is on throughput-maximizing thread scheduling. Each application is profiled to determine its ALU and LDST utilizations, as well as its native duration (i.e., its execution time with exclusive VP access). Using the profile information, it is possible to evaluate combinations of simultaneously executing benchmarks (from the set of 15 in Chapter 6) for: i) A closed system with a fixed number of threads. ii) An open system with randomly arriving threads.

7.1 The Scheduling Algorithm

As observed in Chapter 6, when the ALU and LDST utilizations are both far below 90%, the performance is upper bounded by the speed of the ACs that issue vector instructions, and therefore multiple threads could share the VP with only negligible increase in the per-thread execution time. Due to the one clock cycle delay between consecutive instructions (Section 5.2.1), the proposed VP's saturation threshold is not 100% but a number from 80% to 94% depending on the active threads' VLs. The assumption of a saturation threshold of 90% is used to design a scheduling algorithm that keeps the VP highly busy either with zero or minimum saturation.

With a closed system, no new threads are added into the queue before all threads in the current queue have been completely executed. Once a thread is picked by the scheduler for execution, it will keep executing until the end and then releases the VP resources for other pending threads. All pending threads are arranged in descending order of their native duration. The ALU and LDST utilization as well as the VRF usage of each thread are

provided to the scheduler as input. The scheduler keeps picking pending threads for execution until the VP is filled with four threads, or no pending thread can be accommodated by the VP due to unavailable VRF resources. As shown in the flowchart of Figure 7.1, the scheduler searches down the queue until a fitting thread is found which does not lead to saturation. If no such thread is found, the thread update mechanism ensures that the scheduler has to search down the queue only once to find a fitting thread that results in the minimum saturation.

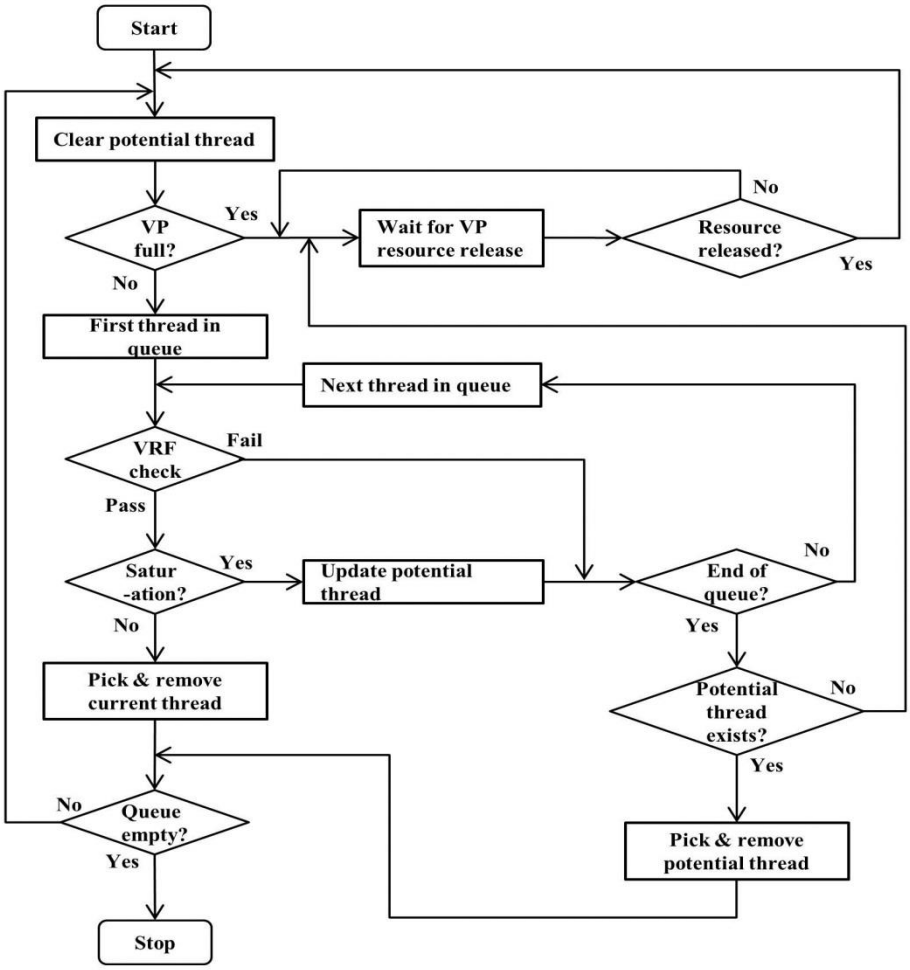


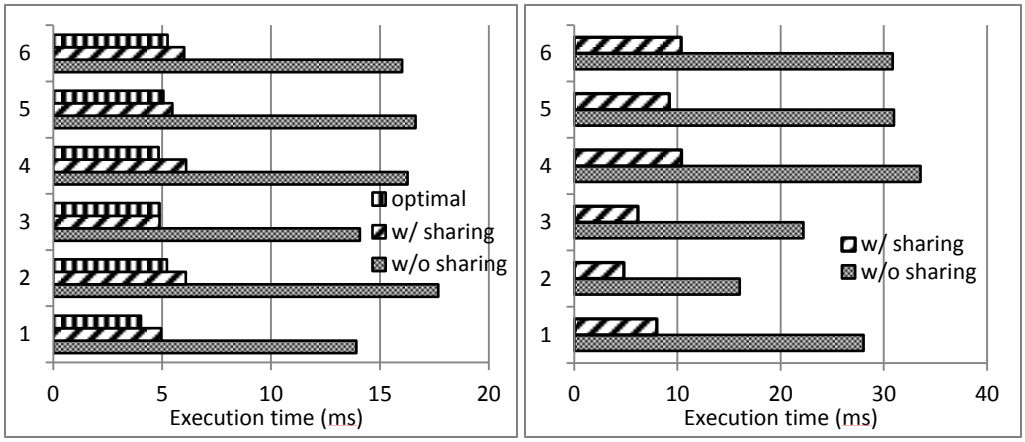
Figure 7.1 Software flowchart of the throughput-driven scheduler.

The scheduler always starts checking from the first pending thread that has the longest native duration among all threads. If the available VRF resources are sufficient to accommodate the thread, utilization saturation check is performed to see whether this thread will lead to an ALU or LDST overall utilization that exceeds 90%. If no saturation will occur, the thread will be chosen for scheduling. Otherwise, the thread will not be directly picked but will become the "potential thread" for scheduling. When another thread in the queue is found to lead to utilization saturation, it will be compared against the currently potential thread. If the former thread is expected to yield smaller ALU and LDST overall utilizations than the currently potential thread, then the former will replace the latter as the potential thread for scheduling. When the entire queue has been searched and all threads are either not fitting or leading to saturation, the currently potential thread will be chosen for immediate scheduling.

7.2 Queues of Fixed Length

The scheduler was tested for a closed system with two queue sizes: 8 and 16 pending threads. Six successive schedules of random thread combinations were tested. Threads and their input data size were chosen with equal probability from the list of 15 benchmarks of Chapter 6. The average execution time per schedule is shown in Figure 7.2. To identify the optimal solution for the six schedules with queue length 8, exhaustive search was applied (i.e., a C program produced the total execution time of all permutations of involved threads). Compared to the optimal case, which cannot be implemented in practice, the achieved execution time is only 14.7% slower on the average and actually achieves optimality in one of the six schedules. For a queue length of eight, the achieved average speedup is 2.83 compared to the case without VP sharing; when the queue length increases

to 16, the average speedup increases to 3.33. With increases in the thread queue, the speedup approaches four, this is ideal since it matches the maximum thread population. One of the six schedules for each queue length was chosen to generate tables with detailed simulation information (Table 7.1 and 7.2).



(a) **(b)**

Figure 7.2 Average execution time per schedule for pending thread queues of length; **(a)** 8, **(b)** 16.

Table 7.1 Detailed Results for a Schedule with Pending Thread Queue Length of 8

Task ID	Application	VL	Native Duration (μs)	% ALU Utilization	% LDST Utilization	Issue Time (μs)	Commit Time (μs)	Actual Duration (μs)
0	MM	16	4820	9	5	11	4905	4894
1	VDP	64	3600	31	49	30	4348	4318
2	DCT	64	2610	24	49	3075	6083	3008
3	FIR	16	2025	9	5	44	2109	2065
4	MM	32	1884	17	9	60	1967	1907
5	RGB2YIQ	64	1268	60	24	2680	4655	1975
6	VDP	16	960	7	12	1994	3048	1054
7	FIR	32	510	20	11	2132	2642	510
Practical issue order based on static scheduling: 0,1,3,4,6,7,5,2. Optimal order based on simulation of all permutations: 0,3,6,4,1,2,5,7. Actual execution time = 6.083ms. Optimal execution time = 5.215ms. Total native duration w/o VP sharing = 17.677ms. Speedup = 2.91.								

Table 7.2 Detailed Results for a Schedule with Pending Thread Queue Length of 16

Task ID	Application	VL	Native Duration (μs)	% ALU Utilization	% LDST Utilization	Issue Time (μs)	Commit Time (μs)	Actual Duration (μs)
0	MM	64	3819	34	18	11	3829	3818
1	MM	32	2826	17	9	24	2873	2849
2	RGB2YIQ	32	1483.2	31	12	54	1705	1651
3	MM	16	964	8	5	1111	2080	969
4	DCT	32	860	12	24	1740	2606	866
5	DCT	64	783	24	49	2632	3460	828
6	DCT	16	693	6	12	78	771	693
7	FIR	64	679	42	22	3533	4338	805
8	FIR	16	675	9	5	2101	2789	688
9	RGB2YIQ	64	634	60	24	4030	4815	785
10	VDP	32	630	13	24	3511	4357	846
11	FIR	32	561	20	11	2907	3468	561
12	RGB2YIQ	16	488.4	16	6	2837	3470	633
13	VDP	64	356.4	31	49	3863	4397	534
14	DCT	32	348	12	24	3559	3988	429
15	VDP	16	240	7	12	820	1070	250
Practical issue order based on static scheduling: 0,1,2,6,15,3,4,8,5,12,11,10,7,14,13,9. Actual execution time = 4.815ms. Total native duration w/o VP sharing = 16.053ms. Speedup = 3.33.								

7.3 Open System with Randomly Arriving Threads

To simulate an open system with randomly arriving tasks, a time slice of 10ms is chosen and all tasks arriving within each 10ms time slice will be scheduled together as a queue of fixed length. A fixed input size was chosen for each benchmark to create 15 distinct tasks. The characteristics of each task are listed in Table 7.3. Dynamic energy measurement is the focus of Section 8.1. The average task native duration is 0.182ms. Task arrival follows the Poisson distribution with a rate of λ tasks arriving per time slice. Tasks arriving in a time slice form a queue which is scheduled for execution in the next time slice. The evaluation is for λ=0.5, 0.75 and 1; for a given λ, queues for six consecutive time slices were generated, and all the average values for the six schedules were calculated. The average of the total execution time for all threads scheduled in a time slice is shown in Figure 7.3. Details of task arrivals and execution times are shown in Tables 7.4 to 7.6. The speedup compared to

the VP without sharing is 2.59, 3.15 and 3.22 for λ=0.5, 0.75 and 1, respectively. The speedups concur with the results obtained earlier for fixed thread queue lengths where the speedup increased with the thread population. Without VP sharing and scheduling, even for the lowest thread arrival rate the queue increases faster than the system can process. With the proposed scheduling, the VP is active only 80% of the time slice for the highest λ= 1. The rest of the time the VP can be power gated to reduce the static energy (Section 8.2).

Table 7.3 Characteristics of Chosen Tasks for an Open System

Task ID	Application_VL	Native Duration (μs)	% ALU Utilization	% LDST Utilization	Vector Registers	Dynamic Energy (μJ)
0	RGB2YIQ_16	4884	16	6	7	766
1	MM_64	3819	34	18	2	792.3
2	MM_32	2826	17	9	2	404.1
3	RGB2YIQ_32	2472	31	12	7	535.8
4	FIR_64	1940	42	22	2	577.2
5	DCT_64	1740	24	49	3	417.8
6	DCT_32	1740	12	24	3	288.2
7	DCT_16	1740	6	12	3	207.8
8	MM_16	1446	8	5	2	152.34
9	RGB2YIQ_64	1268	60	24	7	354.4
10	FIR_32	1020	20	11	2	255
11	VDP_64	720	31	49	4	192.8
12	VDP_32	600	13	24	4	123.6
13	FIR_16	540	9	5	2	85.8
14	VDP_16	480	7	12	4	70.8

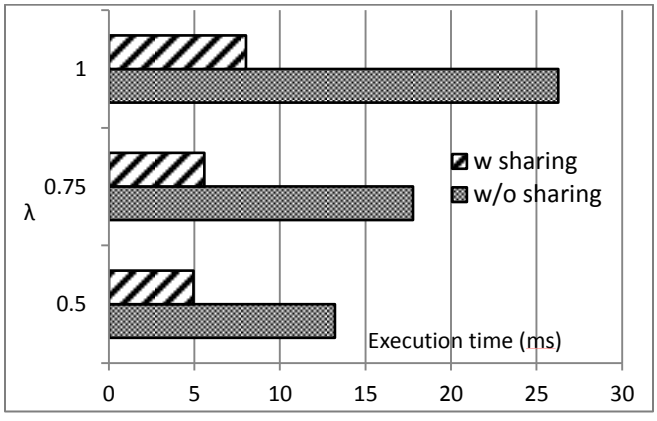


Figure 7.3 The average of the total execution time for all threads scheduled in a time slice, with and without VP sharing, for λ= 0.5, 0.75 and 1. (Time slice: 10ms.)

Table 7.4 Detailed Task Arrivals and Execution Time for λ=0.5

Task ID	Application_VL	Number of Task Arrivals						Average
		Slice1	Slice2	Slice3	Slice4	Slice5	Slice6	
0	RGB2YIQ_16	1	1	1	1	0	0	0.66
1	MM_64	1	0	0	0	0	2	0.5
2	MM_32	0	0	0	0	0	0	0
3	RGB2YIQ_32	2	0	0	0	0	0	0.33
4	FIR_64	1	0	1	1	0	0	0.5
5	DCT_64	0	1	0	0	1	1	0.5
6	DCT_32	0	1	0	0	0	0	0.16
7	DCT_16	0	0	0	1	0	1	0.33
8	MM_16	3	2	1	0	0	1	1.16
9	RGB2YIQ_64	2	0	0	0	0	1	0.5
10	FIR_32	0	0	0	1	0	0	0.16
11	VDP_64	1	0	1	0	1	0	0.5
12	VDP_32	2	1	1	1	2	0	1.16
13	FIR_16	0	1	2	2	1	2	1.33
14	VDP_16	2	0	1	0	0	0	0.5
Total Native Duration (ms)		25.3	12.39	11.15	11.26	4.2	14.9	13.21
Actual Duration (ms)		8.22	4.9	4.9	4.9	1.8	4.7	4.9
Speedup		3.08	2.52	2.26	2.28	2.26	3.12	2.59

Table 7.5 Detailed Task Arrivals and Execution Time for λ=0.75

Task ID	Application_VL	Number of Task Arrivals						Average
		Slice1	Slice2	Slice3	Slice4	Slice5	Slice6	
0	RGB2YIQ_16	0	0	2	0	0	0	0.33
1	MM_64	0	1	0	2	0	0	0.5
2	MM_32	2	0	0	0	1	0	0.5
3	RGB2YIQ_32	1	2	0	1	0	1	0.83
4	FIR_64	1	1	1	0	1	1	0.83
5	DCT_64	1	1	1	2	0	1	1
6	DCT_32	0	0	0	0	0	1	0.16
7	DCT_16	1	2	1	1	0	0	0.83
8	MM_16	2	3	1	1	0	2	1.5
9	RGB2YIQ_64	1	0	2	1	1	0	0.83
10	FIR_32	1	0	1	0	1	0	0.5
11	VDP_64	3	2	0	0	1	1	1.16
12	VDP_32	0	2	1	0	0	0	0.5
13	FIR_16	1	0	1	4	1	0	1.16
14	VDP_16	0	1	0	0	1	0	0.33
Total Native Duration (ms)		21.4	23.38	21.33	20.2	8.79	11.5	17.77
Actual Duration (ms)		6.59	6.75	6.66	6.62	3.05	3.75	5.57
Speedup		3.25	3.46	3.20	3.05	2.88	3.06	3.15

Table 7.6 Detailed Task Arrivals and Execution Time for λ=1

Task ID	Application_VL	Number of Task Arrivals						Average
		Slice1	Slice2	Slice3	Slice4	Slice5	Slice6	
0	RGB2YIQ_16	0	0	2	0	0	0	0.33
1	MM_64	0	1	0	2	0	0	0.5
2	MM_32	2	0	0	0	1	0	0.5
3	RGB2YIQ_32	1	2	0	1	0	1	0.83
4	FIR_64	1	1	1	0	1	1	0.83
5	DCT_64	1	1	1	2	0	1	1
6	DCT_32	0	0	0	0	0	1	0.16
7	DCT_16	1	2	1	1	0	0	0.83
8	MM_16	2	3	1	1	0	2	1.5
9	RGB2YIQ_64	1	0	2	1	1	0	0.83
10	FIR_32	1	0	1	0	1	0	0.5
11	VDP_64	3	2	0	0	1	1	1.16
12	VDP_32	0	2	1	0	0	0	0.5
13	FIR_16	1	0	1	4	1	0	1.16
14	VDP_16	0	1	0	0	1	0	0.33
Total Native Duration (ms)		21.4	23.38	21.33	20.2	8.79	11.5	17.77
Actual Duration (ms)		6.59	6.75	6.66	6.62	3.05	3.75	5.57
Speedup		3.25	3.46	3.20	3.05	2.88	3.06	3.15

CHAPTER 8

VP ENERGY CONSUMPTION

In this chapter, the energy consumption for the benchmarking of Section 6.1 is investigated. Based on the power dissipation of individual benchmarks, a projection is made of the total energy consumption for the dynamic schedules of Section 7.3. Power consumption has three components: device static, design static and design dynamic [Beldianu and Ziavras, 2015]**.** The device static power, also known as leakage power, is a device specific constant not related to resource utilization or switching activity. Under the default simulation conditions for an ambient temperature of 50^0C and an airflow of 250LFM (linear feet per minute), the leakage power for the chosen FPGA is 2.88W. The design static power represents the power consumption when the device is configured but there is no switching activity. It includes the static power in I/O DCI terminations, clock managers, etc., and is related to FPGA resource consumption. The design dynamic power results from the switching of the user configured logic. Accounting for the FPGA resources that the proposed VP actually uses, the power model in this chapter adds the design's static and dynamic powers to estimate the total dissipation.

8.1 VP Dynamic Power

To reliably estimate the dynamic power, the proposed VP design was fully implemented and all signal switching activities of each system node were used as input for power calculation. The proposed VP was fully implemented (i.e., synthesized, translated, placed and routed) using the Xilinx ISE tool chain, and post place-and-route (PAR) ISE simulations were performed. The binaries of the vector instructions of each benchmark

were generated to estimate the dynamic power. All signal switching activities during each simulation were recorded in an SAIF File. The SAIF file, with two other files generated during the implementation of the design, namely the Native Circuit Description and Physical Constraint files, were fed into the Xilinx power analyzer (XPA) to produce the VP's accurate power dissipation for each benchmark [Xilinx Inc., 2012] . The power measurements here include all power consumed by VP subsystems (i.e., VC, HDU, vector lanes, VRF and VM). Also, register name readings from TLT contributed to the figure.

Due to the time consuming nature of PAR simulations, only the average power consumption for one iteration of each vector kernel was measured. For matrix multiplication, the innermost loop that involves three vector instructions is considered as the target kernel. It is repeated VL times to produce one row of the resulting matrix. This kernel includes one load, one vector-scalar multiplication and one vector-vector addition. For FIR filtering, the target kernel for power estimation is the internal loop which is unrolled four times, slides the coefficients four times over the input sequence, and carries out multiplications and additions to produce four elements of the result. This kernel contains twelve vector instructions: four loads, four vector-scalar multiplications and four vector-vector additions. For VDP, the kernel size depends on VL. This kernel contains 11, 14 and 18 vector instructions for VL=16, 32 and 64, respectively. For VL=16, the kernel consists of five loads, two stores, three vector-vector additions and one vector-vector multiplication. For VL=32, one load, one store and one vector-vector addition are added to the former case. For VL=64, two loads and two vector-vector instructions are added to the VL=32 case. For DCT, the inner loop which calculates the output result for one output coefficient is the kernel. This kernel contains six instructions: two loads, two stores, one

vector-vector multiplication and one vector-vector addition. For RGB2YIO, the chosen kernel converts the color space for VL input pixels. It contains 21 instructions: three loads, nine scalar-vector multiplications, six vector-vector additions and three stores.

For VP power measurements of individual benchmarks, the VP is used exclusively without competition. The total dynamic energy consumed by a benchmark is actually the product of its vector kernel power consumption and its native duration. The dynamic power and energy consumptions of individual benchmarks are shown in Table 8.1. The energy numbers shown are based on the input data sizes of Section 6.1. Using the measured power, it can be calculated the total dynamic energy consumption of each benchmark for various native durations; this approach aids the estimation of the energy consumption in dynamic environments. The dynamic energy results for the predefined tasks of Section 7.3 were included in Table 7.3. Using a task's average number of arrivals per time slice, its average dynamic energy consumption per slice can be obtained easily. Figure 8.1 shows that the dynamic energy consumption is related almost linearly to the task arrival rate.

Table 8.1 Power and Energy Consumption for Benchmarks

Applic-ation	VL	Kernel Duration (ns)	VC+4Lanes+Memories Dynamic Power (mW)		Kernel Dynamic Power (mW)	Application Duration (μs)	Application Dynamic Energy (μJ)
			Signal & Logic	BRAM & IO			
MM	16	365	102.04	3.32	105.36	241	25.39
	32	405	136.96	6.04	143	942	134.7
	64	555	198.68	8.8	207.48	3819	792.3
FIR	16	895	153.68	5.44	159.12	27	4.29
	32	935	239.6	10.48	250.08	51	12.75
	64	1575	284.6	13	297.6	97	28.86
VDP	16	765	136.26	11.24	147.5	2.4	0.35
	32	1235	187.4	19.04	206.44	3	0.62
	64	2275	243.28	24.92	268.2	3.6	0.96
DCT	16	525	110.16	9.32	119.48	87	10.39
	32	605	149	16.64	165.64	87	14.41
	64	775	212.28	27.92	240.2	87	20.89
RGB2YIQ	16	1465	152	5	157	244	38.3
	32	1805	209.64	8.2	217.84	123	26.79
	64	2295	267.76	13.52	281.28	63	17.72

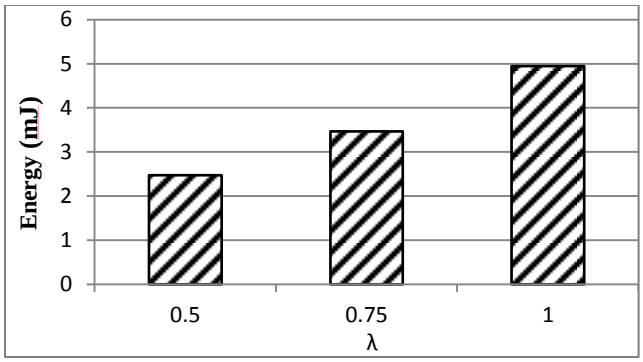


Figure 8.1 Average total dynamic energy consumption per time slice for λ=0.5, 0.75 and 1.

8.2 Total Energy Consumption

The VP's static power is measured without running instructions but just applying the clock signals. For a 100μs measurement after system reset, the average static power is 214mW. Without pending instructions for the VP, power-gating (PG) can be applied to shut off the VP and zero its static power dissipation. Implementing PG requires sleep transistors, isolation cells and circuits to control power signals. It can reduce the design static power by 85% [Beldianu and Ziavras, 2015].

Although commercial FPGAs currently lack PG support, PG in association with the proposed dynamic scheduler of Section 7.3 could yield not only performance gains but also substantial reduction in the overall energy consumption. In each time slice, once the task queue becomes empty, the VP is PGed until the beginning of the next time slice. Using the obtained static power measurements, the assumption of a 85% static power reduction with PG and the measured average execution time in Figure 7.3, it can be projected the VP's average static energy consumption per time slice for a given task arrival rate. Combining the results with the dynamic energy of Figure 8.1, Figure 8.2 shows the effect of PG on the

VP's energy consumption with and without VP sharing. The total energy saved by combining VP sharing, proper scheduling and PG is 33.9%, 36.1% and 37% under task arrival rates of **λ**=0.5, 0.75 and 1, respectively. These are major energy savings on top of the already achieved very substantial performance improvements.

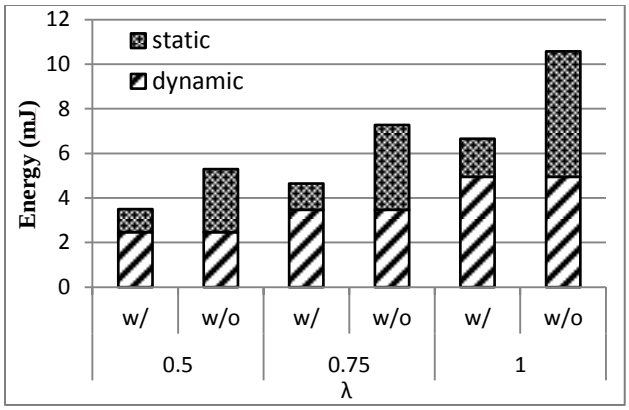


Figure 8.2 Total energy consumption with (w/) and without (w/o) VP sharing, and with power gating, for λ=0.5, 0.75 and 1.

CHAPTER 9

VIRTUALIZED VP FOR THREAD FUSION AND DYNAMIC LANE CONFIGURATION

In this chapter an improved VP prototype is presented that combines the virtualization and instruction fusion technologies. Thread fusion can be triggered once similar threads are identified in the vector task queue. In the fused mode, one vector instruction is interpreted as multiple instructions each working in a per thread independent VRF and VM address space, and therefore the effective vector instruction issue rate from the host processor is multiplied. The new VP prototype is also capable of dynamically deactivating some of its lanes to minimize unnecessary static power consumption with minimum impact to performance. Through the complete virtualization of the VP, vector applications can execute properly under different fusion state and various numbers of active lanes and do not to be recompiled. An accurate power model of the new prototype is derived to help choose the optimal number of active lanes for different applications. Two optimization policies are proposed to minimize the total energy consumption of an application, or to minimum the product of energy consumption and application runtime.

9.1 Virtualized VM Address Space

The new VP prototype still features a distributed VM design, where one of each VM bank's dual ports is assigned exclusively to one VP lane, and yet all VM banks can be accessed by the host processors via a mux connected to the second port of every VM bank. The benefit of this feature is that when a lane is power gated based on the optimization policy, its dedicated VM bank can also be deactivated, thus further reducing the static

power dissipation. Since the VM is accessed by two heterogeneous types of masters (i.e., the on-chip host cores and the VP), it is assigned two different address domains with regard to each one of its masters. The host-to-VM mux accesses VM banks in low-order interleaved fashion to hide the bank selection details from the hosts; therefore, all VM banks appear as one large memory module with a continuous address space on the system bus. Each VP lane, on the other hand, can only access and process elements within its dedicated VM bank based on the VP-to-VM issued address and VL information, which is within vector instructions sent from the hosts.

All vector instructions from the hosts go through the VC that handles hazard detection and virtualization, and then broadcasts them to the ALU or LDST pipeline interface in each lane. To ensure the correct execution of a vector application under various numbers of active lanes, both address domains of the VM as well as the VL information must be virtualized. This is essential for dynamic VP lane configuration, since all address values and the VL for a vector application are determined statically, and therefore the same values must be properly interpreted at runtime by the hardware under disparate VP configurations. To facilitate address virtualization, the host-to-VM mux and the VC are designed to be configurable by host requests. Before starting a new vector thread, a host will submit a request to configure state registers based on the optimal number of lanes needed by the thread.

Figure 9.1 illustrates how a data array with base host-to-VM address of 4N and VL=8 can be accessed by the virtualized VP correctly under different lane configurations. The figure shows the cases of two-lane (Figure 9.1a) and four-lane (Figure 9.1b) configurations in a VP with four lanes; however, this scheme can be easily adapted for any

2^N active lanes with any VL = 2^M, where N and M are both natural numbers and M ≥ N. The VP's lane state register, which can be configured dynamically by the hosts via a simple control instruction, stores the number of active lanes and determines how the VP behaves in the following cases.

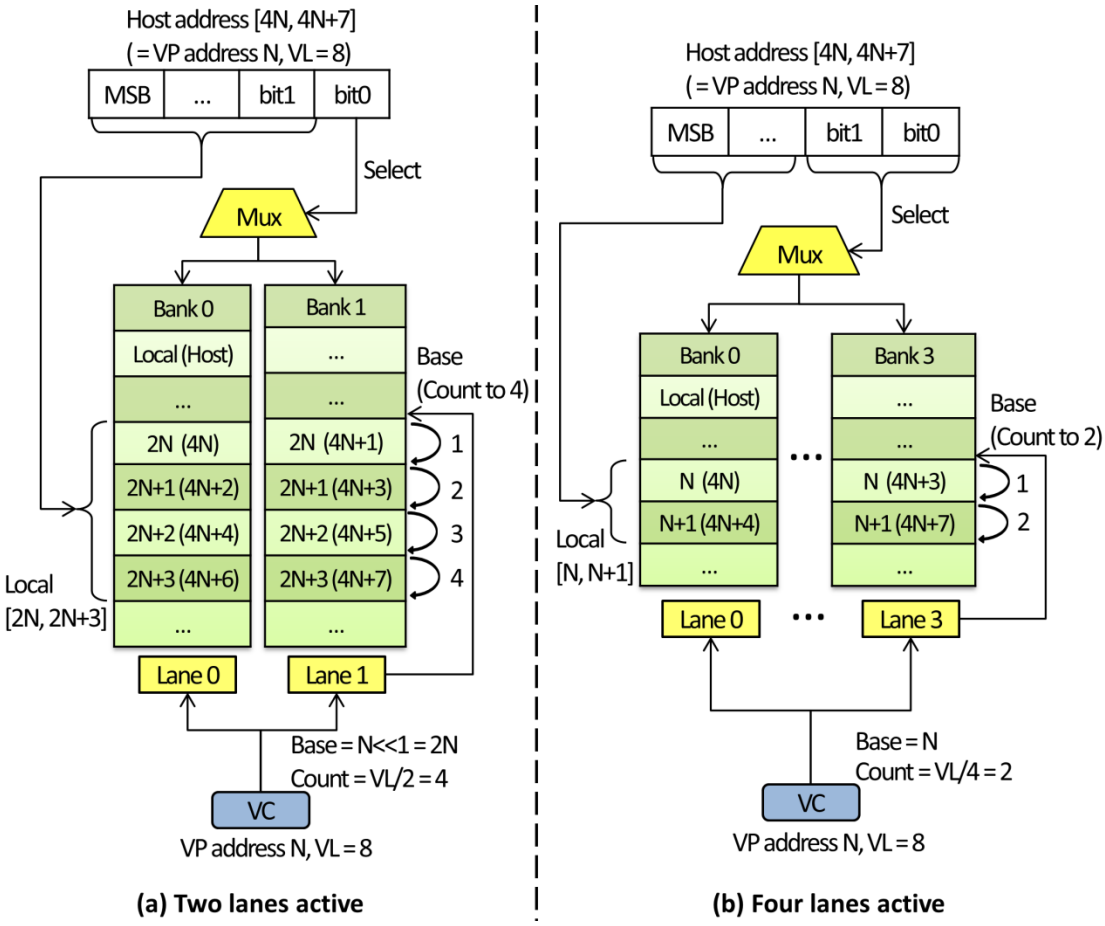


Figure 9.1 Mapping of VL, host-to-VM address and VP-to-VM address via virtualization.

In the case of all lanes being active (four in this example), the lowest two bits of the host-to-VM address will be used as the select signal for the host-to-VM mux, and the remaining bits will be used as the actual physical address for every VM bank. As shown in Figure 9.1b, the data array is mapped to the physical address [N, N+1] of each VM bank,

with two elements per bank. Therefore, the array's base VP-to-VM address is compiled to be N, which is the same as its physical address in each bank. The LDST unit within each lane will start accessing the array with base address N, and based on the VL = 8 and four active lanes information passed from the VC, the instruction decoder will set the counter to two so that each lane will access two elements per instruction.

When the host dynamically deactivates two VP lanes and their attached VM banks, only two banks remain active and therefore the mux must be configured to take only the LSB of the host-to-VM address as the bank select signal. All remaining bits will be used as each bank's physical address, and since the host-to-VM address is compiled at static time and does not change, under the new configuration the array will be mapped to the physical address [2N, 2N+3] of each remaining VM bank, with four elements per bank. To ensure that the VP can still reach the array with the unchangeable VP-to-VM address of N, the VC's virtualization stage simply has to shift left the address by one bit and pass it to all lanes' LDST units. The new configuration also requires that the VC shift left the VL by one bit to make each lane access four elements per instruction. Since the decoder unit in each lane relies on the register name and VL value to locate the right vector registers, shifting VL also ensures that each lane will use the right location and number of registers under the new configuration.

9.2 SMT VP and Thread Fusion

The new VP prototype is modified based on the SMT capable VP introduced in Section 5.2, which already supports per instruction VRF virtualization. To achieve true SMT where instructions from multiple threads can coexist inside the VP pipeline without interference, further support of VM space virtualization is added on a per instruction basis. To simplify

the hardware design, the current VP implementation only supports two simultaneous virtual threads. With the two threads set up, the software VRF management algorithm introduced in Section 4.3 is no longer needed and both VRF and VM virtualization can be performed by hardware. However, the number of virtual threads can be easily expanded by using the software VRF management algorithm and by using a memory management unit with the VP. With SMT virtualization, one SMT capable VP appears as multiple logical VPs (LVPs) to multiple hosts/cores. Shown in Figure 9.2 is a simple example of an SMT VP of degree two. The VP has only one physical instruction input channel; however, the FIFOs and arbitrator structure create two virtual channels. The VP input arbitrator accepts instructions from two different FIFOs in round-robin fashion, and each FIFO can be assigned to a host; in this example, only one host is used and the two LVPs are used to exploit thread level parallelism via thread fusion. Each instruction has its LSB as the thread ID that is filled by the arbitrator based on the source FIFO. For ID = 0, all VRF names are unchanged. When ID = 1, the virtualization stage in the VC properly flips a few bits in each register name based on the instruction's VL. The scheme ensures that LVP0 occupies the lower half of the VRF and LVP1 occupies the higher half. The mechanism achieves VRF resource sharing with significant flexibility in that it allows both LVPs to function correctly as long as (a) the total VRF usage does not exceed the available physical VRF resources, and (b) in the single LVP mode, either LVP0 or LVP1 can occupy the entire VRF space.

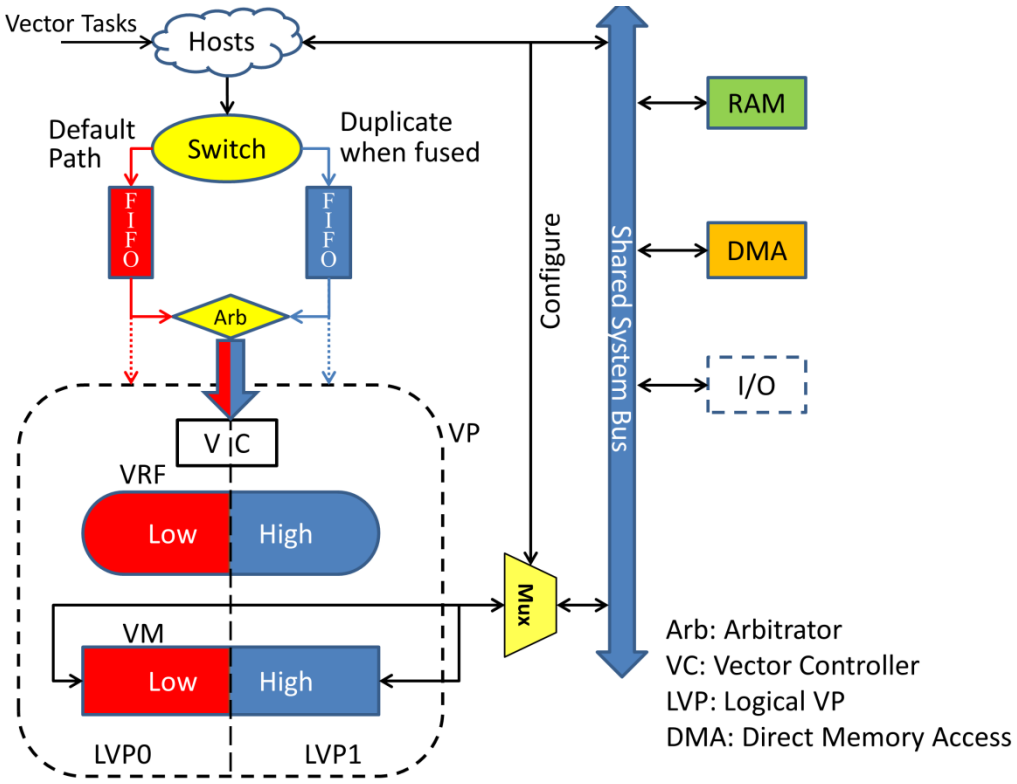


Figure 9.2 System architecture of a fusion capable VP of degree two.

As shown in Figure 9.2, the host-to-VM mux supports data transfers between the hosts' RAM space and both LVPs' virtual VM spaces. Based on the thread state register, which can be configured by the hosts, part of the host-to-VM address is flipped to map the LVP1's virtual address space to the higher half of the VM banks. The data transfer only happens at the beginning and the end of a vector application, and therefore no per instruction switching between LVP0 and LVP1 is required for data transfers. The thread state register can be configured by the hosts using a simple control instruction which is similar to that used for dynamic lane configuration (introduced in Section 9.1). The virtualization for SMT capability does not conflict with that for dynamic lane

configuration, and therefore the prototype is extremely versatile; without recompilation, any two applications can simultaneously function properly on the VP regardless of their assigned thread ID or the number of active VP lanes.

For frequently used computation intensive operations, highly optimized VP routines are implemented and stored in a library. When multiple pending tasks are of the same operation, it is possible to fuse these operations thanks to the VP's per thread virtual VM and VRF space. Figure 9.3 shows how two Discrete Cosine Transform (DCT) operations are accelerated by fusing the threads. Without fusion (Figure 9.3a), the two operations will be executed sequentially. When two threads are fused (Figure 9.3b), the major parts of their execution are merged, so that the hosts' domain issues vector instructions only once while the VP receives two copies. The switch in Figure 9.2 is set to the fusion state for duplicating each vector instruction from the host domain and sending it to both FIFOs. A scheduler of vector threads decides on fusion. Pending threads are ordered in their operation ID (i.e., type of operation performed by the thread) within the task queue. Every time the scheduler picks a thread, it checks the next thread in the queue to compare the operation ID. When a match is found, thread fusion will be triggered and the two threads performing the same operation will be executed in fused mode. Due to the independent virtual nature of each LVP, the two identical instruction flows will perform the same operation but on different input data within each virtual space.

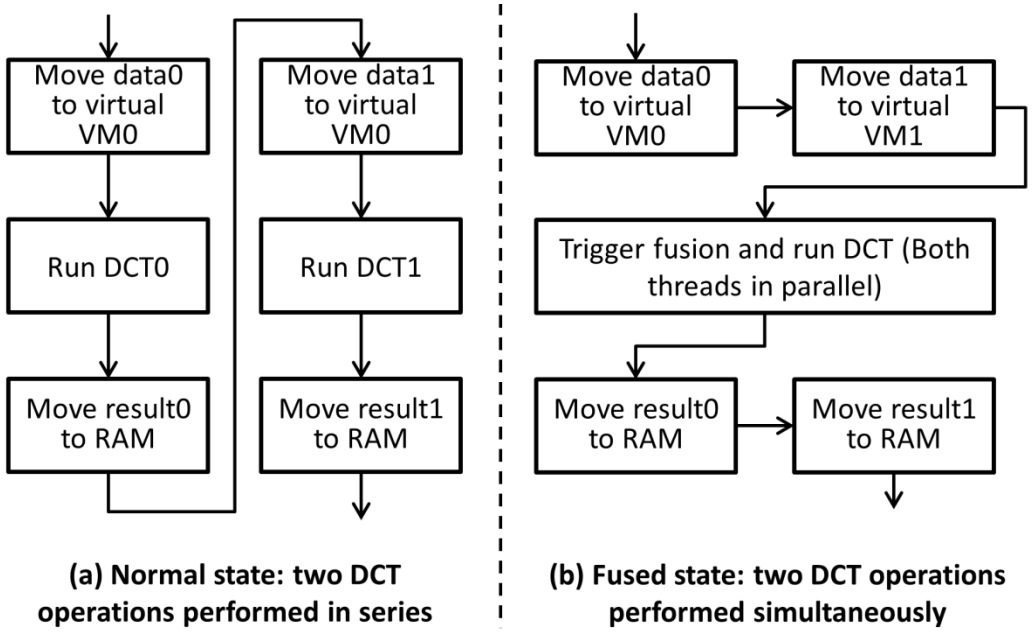


Figure 9.3 Fusion of two DCT operations.

Vector thread fusion has many benefits: (a) it significantly increases the vector instruction issue rate for all hosts; (b) the VP utilization is effectively multiplied by the degree of fusion as long as the aggregate utilization does not exceed 100%; (c) it reduces the overall energy consumption since the host domain only has to run the flow control program once to send out vector instructions for fused threads; (d) since the VP's SMT virtualization is compatible with dynamic lane configuration, fusion can be combined with lane configuration to optimize performance and energy figures

9.3 System Architecture and FPGA Implementation

To evaluate the benefit of dynamic lane configuration and thread fusion, a dual-threaded VP interfaced with a hosts system is prototyped on a Xilinx XC7Z045-1fbg676 FPGA. The system architecture is similar to that in Figure 9.2, with the hosts system replaced by a MB processor that issues vector threads. Various vector kernels are stored in the 16KB local

memory of the MB processor. The system RAM and VM are 64KB each. A DMA engine is attached to the system bus for fast data transfers between the system RAM and VM. The mux connecting the VM and system bus is configurable by the MB to support the virtualization for lane configurability and SMT, as per Sections 9.2 and 9.3. I/O components on the bus are used for debugging purposes and an 8-bit LED is implemented to show the system status. A cycle accurate timer (not shown in the figure) that measures application runtime can interrupt the MB.

The VP has four lanes and is capable of running with 1, 2, or 4 active lanes. Each lane's dedicated VM bank can be deactivated with its assigned lane. The VRF can store 1024 32-bit elements, with all the registers evenly distributed across the four vector lanes. Without loss of generality, the VP supports three VLs (16, 32, and 64) in this implementation. A vector register of length N contains N register elements, and therefore the number of available vector registers depends on the VL of each register. The VP, the fusion switch, the VM data mux and the vector instruction arbitrator are custom hardware designed in VHDL, and the rest of the system components are Xilinx IPs. The target FPGA has a speed grade of -1. The minimum achievable critical delay is 6.01ns; for simplicity, the system is implemented at 100MHz. The resource consumption breakdown for the VP is shown in Table 9.1. The rest of the custom hardware components are not shown as they consume negligible amount of resources compared to the VP (< 1%). RAMB36E1 is a Xilinx block RAM IP that can be configured to various data widths and depths as long as the total memory capacity is within 32Kbits.

Table 9.1 Resource Consumption and Utilization Percentage of the New VP Prototype

Entity	Registers U(%)	LUTs U(%)	RAMB36E1s U(%)	DSP48E1s U(%)
One Lane	9571 (2%)	17437 (7%)	0	5 (<1%)
VM	16 (<1%)	272 (<1%)	16 (2%)	0
VC	287 (<1%)	451 (<1%)	0	0
VP	38674 (8%)	70143 (32%)	16 (2%)	20 (2%)

9.4 Benchmarking

Four vector applications are picked, namely DCT, FIR, RGB, and VDP. Since the current VP implementation supports three different VLs, each picked application is evaluated using all supported VLs, creating a total of 12 benchmarks. The VP has separate pipelines for ALU and LDST operations; each benchmark is characterized by its ALU utilization (U_{ALU}) and LDST utilization (U_{LDST}). Each benchmark is executed under various configurations to measure the corresponding runtime, and the corresponding utilizations of the pipelines are calculated. The utilization is defined as O_{total}/O_{4lanes}, where O_{total} is the total number of operations for an application and O_{4lanes} is the maximum number of operations that can be performed by the four lanes during the application's runtime. Tables 9.2 to 9.4 show the runtime and utilization figures under three VP configurations (**a**. Four lanes active without fusion. **b**. Four lanes active with fusion. **c**. Two lanes active without fusion.)

Table 9.2 Performance Profile Data for Unfused VP with Four Active Lanes

APP	VL	T(µs)	ALU(%)	LDST(%)
DCT	***16***	75	6.8	14.1
	32	75	13.6	28.2
	64	75	27.3	56.4
VDP	***16***	23.7	6.7	11.8
	32	28.3	14.1	25.4
	64	34.4	32.5	51.1
RGB	***16***	243.6	15.8	6.3
	32	123.8	31.0	12.4
	64	64.0	60.0	24.0
FIR	***16***	25.7	10.6	5.5
	32	46.8	22.7	11.5
	64	89.1	47.8	24.0

Table 9.3 Performance Profile Data for Fused VP with Four Active Lanes

APP	VL	T(µs)	ALU(%)	LDST(%)
DCT	***16***	75	13.6	28.2
	32	75	27.2	56.4
	64	86.5	47.36	97.6
VDP	***16***	23.7	13.4	23.6
	32	28.3	28.2	50.8
	64	35.8	62.6	98.4
RGB	***16***	243.7	31.6	12.6
	32	123.5	62.1	24.9
	64	78.3	98.1	39.2
FIR	***16***	25.9	21.1	10.9
	32	46.7	45.5	22.9
	64	89.2	95.7	47.9

Table 9.4 Performance Profile Data for Unfused VP with Two Active Lanes

APP	VL	T(µs)	ALU(%)	LDST(%)
DCT	***16***	75	6.8	14.1
	32	75	13.6	28.2
	64	84.9	24.1	49.8
VDP	***16***	23.7	6.7	11.8
	32	28.3	14.1	25.4
	64	35.7	31.3	49.2
RGB	***16***	243.6	15.8	6.3
	32	123.8	31.0	12.4
	64	77.7	49.4	19.8
FIR	***16***	25.7	10.6	5.5
	32	46.8	22.7	11.5
	64	89.03	47.8	24.0

An application's figures under configuration **a**. are denoted as *native utilization (U)* and *native runtime (T).* Utilization and runtime figures under other configurations are represented by *U'* and *T'*. With two active lanes, the maximum achievable utilization is

50%; it is the average with two active lanes at 100% and the other two lanes at 0%. For benchmarks with ALU and LDST native utilizations below 50%, the runtime and utilizations will not be affected by lane deactivation. For other benchmarks, from U_{ALU} and U_{LDST} the higher will hit the 50% saturation level while the other will decrease proportionally. The runtime increase is related to the higher of U_{ALU} and U_{LDST}. The relation between each benchmark's actual figures for two active lanes and their native figures is shown in Equation 9.1.

$$
\begin{aligned}
&\text{if } (U_{ALU} < 50 \text{ and } U_{LDST} < 50) \text{ then} \\
&\quad U'_{ALU\_2lanes} = U_{ALU};\ U'_{LDST\_2lanes} = U_{LDST};\ T'_{2lanes} = T \\
&\text{else if } (U_{ALU} > U_{LDST}) \text{ then} \\
&\quad U'_{ALU\_2lanes} = 50;\ U'_{LDST\_2lanes} = \frac{U_{LDST}}{U_{ALU}} 50;\ T'_{2lanes} = \frac{U_{ALU}}{50} T \\
&\text{else} \\
&\quad U'_{ALU\_2lanes} = \frac{U_{ALU}}{U_{LDST}} 50;\ U'_{LDST\_2lanes} = 50;\ T'_{2lanes} = \frac{U_{LDST}}{50} T
\end{aligned}
\tag{9.1}
$$

The maximum utilization achievable is 50% when two lanes are deactivated. The equation agrees with the measurements shown in Tables 9.2 to 9.4. U'_{ALU_1lane}, U'_{LDST_1lane} and T'_{1lane} with one active lane can be derived using a similar approach and changing the threshold to 25%. A fused benchmark can be considered as a new one with new native runtime and utilizations, as shown in Table 9.3. The runtime scheduler will use the utilization information to choose the optimal number of active lanes based on the scheduling policy. The scheduler will be discussed after the introduction of the power model.

9.5 The Power Model

A highly accurate VP power consumption model is needed for optimization purposes. To obtain the needed data, simulations are performed based on the fully placed and routed VP implementation. By combining the VP's Native Circuit Description (NCD) with the testbenches of different scenarios, the detailed Switching Activity Information File (SAIF) for various VP utilizations can be obtained. The NCD and SAIF file are fed to the Xilinx Power Analyzer (XPA) tool to calculate exact dynamic and static power. By using testbenches that issue instructions to the VP at various rates, the VP's static and dynamic power under various utilizations can be measured. Figure 9.4 shows dynamic power results.

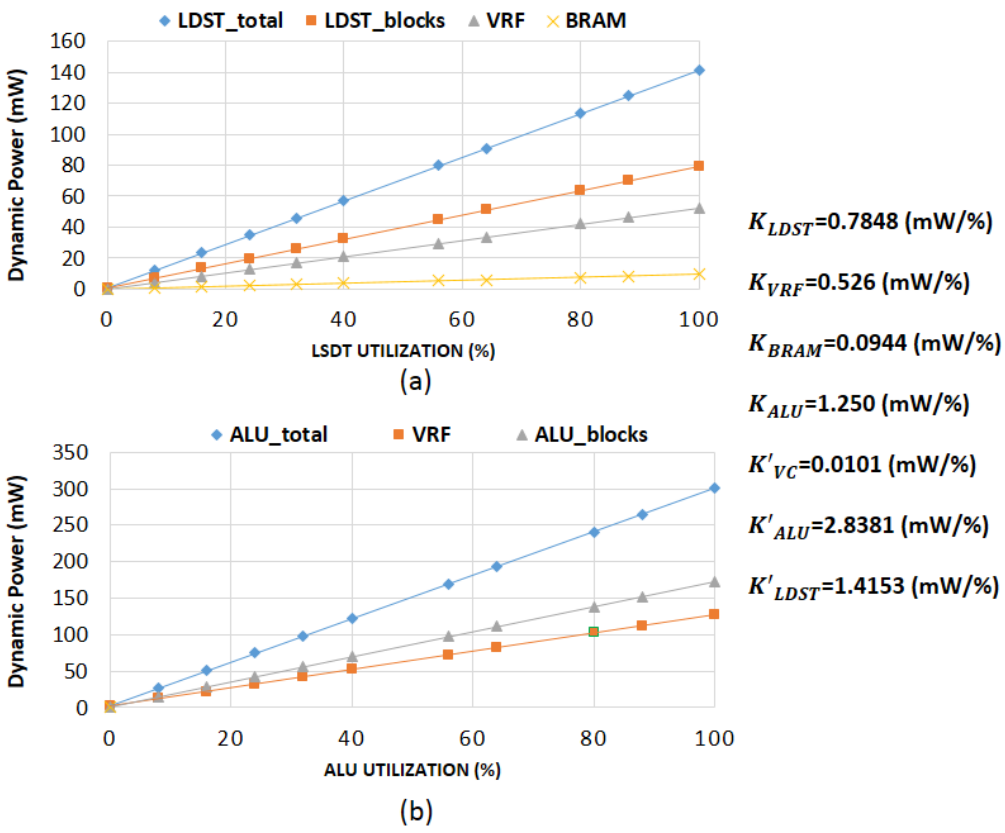


Figure 9.4 Dynamic power vs. utilization for both ALU and LDST data paths.

All VP lanes' dynamic power can be broken down into four components corresponding to the: VRF, VM banks, LDST data path (including LDST FIFO and decoder, address generator, and write back unit) and ALU data path (including ALU FIFO and decoder, execution and write back units). Each component's dynamic power is linear to its utilization, and is therefore related to U_{ALU} and U_{LDST}. Each LDST operation involves one memory access and one VRF access, and each ALU operation involves reading two operands from the VRF and writing one result back to the VRF. Therefore, the relation between VP lanes' dynamic power and their utilizations can be described by Equation 9.2. Each coefficient K is the power per utilization in mW/% for each corresponding component. On the other hand, VC is a common block that processes both ALU and LDST instructions, and therefore its power consumption is linear to the total issue rate (IR) of both types of instruction, and that can be described by Equation 9.3.

$$\begin{aligned} P_{ALU\_dynamic} &= K_{ALU}U_{ALU} + K_{VRF}(3*U_{ALU}) \\ P_{LDST\_dynamic} &= K_{LDST}U_{LDST} + K_{VRF}U_{LDST} + K_{BRAM}U_{LDST} \end{aligned} \tag{9.2}$$

$$P_{VC} = K_{VC} * IR = K_{VC} * (\frac{(U_{ALU} + U_{LDST}) * 4}{VL}) = K'_{VC}(U_{ALU} + U_{LDST}) \tag{9.3}$$

By adding together the terms in Equations 9.2 and 9.3, Equation 9.4 can be obtained which represents the VP's dynamic power as a simple linear function of U_{ALU} and U_{LDST}. This power model matches the measurements of the VP's dynamic power vs. U_{LDST} with idle ALU (Figure 9.4a), and power vs. U_{ALU} with idle LDST (Figure 9.4b). From the measured data, the coefficient K for each component can be extracted; the two most

important coefficients of interest are for the ALU and LDST units: K'_{ALU} = 2.838mW/%, and K'_{LDST} = 1.415mW/%.

$$\begin{aligned} P_{VP\_dynamic} &= K_{LDST}U_{LDST} + K_{ALU}U_{ALU} + K_{VRF}(3*U_{ALU} + U_{LDST}) \\ &\quad + K_{BRAM}U_{LDST} + K'_{VC}(U_{ALU} + U_{LDST}) \\ &= K'_{ALU}U_{ALU} + K'_{LDST}U_{LDST} \end{aligned} \tag{9.4}$$

The VP's total power is given by Equation 9.5. The measured VC static power is 2.2mW, and each lane's static power is 26.5mW with its dedicated memory bank. Since the FPGA does not support power gating, it is implemented using extra logic to isolate the power signal. Power gated components still dissipate about 15% of their original static power [Roy et al., 2009]. Using this assumption and the measured data, P_{static} under different VP configurations can be calculated. P_{static} is 108.2mW, 63.15mW and 40.63mW for the 4, 2, and 1 lane configuration, respectively. As shown in Equation 9.5, the VP's total power can be represented by U_{ALU} and U_{LDST}, which are the applications' actual utilizations under various situations. Combining Equation 9.5 with Equation 9.1, Equation 9.6 can be obtained that describes the relation of an application's power consumption under two active lanes configuration and its native utilizations. A similar equation can be derived for one active lane configuration.

$$
\begin{aligned}
&\text{if } (U_{ALU} < 50 \text{ and } U_{LDST} < 50) \text{ then} \\
&\quad P_{total\_2lanes} = 63.15 + K'_{ALU} U_{ALU} + K'_{LDST} U_{LDST} \\
&\text{elseif} (U_{ALU} > U_{LDST}) \text{ then} \\
&\quad P_{total\_2lanes} = 63.15 + 50K'_{ALU} + 50\frac{U_{LDST}}{U_{ALU}} K'_{LDST} \\
&\text{else} \\
&\quad P_{total\_2lanes} = 63.15 + 50K'_{LDST} + 50\frac{U_{ALU}}{U_{LDST}} K'_{ALU}
\end{aligned}
\tag{9.6}
$$

9.6 The Scheduling Policy

A vector application's power consumption under various VP configurations, P_{4lanes}, P_{2lanes} and P_{1lane} as a function of its native utilizations can be obtained using the equations described above in Section 9.5. The execution times T_{2lanes} and T_{1lane} are also related to T_{4lanes}, and the example for T_{2lanes} as a function of T_{4lanes} is shown in Equation 9.1. The set of *P* and *T* values form two-dimensional matrices with U_{ALU} and U_{LDST} as indexes. Two different scheduling policies are proposed in this section using *P* and *T*. The first policy is to achieve minimum energy consumption. The energy matrix for each configuration can be calculated by $E_{Nlanes} = P_{Nlanes} * T_{Nlanes}$. By comparing E_{4lanes}, E_{2lanes} and E_{1lane}, it can be determined the utilization boundary for different optimal configuration. Figure 9.5a shows a generic contour for minimum energy consumption; the actual values depend on the application. All applications whose native utilizations fall into region A consume minimum energy when executed with one active lane, while region B is for two lanes and region C is for four lanes. Using a similar approach, the boundary for the second scheduling policy can be obtained, which minimizes the product of an application's execution time and energy consumption; it is shown in Figure 9.5b.

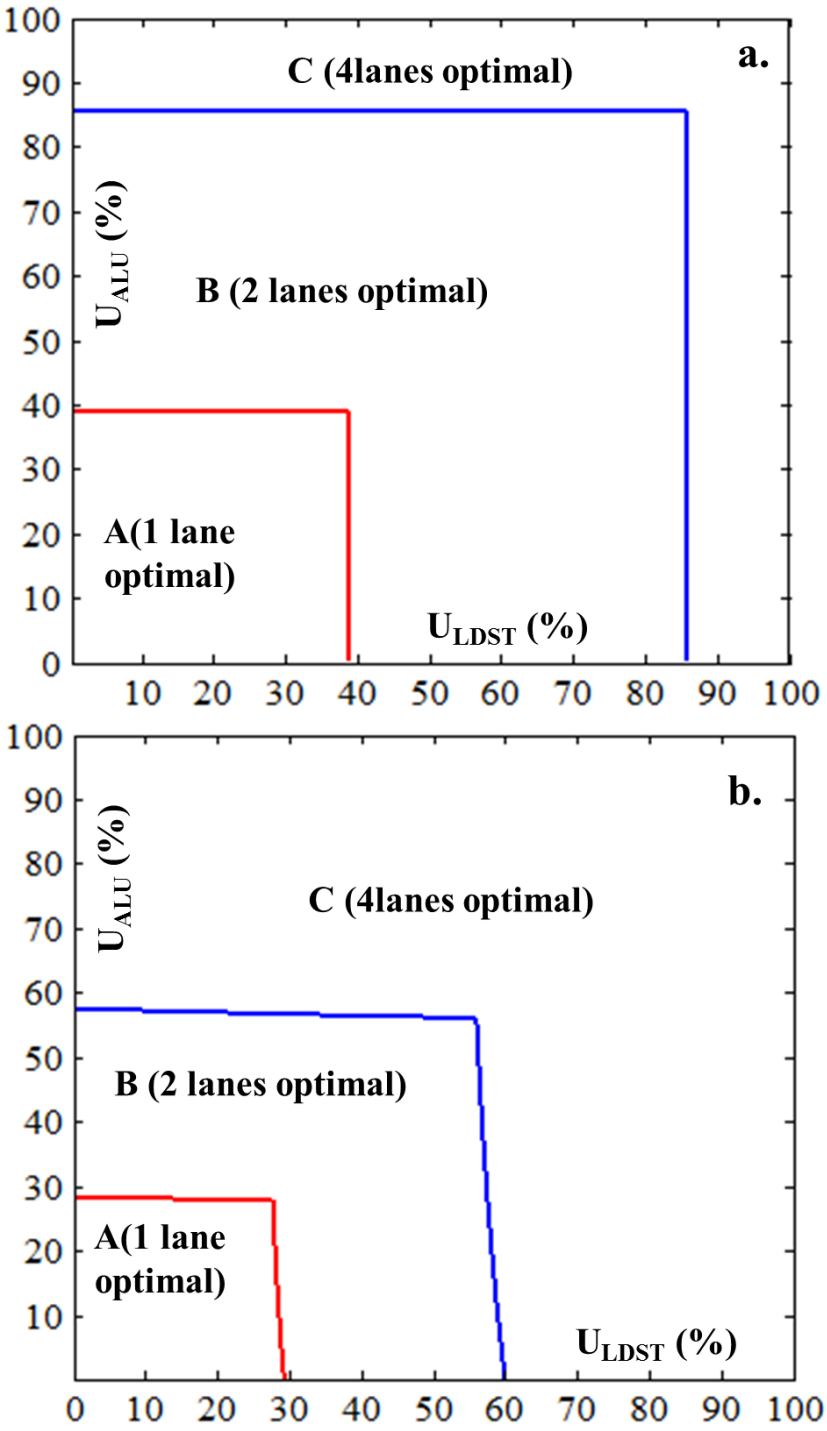


Figure 9.5 Optimal utilization boundaries for **a.** minimum energy **b.** minimum energy-execution time product.

The two scheduling policies (E_{min} and ET_{min}) are tested using an open system model where tasks that arrive within a time slice of size 10ms are scheduled in the following 10ms slice. The arrival of every task follows the Poisson distribution; six arrival rates λ = 1, 3, 5, 7, 9, 11 are tested. Tasks in the queue are ordered by their task type; since similar tasks are adjacent in the queue, the scheduler can easily identify fusable tasks. The tasks are the benchmarks introduced Section 9.4. For each optimization policy, every task has two optimal execution configurations: unfused and fused modes. All optimal configurations can be obtained by combining each task's U_{ALU} and U_{LDST} with the results shown in Figure 9.5. As mentioned previously at the end of Section 9.4, the scheduler will treat a fused task as a new task with its own U_{ALU} and U_{LDST}. The task queues for 1000 time slices are generated using the MATLAB random number generator, and the average parameters are calculated for the two scheduling policies and also for the VP without the proposed techniques. As shown in Figure 9.6, for the E_{min} policy, the proposed techniques reduce the average energy consumption by up to 33.8% while improving the runtime by 40%. The ET_{min} policy reduces the product of energy and runtime by up to 62.7%. For the VP without fusion and lane configuration, the average execution time at λ = 11 is close to 10ms and the system is about to overflow.

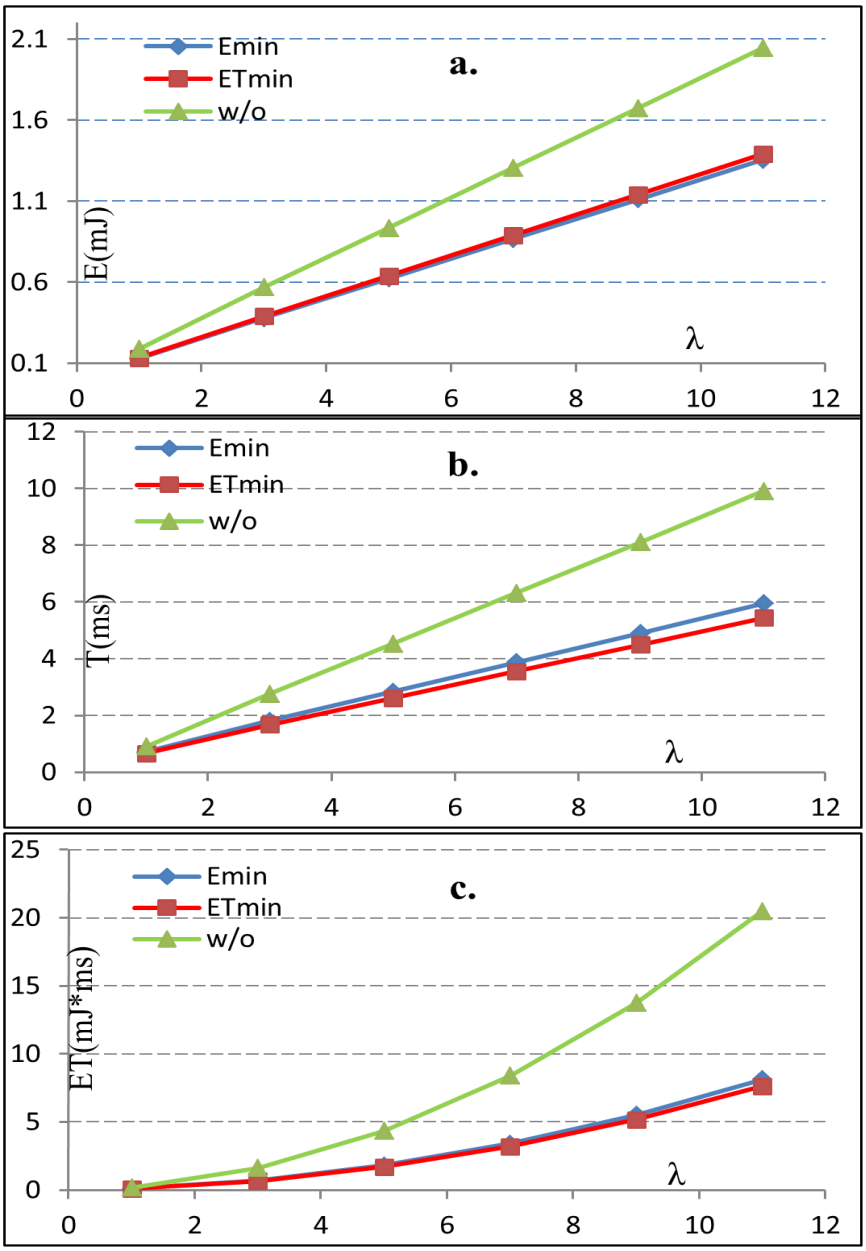


Figure 9.6 Comparison of the E_{min}, ET_{min} policies against a VP w/o fusion and lane configuration over the average of 1000 time slices. **a**. energy **b.** runtime **c.** energy-runtime product.

CHAPTER 10

A PIPELINED INTER-LANE NETWORK FOR EFFICIENT DATA SHUFFLING

The VPs introduced in this work all have a lane-dedicated memory bank architecture. This architecture has the benefit of higher throughput due to the elimination of potential stalls introduced by the access arbitrations in the memory crossbar. However, inter-lane data communication is critical for some applications such as FFT, matrix transpose, etc. In this chapter, a pipelined network is proposed with a novel lane decoder virtualization technique. The pipelined network is capable of performing stall-free high throughput inter-lane data shuffle operations with certain shuffle pattern limitation. The decoder virtualization technique reorders the data access sequences in each VP lane and alleviates network limitations.

10.1 The Benefits of the Network

In [Rooholamin and Ziavras, 2015], a data shuffle engine with a separate data path for the ALU unit was introduced to facilitate inter-lane data communications. However, the shuffle engine has very low throughput due to its non-pipelined nature. In addition, the engine utilizes the second port of each VM bank and, therefore, can potentially compete with host-to-VM data transactions. It is also very difficult to synchronize the data shuffles with the rest of the vector application due to significant speed mismatches, and in an SMT VP only one thread may claim ownership of the shuffle engine at any given time for ensuring proper execution.

The data shuffle network proposed in this section is improved using a novel decoder virtualization technique. With the resulting optimization, the shuffle network can

perform pipelined arbitrary data shuffles within a vector. The proposed design has the following benefits; i) Each node within the network has a constant number of fan-in and fan-out, and therefore the implementable design clock frequency is constant regardless of the size of the network (related to the number of VP lanes). ii) The network is fully pipelined and never stalls, thus yielding a throughput as high as the rest of the VP components. iii) The shuffle instruction works with the VRF and is incorporated into the ALU data path of the VP, and therefore data dependencies can be detected easily using the VP's existing hazard detection mechanism. iv) The shuffle instruction is considered a regular VP ALU instruction, thus the shuffle network works perfectly in an SMT environment. The shuffle network is prototyped here for the SMT VP with four lanes, thus supporting up to four simultaneous threads.

10.2 Structure of the Shuffle Network

To help better understand the shuffle network, the basic knowledge of the target VP model should be reviewed. The VP implementation consists of four VP lanes each having its dedicated ALU and LDST units, VRF, and VM module. The ALU and LDST units have separate data paths and only stall for each other when dependencies are detected. Both the ALU and LDST units have dedicated VRF ports for reading and writing, and the ALU unit does not access the VM due to the LDST architecture of the VP. A vector instruction of vector VL=16 will have all its elements interleaved in all four lanes, with elements 0, 4, 8, and 12 in lane 0. While all four lanes can simultaneously process data elements, multiple elements within each lane are processed sequentially. The decoder unit in each lane increments a counter based on the VL, and uses the value as an offset to physically locate the register and issue the operations to the ALU/LDST unit in ascending order. For

example, as shown in Figure 10.1, elements 0, 4, 8, and 12 within the array have the local order 0, 1, 2, and 3, respectively, inside lane 0.

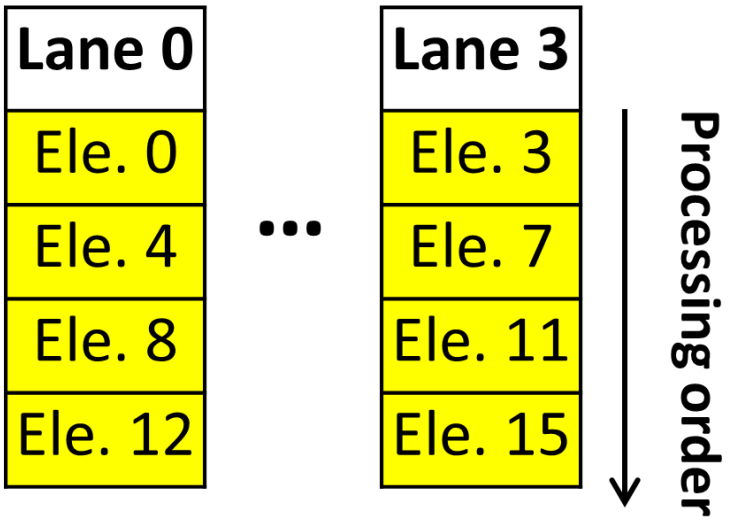


Figure 10.1 Execution order of elements within an array of VL=16.

The structure of the proposed pipelined shuffle network is shown in Figure 10.2. It only illustrates the case of four VP lanes and VM modules. However, the network can be easily expanded to accommodate more lanes and VM modules. As the network grows, the complexity of each node will not increase, and therefore the same clock frequency can be achieved. Although the pipeline will become deeper, the throughput will also increase accordingly. As long as the data shuffle pattern meets certain limitations, the network can always achieve a throughput of N elements per clock cycle, where N is the number of VP lanes. The pattern limitations will be introduced within the next two paragraphs, along with more details of the network structure. The shuffle pattern limit can be removed by applying the decoder virtualization technique, which will be covered in Section 10.3.

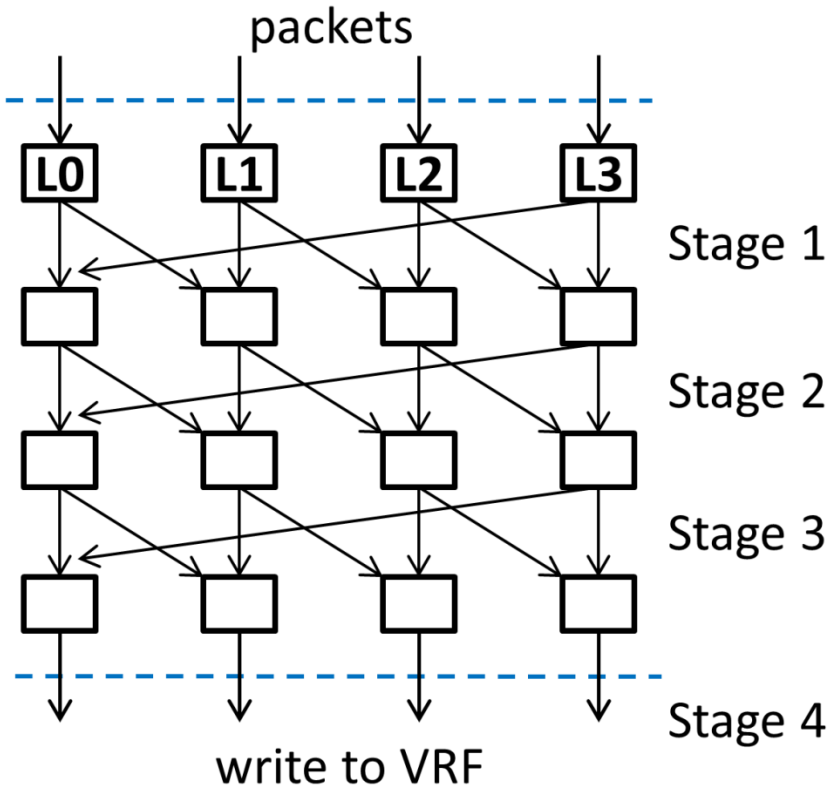


Figure 10.2 The ring-based structure of the data shuffle network.

The network servicing N VP lanes consists of N-1 pipeline stages for switching and one stage for writing results to the VRF. It uses the ALU data path and performs VRF to VRF shuffling. The shuffle instruction takes three vector registers that represent the sources, indices/offsets and destinations. $\mathbf{R_S}$ contains the source data array to be shuffled, and register $\mathbf{R_T}$ contains the shuffle indices of elements in the source register $\mathbf{R_S}$. $\mathbf{R_D}$ is the destination register that will receive the source data after they have been re-ordered as per the indices in $\mathbf{R_T}$. Due to the potentially out-of-order vector element access nature of the shuffle instruction, $\mathbf{R_T}$, $\mathbf{R_S}$, and $\mathbf{R_D}$ must be different registers or the behavior will be undefined. The source data and shuffle offsets enter the network in the first stage, and travel through the network together as a packet. The first two bits of the offset indicate the

destination lane of the packet and are used by the network to perform inter-lane shuffling. When a packet eventually reaches its destination lane, the rest of the bits in the offset are used to write the data into the correct local VRF address.

As shown in Figure 10.2, each packet can only travel to the neighboring lane in each pipeline stage (packet in lane 3 can travel to lane 0, thus the ring structure). Each node checks the header of a packet and passes it on when the packet has not reached its destination lane. When a packet reaches its destination lane, it will bypass future switching and stay within its destination lane; therefore, it is also possible for a node to pass a packet within the same lane, only one stage later. Since there are only four VM banks, a maximum of three switches are needed to send a packet. The network is completely pipelined and can accept four packets per clock cycle. The data shuffle pattern limitation is defined as follows: any four packets that enter the network in the same clock cycle must be destined to VRF locations that belong to four different lanes. As long as this requirement is met, it is easy to see that at any given point there can be a maximum of two packets at a node, one of which is waiting to be switched and the other is in the bypass state (i.e., in the destination lane and needs no more switching). Therefore, in addition to the switching packet buffer, each node is also designed with a bypass packet buffer for storing packets that can potentially reach their destination lanes before the final switching stage. With the pattern limitation, no congestion or buffer overflow could occur within the switching network. At the output end of the network, the writing back to VRF also will not be stalled due to arbitration, since each lane will have only one data element to be written in any given clock cycle.

10.3 The Decoder Virtualization Technique

As described in Section 10.2, to prevent congestion in the switching network and arbitration in the write back stage, a certain requirement must be met. An example data shuffle pattern is shown in this section that does not explicitly meet this requirement; it is used to illustrate how the novel decoder virtualization technique removes the requirement through the reordering of data accesses. A 4*4 matrix can be stored in a vector of VL=16. The transpose operation of the matrix is a good shuffle example that does not meet the requirement described in Section 10.2. Figure 10.3a shows the destination lane of each element. Based on the normal operation issue order of the decoder, the shuffle operation would cause congestion in the network since multiple elements that enter the network in the same clock cycle try to reach the same destination lane.

To make the network capable of performing the transpose operation without any stalls, a decoder virtualization technique is proposed that solves the problem in the following way: the network requires that all four elements entering it within the same clock cycle have different destination lanes. While it is not true for each clock cycle in the example data shuffle pattern, the overall destinations of all the elements within the array are evenly distributed across all four lanes. Smart operation reordering would enable the network to shuffle any arbitrary data pattern while still maintaining the non-congestion requirement physically. A reorder lookup table (RLT) is implemented within each decoder unit, and the RLT can be programmed by a special vector instruction. Instead of directly issuing operations using its local counter, each decoder unit can now use the counter value as a virtual order, and use it to index into the RLT and get the real offset of the operation to be issued. Using the RLT setup shown in Figure 10.3b, the decoder unit in each lane would

issue operations in different orders and the matrix transpose operation will have the real operation order shown in Figure 10.3c. The new operation order complies with the network's non-congestion requirement.

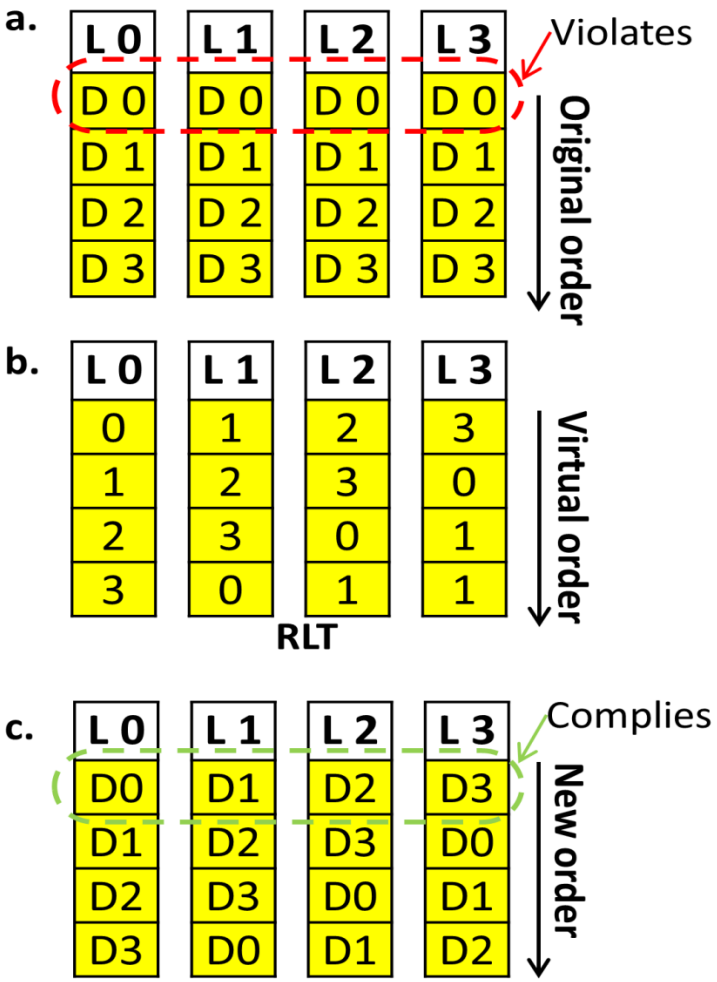


Figure 10.3 Execution order and element destinations of a 4*4 matrix transpose. **a.** The original order. **b.** The RLT setup. **c.** The new order after RLT order translation.

Using the decoder virtualization introduced above, the network is capable of shuffling any data pattern without any stalls, thus achieving throughput of four transactions per clock cycle. For a VP with more than four lanes, the network can be easily expanded

without increasing the complexity of individual nodes, and the throughput scales linearly. The RLT's resource consumption is very minor. 16 entries per lane are needed, and each entry only requires four bits, since the maximum VL supported by the current VP implementation is 64 and there are at most 16 elements per lane. The RLT reorder pattern can be generated by the programmer/compiler at static time, and therefore the shuffle operation is extremely energy efficient compared to memory-memory data shuffles. In an SMT VP, a per thread RLT can be implemented to give SMT support for the shuffle instruction.

CHAPTER 11

CONCLUSIONS AND FUTURE WORK

11.1 Conclusions

This dissertation mainly presents two techniques, namely instruction fusion and VP virtualization, to efficiently exploit DLP using either general-purpose multicore and multi-scalar processors or specialized SIMD processors. The two techniques are also combined to further improve performance. In addition, power analysis is carried out to optimize the energy efficiency of DLP applications based on these techniques. The primary motivation of this work is to apply innovative optimization techniques to improve the exploitation of DLP in terms of performance, energy efficiency and resource utilization.

The instruction fusion technique is first introduced. It utilizes the similarity between instruction blocs within an unrolled loop to enable energy efficient SIMD execution on a general-purpose multicore or multi-threaded processor. To evaluate the technique, a MIPS-like multiscalar processor with fused mode is prototyped on an FPGA, and benchmarking is performed to compare the performance and energy consumption of fused execution with that of normal execution. Compared to normal execution, the dynamic energy consumption of unrolled loops is reduced by up to 11.9% when executed under the fused mode. The power and performance of non-vector code execution is hardly affected due to the processor's hardware modification to support instruction fusion. Moreover, the code size for unrolled loops is reduced by up to 45% through instruction fusion, which is of tremendous value for embedded systems and FPGAs that have limited on-chip memories. Furthermore, the reduced code size can potentially improve performance due to higher instruction cache hit rates.

The VP virtualization technique is then introduced. The technique is proposed to improve SMT execution of the VP. The most important component of the virtualization technique is the software VRF management algorithm, which dynamically maps the virtual VRF space of each simultaneous thread to the available physical VRF space. The virtualization technique significantly improves programmer productivity by eliminating the need to solve VRF name conflicts statically, which is complicated and sometimes prohibitive. The algorithm also enables sharing of the VP between threads with different VLs. The register fragmentation effect of the algorithm is evaluated using random experiments and it turns out that its impact is negligible. The performance overhead of the algorithm is also measured and is also minor compared to the execution time of an average vector application.

To evaluate the virtualization technique, an SMT VP interfaced with a subsystem having five RISC processors as hosts is prototyped on an FPGA. The VP supports efficient SMT through the usage of a per instruction thread ID and VL mechanism. The hosts subsystem interfaces the SMT VP via a FIFO and arbitrator interface. The FIFO removes potential VP clock cycle wastage due to the arbitration among multiple host processors. The round-robin arbitrator, together with the VP's per instruction thread ID mechanism, facilitates fine-grain SMT where instructions from multiple threads can enter the VP every other clock cycle and can coexist within the pipeline.

Benchmarking for the prototype is performed using both homogeneous and heterogeneous threads. Each benchmark's utilization is characterized through single threaded execution. The VP's utilization saturation effect is observed in the homogeneous execution of multiple copies of certain high utilization benchmarks. Under homogeneous

execution, the throughput and VP utilization can be multiplied by the number of simultaneous threads as long as the aggregate utilization does not reach the VP's saturation level. Using the utilization characteristics of the benchmarks, a throughput-driven scheduling algorithm is proposed for the SMT VP. The scheduler picks the threads within the thread queue in proper order to achieve best average VP utilization. Under dynamic thread creation of different benchmarks with diverse VLs, the scheduler improves the SMT VP's performance by up to 333% compared to a single threaded VP. Power measurement and projection also show that, with proper power gating, compared to a similar-sized single-threaded VP, the SMT VP consumes 37% less energy in addition to boosting the performance significantly.

Eventually, the idea of combining the VP virtualization technique with the instruction fusion technique is proposed. A new VP prototype is implemented which is fully virtualized to support thread level vector instruction fusion. Thread fusion can be triggered by the scheduler once similar tasks are identified in the pending tasks queue. Under fused execution, the effective vector instruction issue rate from the host processor is multiplied, and therefore performance is improved and the host processor's energy consumption is also significantly reduced. The new VP prototype is also capable of dynamically deactivating some of its lanes to minimize static power. An accurate power model for the new VP prototype is derived and two optimization policies are proposed to optimize the VP energy configuration based on a given application. With the virtualization and instruction fusion techniques, the minimum energy policy reduces VP energy consumption by up to 33.8% and improves the runtime by 40%. The other policy reduces the product of energy and runtime by up to 62.7%.

At the end of the dissertation, a pipelined inter-lane data shuffle network is proposed to overcome the most significant insufficiency of the VP used throughout this work. Combined with a novel decoder virtualization technique, the network is capable of performing unconstrained data shuffles within a vector register using the VP's ALU data path. The shuffle instruction is a normal vector instruction and therefore data dependencies can be easily detected.

11.2 Future Work

There are a few possible architectural improvements that can be applied to further increase the efficiency and flexibility of the SMT VP framework proposed in this dissertation.

Hardware implementation of the software VRF virtualization algorithm. The software-based VRF management algorithm can be implemented using hardwired logic, and therefore become extremely fast and energy efficient. With the negligible overhead of the hardwired VRF management, finer-grain SMT could be achieved where vector applications could be broken down into lighter threads so that the scheduler can have a longer queue in order to achieve near optimal utilization.

Sharing of the SMT VP across multiple multicore processor groups. With a carefully designed sharing switch, each SMT VP can be shared among multiple processor groups. On top of the high utilization already achieved through SMT, the energy efficiency of the new system could be further improved by merging the workload of multiple processor groups targeting one SMT VP, thus power gating unused VPs. An example of such architecture is shown in Figure 11.1. In this system, each VP functions as multiple LVPs and any host processor can trigger thread fusion using multiple LVPs. A scheduler

can be used to optimize the overall system behavior (i.e., determine the number of LVPs to be used and how many VPs need to stay active based on the system workload).

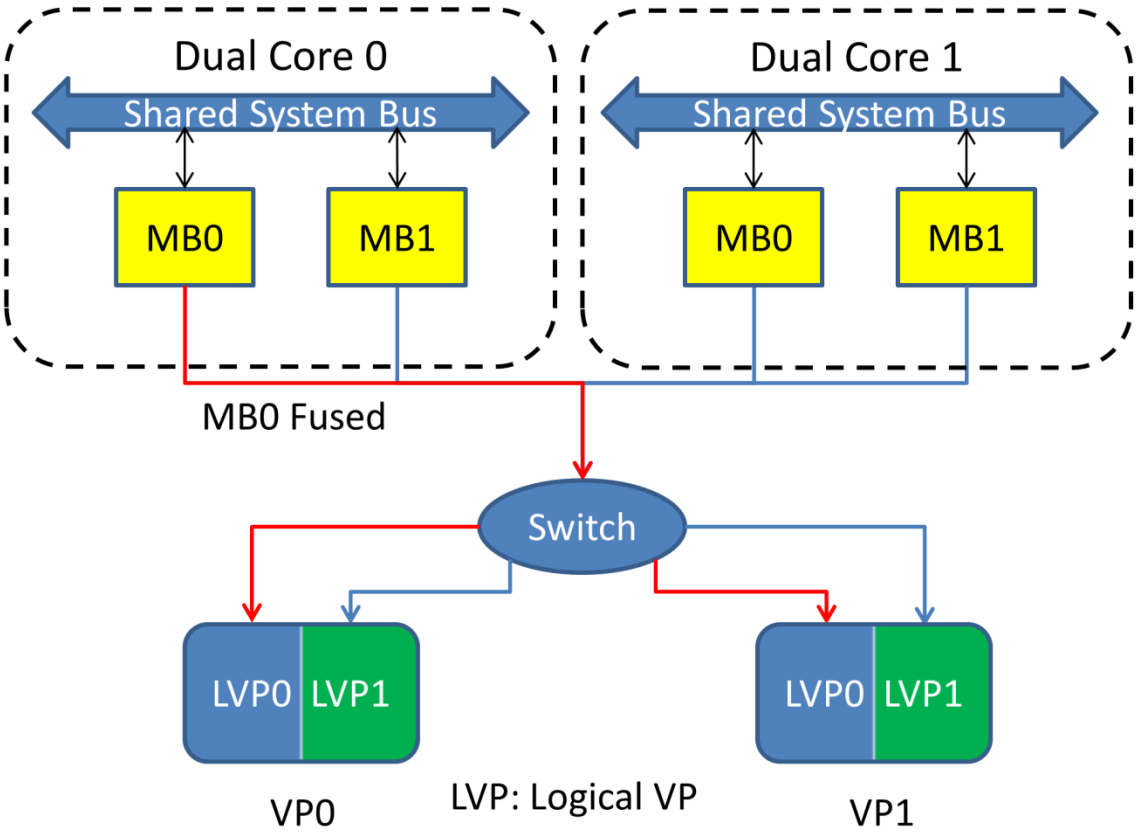


Figure 11.1 An example of multiple SMT VPs shared across two processor groups.

Programmable vector instruction arbitrator. The vector instruction arbitrator could be modified to be programmable by the host processors. Host processors could modify the arbitration policy (by modifying some state registers in the state machine of the arbitrator) to support priority based scheduling. Vector applications with higher priority could then occupy a larger portion of the VP execution time and therefore complete sooner.

REFERENCES

Ansari, B., & Hasan, M. A. (2008). High-performance architecture of elliptic curve scalar multiplication. *IEEE Transactions on Computers*, 57(11), pp. 1443-1453.

Beldianu, S. F., & Ziavras, S. G. (2013). Multicore-based vector coprocessor sharing for performance and energy gains. *ACM Transactions on Embedded Computing Systems*, 13(2).

Beldianu, S. F., & Ziavras, S. G. (2015, March). Performance-energy optimizations for shared vector accelerators in multicores. *IEEE Transactions on Computers*, 64(3), pp. 805-817.

Cho, J., Chang, H., & Sung, W. (2006, May). An FPGA based SIMD processor with a vector memory unit. *IEEE International Symposium on Circuits and Systems*, pp. 525-528.

Chou, C. H., Severance, A., Brant, A. D., Liu, Z., Sant, S., & Lemieux, G. G. (2011, February). VEGAS: soft vector processor with scratchpad memory. *19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 15-24.

Dally, W. J., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R. C., Parikh, V., & Sheffield, D. (2008). Efficient embedded computing. *Computer*, (7), pp. 27-32.

Dennard, R. H., Rideout, V. L., Bassous, E., & LeBlanc, A. R. (1974). Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), pp.256-268.

Goulding-Hotta, N., Sampson, J., Zheng, Q., Bhatt, V., Auricchio, J., Swanson, S., & Taylor, M. B. (2012, January). Greendroid: An architecture for the dark silicon age. *17th IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 100-105.

Heil, T., Krishna, A., Lindberg, N., Toussi, F., & Vanderwiel, S. (2014). Architecture and performance of the hardware accelerators in IBM's PowerEN processor. *ACM Transactions on Parallel Computing*, 1(1).

Iranpour, A. R., & Kuchcinski, K. (2004, August). Evaluation of SIMD architecture enhancement in embedded processors for MPEG-4. *IEEE Euromicro Symposium on Digital System Design*, pp. 262-269.

Kim, Y. H., Yoo, J. W., Lee, S. W., Paik, J., & Choi, B. (2005, January). Optimization of H. 264 encoder using adaptive mode decision and SIMD instructions. *IEEE International Conference on Consumer Electronics, Digest of Technical Papers*, pp. 289-290.

Kozyrakis, C. E., & Patterson, D. A. (2003). Scalable, vector processors for embedded systems. *IEEE Micro*, 23(6), pp. 36-45.

Lee, J., Jeon, G., Park, S., Jung, T., & Jeong, J. (2008, June). SIMD Optimization of the H. 264/SVC decoder with efficient data structure. *IEEE International Conference on Multimedia and Expo*, pp. 69-72.

Lee, J., Moon, S., & Sung, W. (2004, December). H. 264 decoder optimization exploiting SIMD instructions. *IEEE Asia-Pacific Conference on Circuits and Systems*, 2, pp. 1149-1152.

Lee, Y., Avizienis, R., Bishara, A., Xia, R., Lockhart, D., Batten, C., & Asanović, K. (2013). Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *ACM Transactions on Computer Systems*, 31(3).

Lin, Y., Lee, H., Woh, M., Harel, Y., Mahlke, S., Mudge, T., & Flautner, K. (2006, June). SODA: a low-power architecture for software radio. *ACM SIGARCH Computer Architecture News*, 34(2), pp. 89-101.

Lo, W. Y., Lun, D. P., Siu, W. C., Wang, W., & Song, J. (2011). Improved SIMD architecture for high performance video processors. *IEEE Transactions on Circuits and Systems for Video Technology*, 21(12), pp. 1769-1783.

Marr, D. T., F. Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A. & Upton, M. (2002, Feb.). Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(2), pp. 1–12.

Nvidia Corp. (2014). Featuring Maxwell, the most advanced GPU ever made. *Nvidia Gefore GTX 980 White Paper.*

Pajuelo, A., González, A., & Valero, M. (2002). Speculative dynamic vectorization. *29th IEEE Annual International Symposium on Computer Architecture.* pp. 271-280.

Rakvic, R., González, J., Cai, Q., Chaparro, P., Magklis, G & A. Gonzalez, A. (2010) Energy efficiency via thread fusion and value reuse. *IET Computers and Digital Techn*iques. 4(2), pp.114-125.

Rooholamin. SA, and Ziavras. S.G. (2015, June). Modular vector processor architecture targeting at data-level parallelism, *Microprocessors and Microsystems.* 39(4), pp. 237-249.

Roy, S., Ranganathan, N., & Katkoori, S. (2009, November). A framework for power-gating functional units in embedded microprocessors. *IEEE Transactions on VLSI Systems*, 17(11), pp. 1640-1649.

Severance, A, & Lemieux, G. (2012). VENICE: A compact vector processor for FPGA applications. *IEEE International Conference on Field-Programmable Technology*, pp. 261-268.

Severance, A., Edwards, J., Omidian, H., & Lemieux, G. (2014, February). Soft vector processors with streaming pipelines. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 117-126.

Shengfa, Y., Zhenping, C., & Zhaowen, Z. (2006, June). Instruction-level optimization of H. 264 encoder using SIMD instructions. *Proceedings of IEEE International Conference on Communications, Circuits and Systems*, 1, pp. 126-129.

Sung, W., & Mitra, S. K. (1987). Implementation of digital filtering algorithms using pipelined vector processors. *Proceedings of the IEEE*, 75(9), pp. 1293-1303.

Taylor, M. B. (2012, June). Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. *Proceedings of 49th ACM Annual Design Automation Conference* (pp. 1131-1136).

Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., & Taylor, M. B. (2010, March). Conservation cores: reducing the energy of mature computations. *ACM SIGARCH Computer Architecture News.* 38(1), pp. 205-218.

Wang, S., Li, X. S., Xia, J., Situ, Y., & De Hoop, M. V. (2013). Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures. *SIAM Journal on Scientific Computing*, 35(6), pp. 519-544.

Wang, X., Ziavras, S. G., Nwankpa, C., Johnson, J., & Nagvajara, P. (2007). Parallel solution of Newton's power flow equations on configurable chips. *International Journal of Electrical Power & Energy Systems*, 29(5), pp. 422-431.

Xilinx INC. (2010). MicroBlaze Processor Reference Guide, http://www.xilinx.com/support/ documentation/sw_manuals/mb_ref_guide.pdf (accessed on March 10, 2016).

Xilinx INC. (2012). AXI Reference Guide. http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf (accessed on March 10, 2016).

Xilinx INC. (2011). Power Methodology Guide., http://www.xilinx.com/support/ documentation/sw_manuals/xilinx13_1/ug786_PowerMethodology.pdf (accessed on March 10, 2016).

Yang, H., & Ziavras, S. G. (2005, September). FPGA-based vector processor for algebraic equation solvers. *IEEE International System on Chip Conference*, pp. 115-116.

Yiannacouras, P., Steffan, J. G., & Rose, J. (2008, October). VESPA: portable, scalable, and flexible FPGA-based vector processors. *ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 61-70.

Yu, J., Lemieux, G., & Eagleston, C. (2008, February). Vector processing as a soft-core CPU accelerator. *Proceedings of the 16th ACM international ACM/SIGDA symposium on Field programmable gate arrays.* pp. 222-232.