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

ABSTRACT

VECTOR PROCESSOR VIRTUALIZATION: DISTRIBUTED MEMORY HIERARCHY AND SIMULTANEOUS MULTITHREADING

by
SeyedAmin Rooholamin

Taking advantage of DLP (Data-Level Parallelism) is indispensable in most data streaming and multimedia applications. Several architectures have been proposed to improve both the performance and energy consumption for such applications. Superscalar and VLIW (Very Long Instruction Word) processors, along with SIMD (Single-Instruction Multiple-Data) and vector processor (VP) accelerators, are among the available options for designers to accomplish their desired requirements. On the other hand, these choices turn out to be large resource and energy consumers, while also not being always used efficiently due to data dependencies among instructions and limited portion of vectorizable code in single applications that deploy them. This dissertation proposes an innovative architecture for a multithreaded VP which separates the path for performing data shuffle and memory-indexed accesses from the data path for executing other vector instructions that access the memory. This separation speeds up the most common memory access operations by avoiding extra delays and unnecessary stalls. In this multilane-based VP design, each vector lane uses its own private memory to avoid any stalls during memory access instructions. More importantly, the proposed VP has an innovative multithreaded architecture which makes it highly suitable for concurrent sharing in multicore environments. To this end, the VP which is developed in VHDL and prototyped on an FPGA (Field-Programmable Gate Array), serves as a coprocessor for one or more scalar cores in various system architectures presented in the dissertation.

In the first system architecture, the VP is allocated exclusively to a single scalar core. Benchmarking shows that the VP can achieve very high performance. The inclusion of distributed data shuffle engines across vector lanes has a spectacular impact on the execution time, primarily for applications like FFT (Fast-Fourier Transform) that require large amounts of data shuffling.

In the second system architecture, a VP virtualization technique is presented which, when applied, enables the multithreaded VP to simultaneously execute many threads of various vector lengths. The threads compete simultaneously for the VP resources having as a goal an improved aggregate VP utilization. This approach yields high VP utilization even under low utilization for the individual threads. A vector register file (VRF) virtualization technique dynamically allocates physical vector registers to running threads. The technique is implemented for a multi-core processor embedded in an FPGA. Under the dynamic creation of threads, benchmarking demonstrates large VP speedups and drastic energy savings when compared to the first system architecture.

In the last system architecture, further improvements focus on VP virtualization relying exclusively on hardware. Moreover, a pipelined data shuffle network replaces the non-pipelined shuffle engines. The VP can then take advantage of identical instruction flows that may be present in different vector applications by running in a fused instruction mode that increases its utilization. A power dissipation model is introduced as well as two optimization policies towards minimizing the consumed energy, or the product of the energy and runtime for a given application. Benchmarking shows the positive impact of these optimizations.

VECTOR PROCESSOR VIRTUALIZATION: DISTRIBUTED MEMORY HIERARCHY AND SIMULTANEOUS MULTITHREADING

by
SeyedAmin Rooholamin

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

APPROVAL PAGE

VECTOR PROCESSOR VIRTUALIZATION: DISTRIBUTED MEMORY HIERARCHY AND SIMULTANEOUS MULTITHREADING

SeyedAmin Rooholamin

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. Sui-hoi E. 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 Gerbessiotis, Committee Member — Date
Associate Professor of Computer Science, NJIT

BIOGRAPHICAL SKETCH

Author: SeyedAmin Rooholamin

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,
  Isfahan University of Technology, Isfahan, Iran, 2008
- Bachelor of Science in Electrical Engineering,
  Isfahan University of Technology, Isfahan, Iran, 2005

Major: Electrical Engineering

Presentations and Publications:

Rooholamin, S.A., and Ziavras, S.G. (June 2015). Modular Vector Processor Architecture Targeting at Data-level Parallelism, *Microprocessors and Microsystems.* Elsevier, 39, 4, pp. 237-249.

Lu, Y, Rooholamin, S.A., and Ziavras, S.G. (March 2016). Vector Processor Virtualization for Simultaneous Multithreading, *ACM Transactions on Embedded Computing Systems.* Accepted for publication.

Lu, Y, Rooholamin, S.A., and Ziavras, S.G. (March 2016). Power-Performance Optimization of a Virtualized SMT Vector Processor via Thread Fusion and Lane Configuration. *IEEE Computer Society Annual Symposium on VLSI.* Accepted for publication.

To my Family, with Love and Gratitude

تقدیم به پدر و مادرم با عشق و سپاس

ACKNOWLEDGMENT

I would like to express my heartfelt gratitude and deepest appreciation to my adviser Dr. Sotirios G Ziavras for being my mentor through my PhD career. His wisdom, commitment, guidance and inspiration as well as his extraordinary skills and depth of knowledge guided and motivated me to find and pursue my research. His constructive criticism and trusted support made this work a great learning experience.

I would like to express my sincere gratitude to Dr. Durga Misra, Dr. Edwin Hou, Dr. Roberto Rojas-Cessa and Dr. Alexandros V. Gerbessiotis for serving as committee members. I appreciate their time as well as their encouraging and constructive comments, and feedback on the dissertation.

Moreover, I am truly indebted and thankful to the ECE Department at NJIT for the TA and fellowship support. This work would not have been possible without this support.

Further thanks go to the staff of the offices of international students and graduate studies, and the staff of the ECE Department for their advice, help and support with administrative matters during my PhD studies.

Additionally, I would like to thank my friends Yaojie, Behzad and Mina for all the great and unforgettable moments we shared together during these years.

Finally, it is my honor to express my deepest gratitude and respect to my family for being supportive and encouraging. I am grateful to my fantastic parents Nahid and Akbar who always give me their unconditional love and support. I would also like to thank my brother, Ehsan, who has always been there for me.

TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION	1
	1.1 Background History	1
	1.2 Motivations and Objectives	3
	1.2.1 Single Host System	3
	1.2.2 Multiple Hosts System	5
	1.2.3 Virtualized SMT VP	7
2	RELATED WORK	10
	2.1 Related Work in Vector Processing	10
	2.2 VP Sharing Techniques and Comparisons	12
3	PROPOSED VECTOR COPROCESSOR	16
	3.1 Single Host System Architecture	16
	3.1.1 VP Architecture and Instructions in the Single Host System	20
	3.1.2 Pipelined ALU and LDST Unit	25
	3.1.3 Resource Utilization in Single Host System	27
	3.2 Multiple Hosts System Architecture	28
	3.2.1 The System Core	30
	3.2.2 Application Cores	31
	3.2.3 Vector Instruction FIFOs and Arbitrator	32
	3.2.4 The System Bus	33
	3.3 VP Virtualization	34

TABLE OF CONTENTS
(Continued)

Chapter **Page**

3.3.1 The Vector Register File 34

3.3.2 The Vector Register Management Algorithm 35

3.3.3 Assigning/Releasing VRF Resources 38

3.3.4 VRF Fragmentation Issues 39

3.4 VP Architecture in the Multiple Hosts System 40

3.4.1 VP-MB Interface 42

3.4.2 Hazard Detection 42

3.4.3 Vector Lane Structure in the Multiple Hosts System 44

3.4.4 Resource Utilization in the Multiple Hosts System 45

4 BENCHMARKING 48

4.1 Benchmark Suite for the Single Host System 48

4.1.1 Vector Instruction in the Single Host System 48

4.1.2 Benchmark Applications for Single Host System 50

4.2 Benchmark Suite for the Multiple Hosts System 52

4.2.1 VP Instruction and Compilation for the Multiple Hosts System 53

4.2.2 Benchmark Applications for the Multiple Hosts System 55

5 PERFORMANCE ANALYSIS 57

5.1 Single Host System Performance Analysis 57

TABLE OF CONTENTS
(Continued)

Chapter **Page**

5.1.1 Simulation Results and Performance Analysis in the Single Host System 57

5.1.2 Performance Exploration in the Single Host System 65

5.1.3 Comparison with Prior Work..... 69

5.2 Multiple Hosts System Performance Analysis..... 71

5.2.1 Simulation Results..... 71

5.2.2 Comparison with the Single Host System and Prior Works..... 75

6 SCHEDULING VECTOR THREADS..... 77

6.1 Scheduling Algorithm Implemented on the System Core..... 77

6.2 Queues of Fixed Length 79

6.3 Open System with Randomly Arriving Threads 82

7 POWER ANALYSIS AND ENERGY CONSUMPTION 86

7.1 Power Analysis for the Single Host System..... 86

7.2 Power Analysis for the Multiple Hosts System..... 90

7.2.1 VP Dynamic Power for the Multiple Hosts System 91

7.2.2 Total Energy Consumption in the Multiple Hosts System 94

8 VIRTUALIZED SMT VP AND OPTIMIZATION VIA THREAD FUSION AND LANE CONFIGURATION 96

8.1 Virtualized VP Architecture..... 96

8.1.1 Increasing the VP Saturation Level 96

8.1.2 VP Pipelined Data Shuffle Network..... 97

TABLE OF CONTENTS
(Continued)

Chapter **Page**

8.1.3 Virtualized VM Address Space 101

8.1.4 Configurable Components 102

8.1.5 VRF and VM Virtualization Under SMT 104

8.1.6 Fusion of Similar Threads 106

8.2 System Architecture and FPGA Implementation 107

8.3 Benchmarking the Virtualized VP 109

8.4 Power Model 111

8.5 The Scheduling Policy 114

9 COCNCLUSION AND FUTURE WORK 118

9.1 Conclusion 118

9.2 Future Work 119

REFERENCES 122

LIST OF TABLES

Table **Page**

3.1 Resource Consumption for Single Host System. 28

3.2 Resource Consumption for Multiple Hosts System. 47

5.1 Performance Comparison for Three Multiplication Algorithms and Various VLs on the Single Host System. Algorithms 1 and 2 Use Both the VP and MB. Algorithm 3 Uses Only the VP. The Execution Time is Shown for Each Element Produced in the Product Matrix. (a) Without Compiler Optimization and (b) with Compiler Optimization. 58

5.2 Performance Comparison for FIR Filtering with Various Filter Sizes in the Single Host System. The Times for Data Exchanges Between the Global and Private Memories are Included. The Times are for Calculating All the Output Elements. (a) Without Compiler Optimization and (b) with Compiler Optimization. 59

5.3 Performance Comparison for FFT of Various Sizes in the Single Host System. The Execution Time Includes the Overhead of Writing and Reading Between the Global and Vector Memories. The Numbers are for Calculating the Results (a)Without Compiler Optimization and (b) with Compiler Optimization. 60

5.4 Performance Comparison for RGB2YIQ with Various VLs in the Single Host System. The Time is for Calculating a Block of 8*8 Pixels in YIQ Color Space. (a) Without Compiler Optimization and (b) with Compiler Optimization. 61

5.5 Matrix Multiplication Performance Comparison for Various VLs in the Single Host System. 67

5.6 FIR Performance Comparison for Various VLs in the Single Host System..... 68

5.7 FFT Performance Comparison for Various VLs in the Single Host System. 68

5.8 RGB2YIQ Performance Comparison for Various VLs in Single Host System. . 68

5.9 Speedups of the VP in Single Host System and the Design in [Beldianu et al., 2013] vs. the MB for VL=32. 70

5.10 Matrix Multiplication Performance in the Multiple Hosts System (Input and Output Matrix Size: VL*VL, 1 Iteration per Core). 72

5.11 FIR Performance in the Multiple Hosts System (Input Vector Size: VL, 1 Iteration per Core). 72

LIST OF TABLES
(Continued)

Table **Page**

5.12 VDP Performance in the Multiple Hosts System (Input Vector Size : VL, 1 Iteration per Core). 72

5.13 DCT Performance in the Multiple Hosts System (Input: VL/8 Blocks of Size 8*8, 1 Iteration per Core). 73

5.14 RGB2YIQ Performance in the Multiple Hosts System (Input: 1024 Pixels, 1 Iteration per Core). 73

5.15 Speedup Comparison With the Single Host and Previously Shared VP 76

6.1 Detailed Results for a Schedule with Pending Thread Queue Length of 8. 81

6.2 Detailed Results for a Schedule with Pending Thread Queue Length of 16. 81

6.3 Characteristics of Chosen Tasks for an Open System. 83

6.4 Detailed Task Arrivals and Execution Time for λ=0.5. 83

6.5 Detailed Task Arrivals and Execution Time for λ=0.75. 84

6.6 Detailed Task Arrivals and Execution Time for λ=1. 84

7.1 Power and Energy Consumption for Matrix Multiplication (f=50 MHz). 88

7.2 Power and Energy Consumption for FIR Filtering (f=50 MHz). 89

7.3 Power and Energy Consumption for FFT (f=50 MHz). 89

7.4 Power and Energy Consumption for RGB2YIQ (f=50 MHz). 90

7.5 Power and Energy Consumption for MM (f= 100MHz). 92

7.6 Power and Energy Consumption for FIR (f= 100MHz). 93

7.7 Power and Energy Consumption for VDP (f= 100MHz). 93

7.8 Power and Energy Consumption for DCT (f= 100MHz). 93

7.9 Power and Energy Consumption for RGB2YIQ (f= 100MHz). 93

LIST OF TABLES
(Continued)

Table **Page**

8.1 FFT Performance and Utilization Comparison Between the Previous VP with the Shuffle Engines and the Modified VP with the Pipelined Data Shuffle Network (f=100MHz) 101

8.2 Resource Consumption and Utilization Percentage 108

8.3 Performance Profile Data for 4Lanes Unfused VP 109

8.4 Performance Profile Data for 4Lanes Fused VP 110

8.5 Performance Profile Data for 2Lanes Unfused VP 110

LIST OF FIGURES

Figure **Page**

3.1 High-level architecture of the multi-lane VP prototyped on a Xilinx FPGA in single host architecture. The vector memory is low-order interleaved. Each vector lane is attached to a private memory..... 17

3.2 Detailed architecture of the four-lane VP applied in single host architecture (FP: Floating-point). 19

3.3 Lane architecture for VP in single host architecture..... 23

3.4 Pipelined structures in the ALU and LDST data paths..... 27

3.5 Multicore architecture for VP sharing (Instr Arb: vector instruction arbitrator).. 29

3.6 The FSM model for AC behavior (CMD: command; APP: application; ACK: acknowledgment)..... 32

3.7 VRF structure in multiple hosts system..... 35

3.8 RMM (Register Management Module) and its TLT interface..... 36

3.9 Data structures used to manage the VRF..... 37

3.10 Duration of fragmented registers for VL=32 and 64. 40

3.11 Detailed architecture of the VP applied in the multiple hosts architecture..... 41

3.12 Vector lane architecture for the multiple hosts system..... 45

4.1 Example of C code showing VP instructions implemented as macro calls for vector addition in single host system. 51

4.2 ISA of the VP..... 51

4.3 Macros to define vector instructions in multiple hosts system. 54

5.1 Speedup for matrix multiplication with and without optimization. (a) VP vs. MB for Algorithm 3 and (b) VP+MB vs. MB for Algorithm 2. 63

5.2 VP vs. MB speedup for FIR filtering with and without optimization. (a) Algorithm 2and (b) Algorithm 1..... 63

5.3 Speedup for FFT. (a) VP with the data shuffle engine vs. MB for Algorithm 2 and (b)VP+MB without the shuffle engine vs. MB for Algorithm 1..... 64

LIST OF FIGURES
(Continued)

Figure **Page**

5.4 VP vs. MB speedup for RGB2YIQ in single host system. 65

5.5 Performance/Area improvement for the VP over the MB in single host system. 66

5.6 Maximum utilization of the LDST and ALU units. 75

6.1 Scheduler flowchart. 78

6.2 Execution time for thread queues of fixed length.(a) Length = 8.(b) Length =16. 80

6.3 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.)..... 85

7.1 Average total dynamic energy consumption per time slice for λ=0.5, 0.75 and 1 94

7.2 Total energy consumption with (w/) and without (w/o) VP sharing, and with power gating, for λ=0.5, 0.75 and 1. 95

8.1 Data shuffling example for the pipelined shuffle network. 99

8.2 Overall architecture of the pipelined data shuffle network. 100

8.3 Mapping of VL, host-to-VM address and VP-to-VM address via virtualization 103

8.4 System architecture of a fusion capable VP of degree two 107

8.5 Fusion of two DCT operations..... 107

8.6 Dynamic power vs. utilization for both ALU and LDST data paths. 112

8.7 Optimal utilization boundaries for a. minimum energy b. minimum energy-execution time product..... 115

8.8 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. 116

9.1 Abstracted architecture of a distributed system with enhanced VPs. 120

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
FSL	Fast Simplex Link
FTS	Fine-grain Temporal Sharing
GPU	Graphical Processing Unit
HDU	Hazard Detection Unit
ILP	Instruction Level Parallelism
LDST	Load Store
LMB	Local Memory Bus
MIMD	Multiple Instruction Multiple Data
MM	Matrix Multiplication
PLB	Processor Local Bus
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SMT	Simultaneous Multithreading
SPS	Scalar Processor Subsystem
TLT	Translation Lookup Table
VC	Vector Controller
VDP	Vector Dot Product

VLIW	Very Long Instruction Word
VLS	Vector Lane Sharing
VM	Vector Memory
VP	Vector Processor
VRF	Vector Register File
VT	Vector Thread
WB	Write Back

CHAPTER 1

INTRODUCTION

1.1 Background History

In the computer world, there has always been an evolving demand for parallel processing and supercomputing. In recent years, this demand has intensified and is complicated by ongoing and unpredictable increases in the size of data and applications. To address this demand, parallel attempts have been made to improve the processing and computing performance of computers in terms of both hardware and software aspects. The ideas behind both approaches are based on exploiting or creating parallelism in the context of processing. This can be inherited parallelism in data load or instruction flow for a single application, or it can be created by simultaneous execution of multiple applications. In this work, various levels of parallelism and techniques are combined in efforts to achieve maximum performance from given resources.

SIMD architectures are highly efficient in exploiting DLP in applications due to their specialized hardware design. 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. Serving as an accelerator, a VP can offload the DLP workload from general-purpose scalar processors, thus enhancing the overall performance and energy efficiency. [Espasa et al., 1997] show that instruction level parallelism (ILP) and DLP can be merged in a single simultaneous vector multithreaded architecture for higher performance. Several VP accelerators have been proposed. The VIRAM's multi-lane architecture has become the basis for several VP designs [Kozyrakis et al., 2003]. It has a basic multi-lane architecture that can be used to build VPs for exploiting DLP through

SIMD processing. Each lane contains similar pipelined execution and load-store units. Each vector register is uniformly distributed among the lanes. All the elements from a vector in a lane are processed sequentially in its pipelined units while corresponding elements from different lanes are processed simultaneously. Using EEMBC benchmarks, it was demonstrated that a cache-less VIRAM is much faster than a superscalar RISC or a cache-based VLIW processor [Kozyrakis et al., 2002].

In this dissertation, a new architecture for a lane-based VIRAM-like VP is first proposed and implemented. The VP can accelerate vector-oriented floating-point applications by using a high-speed load-store unit and dedicated scratch pad memory in each lane. In addition, the designed VP has a multithreaded architecture where several threads may utilize VP resources simultaneously. In this case, resource conflicts are resolved at static time.

To further improve this system, the VP is then augmented to support a register file virtualization technique in order to dynamically resolve relevant resource conflicts. This is joint work with another Ph.D. student. In this scenario, a scalar core is in charge of managing the virtualization process. It is accelerated by dedicated hardware and the VP is modified accordingly to take advantage of virtualization. The managing core can further be utilized for thread scheduling.

In the last part of this dissertation, the VP architecture is first improved to remove some of its structural limitations. The modified VP is capable of achieving higher resource utilization (close to 100 %). It introduces both register file and memory virtualization. The virtualization is performed completely in hardware which results in less overhead. The old non-pipelined shuffle engines are replaced by a pipelined shuffle network. It yields a

scalable and yet flexible VP that is capable of dynamically deactivating some of its computing lanes in order to reduce the static power with minimum performance loss. In addition, the modified simultaneous multi-threaded (SMT) VP can exploit identical instruction flows that may be present in different vector applications by running in a novel instruction fused mode that increases the overall resource utilization. Under instruction fusion, similar copies of an instruction to be run on multiple threads or cores are merged into a single copy for simultaneous execution.

1.2 Motivations and Objectives

In this work, two system architectures are first proposed and implemented, namely single host and multiple host systems. Although both include a VP as a coprocessor, they have different goals. The specific aspects of vector processing targeted by each system are covered in Sub-sections 1.2.1 and 1.2.2 . In the last part of this dissertation, the VP is modified to become more scalable and yet flexible. The modified architecture is called virtualized SMT VP and is subjected to two energy optimization scenarios. The objectives are covered in Sub-section 1.2.3.

1.2.1 Single Host System

In a VIRAM-like architecture, a memory crossbar often connects the vector lanes to the memory banks to facilitate index memory addressing and data shuffling. This crossbar adds extra delay when not actually needed, such as for stride-based data loads and stores. Moreover, it increases the energy consumption. Adding a cache to each lane may solve this problem to some extent but the cache coherence problem will require an expensive solution, often prohibitive for embedded systems. Since in practical applications stride

addressing is more common than other types of addressing [Kennedy et al., 1992], here a VP model is introduced that does not sacrifice performance for less likely memory access instructions. A VIRAM-based, floating-point VP is developed and embedded in an FPGA that connects to a scalar processor. This VP comprises four vector lanes, and provides two separate data paths for each lane to process and execute load and store operations in the LDST (Load-Store) unit in parallel with floating-point operations in the ALU. Each cache-less lane is directly attached to its own local memory. Data shuffle instructions are supported by a shuffle engine in each lane which is placed after the lane's local memory and connects to other lanes via a combinational crossbar. All the local memories connect to the shared bus which is used to exchange data between these memories and the global memory. The prototyping of a system with four lanes shows substantial increases in performance for a set of benchmarks compared to similar systems that do not contain the shuffle engines. This VP is highly flexible for applications with varying VL (Vector Length; it represents the number of elements in the vector), 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. In the single host system architecture, threads of disparate VLs running on the same scalar processor can exploit the VP as long as they do not result in vector register name conflicts. Benchmarking shows speedups of up to 1500 compared to running vector code on a scalar processor with the same clock frequency.

Previously proposed VPs are not versatile enough in multithreading environments. They were mostly capable of handling simultaneously multiple threads using the same vector length in predefined contexts. However, this approach is not often efficient for real

applications since a VP is a rather high-cost, high-performance accelerator that consumes considerable area and energy in multicore processors. A more flexible VP that can be shared dynamically by multiple cores results in better resource utilization, higher performance and lower energy power dissipation. The proposed solution supports the simultaneous processing of multiple threads having diverse VLs. In fact, the VLs used by any given thread are allowed to change during execution. To fully exploit this capability, VP virtualization is proposed and implemented for the multiple hosts system architecture as well as the virtualized SMT VP architecture.

1.2.2 Multiple Hosts System

This work is motivated by the fact that VPs dedicated to single-thread execution on multithreaded or multicore processors are often not efficiently utilized due to the following reasons. First, every application contains some unvectorizable serial code for flow control or other system management; the scalar host processor cannot issue vector instructions to the VP for such an application at a rate sufficient to keep it highly utilized. Second, data dependencies within some applications' vector instruction flows can cause frequent stalls, wasting precious clock cycles in the super-pipelined floating-point units (FPUs) of the VP. Finally, it may be preferable sometimes for applications containing small vectorizable code to be executed on the host scalar processor in order to allow another application with more vector code to exclusively use the VP. However, the execution of the former applications as well could be enhanced given the chance to simultaneously use the VP. Our benchmarking shows that some applications with such a low VP utilization as 5% can yield a speedup of 83 when executed on a VP compared to a scalar processor with the same clock frequency.

Traditional VPs designed to service exclusively one host scalar processor are normally optimized for applications of a certain level of DLP and usually more vector lanes can be added to exploit the increased DLP in new applications [Kozyrakis et al., 2003], [Yiannacouras et al., 2008],[Yu et al., 2009]. However, an increased number of lanes will reduce the utilization of VP resources for other applications with lower DLP. For example, for maximum utilization a VP with four lanes running an application of VL=16 needs to be fed with a new vector instruction every four clock cycles. If the number of lanes is increased to 16 to also accommodate larger applications, for the former application to achieve maximum utilization the VP must receive one vector instruction every clock cycle that the host processor may not be capable of.

To address these challenges, a VP sharing technique named VP virtualization is proposed, for simultaneous multithreading that achieves high aggregate VP utilization independent of the DLP of individual vector threads. The developed multithreaded VP accommodates up to four threads of diverse VLs simultaneously, and can scales effortlessly to support more threads. VP virtualization solves the register name conflicts among threads using a novel VRF virtualization algorithm, which dynamically allocates physical registers of different lengths to threads. With the easy-to-use VRF management kernel functions, programmers are provided with a constant register name space and the management of VRF becomes transparent. VP sharing is applied to the aforementioned multi-lane VP [Rooholamin et al., 2015], and the performance and energy improvement are benchmarked. The new system consists of a VP interfaced with a five-core host subsystem. Four cores share the VP simultaneously for running vector applications whereas the fifth core does VP management and vector thread scheduling. The new VP can run simultaneously up to

four vector threads of various VLs. 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. Benchmarking shows a throughput improvement of up to 400% for many low VP utilization applications compared to the older VP that did not support simultaneous multithreading. A high throughput runtime scheduler for VP threads is also proposed. Experiments show 322% throughput improvement and energy savings of 37% with proper scheduling and power-gating that reduces static energy.

1.2.3 Virtualized SMT VP

VP (co)processors exploit DLP due to their SIMD specialized architecture. The modular design of lane-based VPs empowers scalability. As a VP scales up, however, higher DLP is required to keep its lanes fully utilized. Vector applications optimized for a given VP size would yield poor utilization on its scaled up version with an increased number of vector lanes. Therefore, VP sharing among many on-chip cores is recommended. Without proper resource management, scaling will adversely lead to many idling cycles in each lane for an otherwise efficient vector application, and thus unnecessarily drain static power [Beldianu et al., 2015].

A flexible lane-based VP design is proposed that can wisely and dynamically deactivate some of its lanes toward reducing the static power consumption with minimum performance loss. A novel thread fusion technique is presented as well to be used by our

SMT VP for multiplying its utilization when similar threads coming from different applications are identified in a pending vector task queue. A highly accurate power dissipation model for the VP is developed that is used toward runtime optimization of the energy and/or performance. The complexities of managing the VP's fusion process and dynamic lane configuration are hidden from application programmers via complete VP virtualization; the VP management kernel sets VP state registers for controlling configurable hardware components, and handling vector instruction synchronization, vector memory (VM) access, and vector VRF usage. Each vector application is executed as a thread with its own virtual VM address and VRF name space, and does not need to be recompiled to run under a different VP configuration or fusion state.

In [Beldianu et al., 2015], a dynamic power-gating technique is proposed to control the VP's width (i.e., number of active lanes) in order to achieve optimized performance and/or energy. Compared to [Beldianu et al., 2015], our lane deactivation process is capable of power-gating the entire lane including its dedicated VM bank due to our distributed memory architecture, while the memory crossbar connecting the lanes to the VM and all memory modules always have to stay active in [Beldianu et al., 2015]. Thread fusion [Rakvic et al., 2010] fuses parallel threads that run on the same SMT processor/core. Instruction fusion [Lu et al., 2015] fuses identical instruction flows within unrolled loops. While both fusion techniques are applied to general purpose RISC processors mainly towards energy reduction, the fusion technique presented here boosts VP utilization while reducing the host processors' energy.

This dissertation is organized as follows. Chapter 2 mainly includes related works which target vector processing and multithreading. The similarities and differences

between our work and these works are discussed in this chapter. The architecture of the sub-system of scalar processors and its VP interface for the single host and multiple hosts architecture are covered in Chapter 3. The VP's architecture, including the designs of the hazard detection unit and the VM banks, are discussed for each system separately in this chapter. The VRF virtualization technique and resource consumption of the FPGA prototype are also covered in Chapter 3. Chapter 4 introduces the benchmarks for the evaluation of the proposed systems. Chapter 5 includes performance analysis involving all the results of benchmarking on single host and multiple hosts systems. In Chapter 6, various scheduling algorithms are proposed for the multiple hosts system. Chapter 7 analyzes the power and energy consumption for both aforementioned systems. Chapter 8 covers the hardware modifications in the SMT virtualized VP as well as optimization processes targeting energy consumption via thread fusion and lane configuration. Finally, conclusions and future work are drawn in Chapter 9.

CHAPTER 2

RELATED WORK

This chapter includes an overview of previously proposed hardware accelerators. In Section 2.1, different types of single-threaded vector processors, which are designed to improve high performance computing, are presented. These accelerators may be developed for ASIC or FPGA platforms. They are versatile or application oriented. Like our single host architecture, they are all dedicated exclusively to a single scalar processor. In Section 2.2, various VP sharing techniques are introduced. Various types of multi-threaded architectures are also discussed.

2.1 Related Work in Vector Processing

The SODA VP has a fully programmable architecture for software defined radio [Lin et al., 2006]. Using SIMD parallelism and being optimized for 16-bit computations, it supports the W-CDMA and IEEE802.11a protocols. Embedded systems using a soft core or hard core processor for the main execution unit also have the option to attach a hardware accelerator to increase their performance for specialized tasks. Sometimes these accelerators are realized using FPGA resources to speed up applications with high computational cost. They are often referred to as soft vector processors (SVPs). Designing a custom hardware accelerator that will yield outstanding performance needs good knowledge of HDL (Hardware Description Language) programming. Another SIMD, FPGA-based processor uses a 16-way data path and 17 memory blocks as the vector memory in order to perform data alignment and avoid bank conflicts [Cho et al., 2006]. VESPA [Yiannacouras et al., 2008] is a portable, scalable and flexible soft VP which uses

the same instruction set as VIRAM but the coprocessor architecture was hand-written in Verilog with built-in parameterization. It can be scaled with regards to the number of lanes and yields x6.3 improvement with 16 lanes for EEMBC benchmarks compared to a one-lane VP. It is flexible as the size of the vector length and its width, as well as the memory crossbar, can vary according to the target application.

The VIPERS soft VP is a general-purpose accelerator that can achieve a x44 speedup compared to the Nios II scalar processor [Yu et al., 2009]; it increases the area requirements 26-fold. It supports specific instructions for the applications, such as motion estimation and median filters, and can be parameterized in terms of number of lanes, maximum vector length and processor data width. VEGAS [Chou et al., 2011] is a soft VP with cache-less scratchpad memory instead of a vector register file. It achieves x1.7-3.1 improvements in the area-delay product compared to VESPA and VIPERS. Using scratchpad memory instead of a Vector Register File (VRF), it achieves a speedup of up to 208 compared to the Nios II scalar processor. Further improvements eliminated its ALU bottleneck and the resulting VENICE [Severance et al., 2012] SVP doubled the performance-per-logic block compared to VEGAS. With the integration of a streaming pipeline in the data path of VENICE, a x7000 times speedup results for the N-body problem [Severance et al., 2014].

Application specific VPs are another type of accelerator designed and optimized to expedite specific types of applications. An application-specific floating-point accelerator is built using a fully automated tool chain, co-synthesis and co-optimization for SIMD extension with a parameterizable number of vector elements [Hagiescu et al., 2011]. An application-specific VP for performing sparse matrix multiplication was presented in [Yang et al., 2005]. IBM's PowerEN processor integrates five hardware application

specific accelerators in a heterogeneous architecture to perform key functions such as compression, encryption, authentication, intrusion detection and XML processing for big workload network applications. Hardware acceleration facilitates energy-proportional performance scaling [Heil et al., 2014]. Multimedia applications containing video processing kernels deal with massive DLP. SIMD vector architectures (i.e., VPs) are the best candidates to exploit the parallelism in video frames. In recent years many researchers have tried to optimize codecs for the implementation of new video coding standards such as H.264 or MPEG4. [Iranpour et al., 2004], [Lee et al., 2004], [Kim et al., 2005], [Shengfa et al., 2006] and [Lee et al., 2009] all proposed SIMD-based video codecs focusing on optimizations enhancing the performance.

A major challenge with these VPs is their slow memory accesses. Comprehensive explorations of MIMD, vector SIMD and vector thread architectures in handling regular and irregular DLP efficiently confirm that vector-based microarchitectures are more area and energy efficient compared to their scalar counterparts even for irregular DLP [Lee et al., 2013]. [Lo et al., 2011] introduced an improved SIMD architecture targeting video processing. It has a parallel memory structure composed of various block sizes and word lengths as well as a configurable SIMD architecture. This structure can perform random register file accesses to realize complex operations, such as shuffling, which is quite common in video coding kernel functions. A crossbar is located between the ALU (Arithmetic Logic Unit) and register file.

2.2 VP Sharing Techniques and Comparisons

The idea of VP sharing for multiple threads or cores was first proposed by Beldianu and Ziavras [Beldianu et al., 2013]. Three VP 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). Their FPGA prototype contained two scalar core processors. Under CTS, each core reserves the entire VP exclusively until its current vector thread stalls or completes execution, and then hands over VP access to the other core. FTS is similar to the VP sharing scheme which is proposed here, where all cores access the entire VP simultaneously and VP resource conflicts are resolved by an arbitrator. CTS and FTS support sharing only for threads with the same VL (vector length; it represents the number of elements in the vector). VLS is the only mode under which active threads using different VLs can coexist in the VP which is split into two independent sets of vector lanes, one set for each core; VLS relies on two vector controllers (VCs) to control the two sets. FTS achieves the best VP utilization and may double the speedup compared to CTS while reducing the dynamic energy by 50% [Beldianu et al., 2015].

The work here differs from [Beldianu et al., 2013] in four major aspects. Register name conflicts for VP sharing are solved, a problem that was not mentioned in their work. VRF virtualization greatly improves in practice simultaneous VP sharing. Otherwise, application programmers must rename vector registers statically based on thread combinations that will be present simultaneously in the VP; this is hardly possible in dynamic environments with an unknown, large or infinite number of combinations. Second, [Beldianu et al., 2013] supports VP sharing for two threads of different VL only under the VLS execution mode that configures two independent sets of vector lanes using two VCs. In contrast, we maximize the VP's utilization by allowing multiple threads of different VLs to run simultaneously on the VP. This results in substantial throughput increases. A single VC broadcasts vector instructions to all lanes. The thread ID and VL

reside in each broadcasted instruction. With multiple non-empty instruction FIFOs, round-robin arbitration decides each clock cycle the vector instruction to enter the VP. The thread population in the VP can be increased by modifying the arbitrator's state machine. Third, an added FIFO structure between the VP interface and host cores eliminates frequent stalls of the latter due to vector instruction arbitration. Under low VP utilization, an application's speed is bounded by its host core.

However, when multiple hosts send simultaneously vector instructions to the VP, then only one host will get VP access in the next clock cycle. Such wastage of clock cycles can be avoided with the implemented FIFOs since a core can keep sending vector instructions until its FIFO becomes full; this will occur for peak VP utilization. Finally, the crossbar between the lanes and VM banks 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 its pipelined units that target array operations. The removal of the crossbar also improves scalability of the VP. With both the VM and VRF distributed across the VP lanes, scaling the VP can be effortlessly achieved by attaching more identical lanes to the VC, which will not increase the complexity of individual lanes.

One of the innovations in the proposed VP sharing technique is similar to Intel's proprietary Hyper-Threading Technology (HTT), which is a simultaneous multithreading technique for general-purpose processing [Marr et al., 2002]. The basic differences are: (a) simultaneous multithreading is applied to vector code; (b) the threads may arrive from different core processors; and (c) each logic processor in HTT contains a complete set of general-purpose registers due to a rather small register space; however, a similar VP

approach (i.e., a distinct VRF for each vector thread) would not only require a substantial number of register resources (due to two-dimensional vector storage) but also their average utilization would be drastically reduced due to a much larger register population. This issue is addressed by the proposed VRF virtualization technique; although it maintains a separate logical vector register space for each thread, a shared physical VRF is implemented.

A general-purpose graphics processing unit (GPGPU) is capable of running hundreds of threads simultaneously in each of its streaming multiprocessors (SMs); however, all of the simultaneously executing threads have to be homogeneous. GPGPU relies on thousands of homogenous threads to exploit the DLP in an application, and can only service one host thread at a time. In contrast, a VP thread is already parallel due to the explicit vector nature of its instructions, and the virtualized VP is capable of simultaneously exploiting the DLP in multiple heterogeneous host threads. A VP also consumes significantly less resource compared to a modern GPGPU. Nvidia's latest Maxwell GPU GTX 980 consists of 16 SMs, each with 128 CUDA cores, and the GPU has 5.2 billion transistors [Nvidia Corp. 2014]. Without highly sustained DLP and a fine-grained power management mechanism, many CUDA cores in an SM may idle frequently, and thus lead to prohibitively low resource utilization and high static energy consumption.

CHAPTER 3

PROPOSED VECTOR COPROCESSOR

In this chapter, two system architectures are proposed. For the single host system architecture, the proposed VP is exclusively utilized by one scalar processor while it is placed on the shared bus and any scalar processor connected to that shared bus can use the VP. In the multiple hosts architecture, VP virtualization is presented where multiple cores can share the VP using simultaneous multithreading.

3.1 Single Host System Architecture

Figure 3.1 depicts the basic architecture of the FPGA-prototyped VP in the single host architecture. A single scalar core is based on Xilinx's soft core MicroBlaze (MB), fetches instructions from its instruction memory (not shown in the figure) and issues them to appropriate execution units. The MB is in charge of executing all scalar and control instructions while vector instructions are sent to the VP. The shuffle engine, which is distributed along the lanes, is activated only to realize vector data shuffling with multiple vector lanes. The design introduces two innovative concepts. First, it removes the competition of lanes to access memory banks, which is the case for earlier works, by employing cache-less private memories for the lanes; the private memories form a low-order interleaved space that resides between the lanes and the global memory. Second, the vector length can vary even between instructions in the same thread. In all previously introduced VPs, the vector length was defined for each working context, program or thread. It was usually a fixed number for each thread and was set in advance by the scheduler. In contrast, this model allows the programmer to define the vector length for each individual

instruction. As a result, the vector length can vary widely, even for instructions in the same loop. This unique feature of the VP is well exploited through VP virtualization.

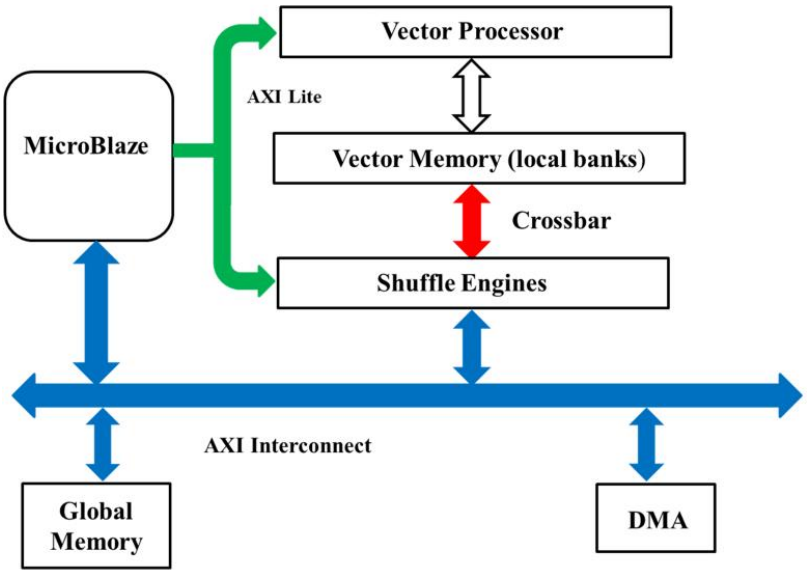

Figure 3.1 High-level architecture of the multi-lane VP prototyped on a Xilinx FPGA in single host architecture. The vector memory is low-order interleaved. Each vector lane is attached to a private memory.

Data needed by applications running on the VP should be preferably stored in the private memories of lanes. Since these private memories connect to the AXI (Advanced eXtensible Interface) shared bus, copying the data from the global memory could be done either by the MB or the DMA engine as both have access to the shared bus. If the instruction and data caches also of the MB are placed on the AXI interconnect, the time needed to copy the data from the global memory to either the vector memory or the MB data cache will basically be the same. The same principles are applied for writing back from the VP private memories or the MB data cache to the global memory. Block data are placed in consecutive locations in the MB data cache while low-order interleaving among lanes is used for the vector memory.

To evaluate the proposed VP model, an FPGA prototype is created with four lanes and four on-chip memory banks that serve as local memories. The VP model is modular and can be easily extended to include more lanes. The Xilinx Virtex6 xc6vlx240t-FF748 FPGA device is used. To reduce the complexity of the hardware design in order to track operations progressing through the data path, rather simple execution units are included in the vector lanes. Since each lane directly connects only to its private memory in order to avoid contention when accessing memory banks, a very fast load-store unit was designed in each lane as there is no chance of stalling during memory access instructions. Contention when accessing a memory bank can only happen in the case of data shuffle instructions which, however, are totally handled by each lane's shuffle engine. Since the distributed shuffle engines employ other ports of the private memory banks than those that connect to lanes, other vector instructions can be executed while realizing data shuffling as long as no data hazard exists between the involved instructions. Figure 3.2 shows the detailed architecture of the single host system prototype.

The hardware design of the vector lanes, vector controller, scheduler, data shuffle controller, data shuffle engines, and combinational crossbar and mux was done by writing VHDL code. Xilinx IPs (Intellectual Properties) were used for the realization of the memory banks, memory controllers, MB, and AXI4 and AXI4 Lite interconnects. The VP was developed using Xilinx ISE version 14.5. The MB was added to the project using the EDK tool while its configuration and connection to the peripherals was done using Xilinx XPS. Since this work focuses on proof of concept, the prototyped VP consists of four lanes and has 1024 32-bit registers in the VRF. It also contains a 64Kbyte vector memory that

can accommodate the largest developed benchmark. The rest of this section contains the VP details.

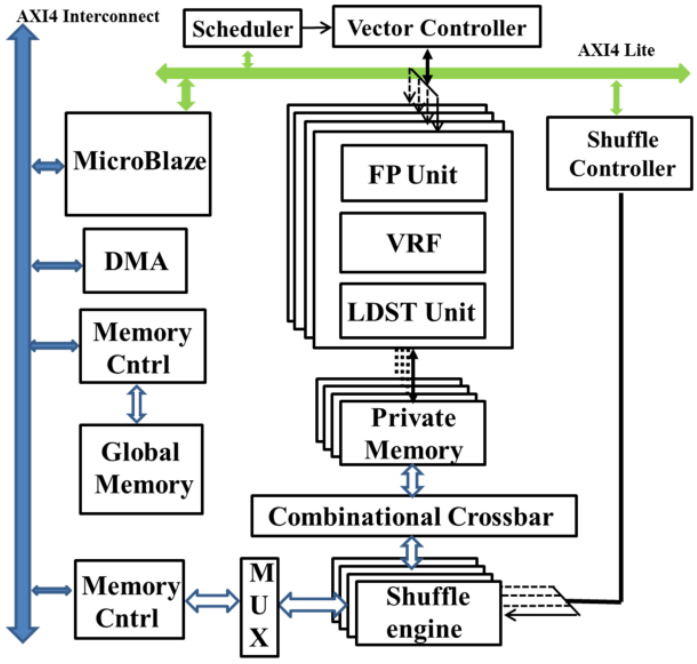

Figure 3.2 Detailed architecture of the four-lane VP applied in single host architecture (FP: Floating-point).

The MB soft core is a 32-bit RISC Harvard architecture [Xilinx Inc., 2010] that supports the AXI4 and LMB (Local Memory Bus) interfaces. Version 8.40.a is implemented with five pipeline stages and an FPU. Data and instruction caches can be connected to either bus. For flexibility, the memory blocks are connected to both the AXI4 and LMB buses. Each bus requires its own memory controller. Only one AXI4 memory controller is used to create a slave interface for the vector memory. The AXI4 interconnect is good as a shared bus for high-performance memory mapping and can support up to 16 slaves and 16 masters [Xilinx Inc., 2012]. The AXI4 crossbar can realize every transfer between interconnected IPs, like memories. Moreover, it supports the DMA-based burst

mode for up to 256 data transfer cycles which is suitable for transfers between the global and private memories.

To connect the VP and shuffle controller to the MB for vector instruction transfers from the MB, the AXI4 Lite interconnect is used which is appropriate for this type of non-DMA memory-mapped transfers. The slave interfaces for connecting the VP and shuffle controllers to the shared bus are developed using the create-and-import peripheral wizard in Xilinx XPS. They both contain control registers which can be read and written by the MB through the AXI4 lite interconnect. A hardwired scheduler for accessing the VP is included in the VP interface. The main responsibility of the scheduler is to grant VP access to a requesting MB based on the vector length it asks for and the current availability of VP vector registers. Vector instructions are written into the VP using memory mapping.

3.1.1 VP Architecture and Instructions in the Single Host System

Two types of vector instructions are used by the VP. The first type does not contain data and all the required fields for executing the instruction are placed in the 32-bit instruction; vector-vector ALU instructions are of this type. The other instruction type consumes 64 bits that contain a 32-bit operand value; e.g., vector-scalar ALU instructions are of this type. Since the main focus here is proof of concept for the hardware design, an advanced compiler for the VP was not developed. Inline function calls are included in the C code for the MB; they represent VP instructions and their realization involves macros. Since the VP's instruction input port is viewed as a memory location by the MB in this memory mapped system, a small delay may occur between issuing an instruction of the second type and the arrival of the needed operand. Thus, the scheduler sends them together to the VP when the data becomes available.

Every MB that has access to the AXI4 Lite interconnect can send a request to the scheduler for VP resource access. Each MB can access the VP as a peripheral device using two different addresses, for sending a VP request or release instruction and a vector instruction upon VP granting, respectively. Requests are granted by the scheduler. Threads initiate a request to the scheduler in advance using a 32-bit instruction. This request instruction includes the VL per register and the number of vector registers needed by the thread. An affirmative reply by the scheduler will include a 2-bit thread ID that can be used to get VP access. This will occur only if there is enough space in the vector register file to accommodate the request. The MB running the thread will include this ID in all vector instructions sent to the VP. Vector register renaming and hazard detection rely on this type of ID. If the aforementioned conditions for the thread are not satisfied, the scheduler will reject the thread request with information about the currently available VL and vector registers. Although our hardware implementation allows different vector instructions to employ different VLs, a complicated register renaming unit will be needed. Therefore, for the sake of simplicity, in single host system it is assumed that the VP can handle two threads at a time, from the same or different MB cores, where all instructions in both threads use the same VL. Otherwise, it will be the compiler's or programmer's responsibility to employ registers that will guarantee no conflicts in the VRF. Threads release VP resources by issuing a release instruction to the scheduler.

The VP scheduler interfaces the VP via the VP controller (VC). The latter has a pipelined architecture that consumes three clock cycles for register renaming and hazard detection. Since the VP connects to AXI4 Lite via the shared bus, it can receive instructions from any scalar processor that connects to that bus in a multicore environment; thus, the

VC unit can accept vector instructions from a multitude of threads and carry out register renaming, if needed. The register renaming stage for single host system is completely implemented in hardware and takes only one clock cycle. However, it cannot be applied for comprehensive multithreading as it requires threads of similar VLs. RAW (Read-After-Write), WAW (Write-After-Write) and WAR (Write-After-Read) data hazards are resolved by the hazard detection unit in the VC. This unit resolves all possible hazards in accessing vector registers in the lanes by using an appropriate instruction tagging mechanism. Adding a tag to each instruction allows handshaking between the VC and VP. The same instructions are issued simultaneously to all four lanes.

The detailed architecture of each lane is depicted in Figure 3.3. The data paths for memory and ALU instructions are completely separated in each lane, and related instructions and data are queued in different FIFOs. All the instructions and data in a lane are represented using 32 bits. Memory accessing instructions always contain 32-bit additional data to represent the private memory base address to be used. ALU instructions for vector-scalar operations also contain a 32-bit floating point scalar. ALU instructions are decoded by the ALU decode unit and the needed operands are fetched from the VRF. The VRF in each lane consists of 256 32-bit locations that can store 256 single-precision floating-point vector elements. It is accessed using three read and two write ports since the ALU and load (part of the load-store LDST) units need two and one read port in order to simultaneously read two and one operand respectively, and the register WB (Write-Back)

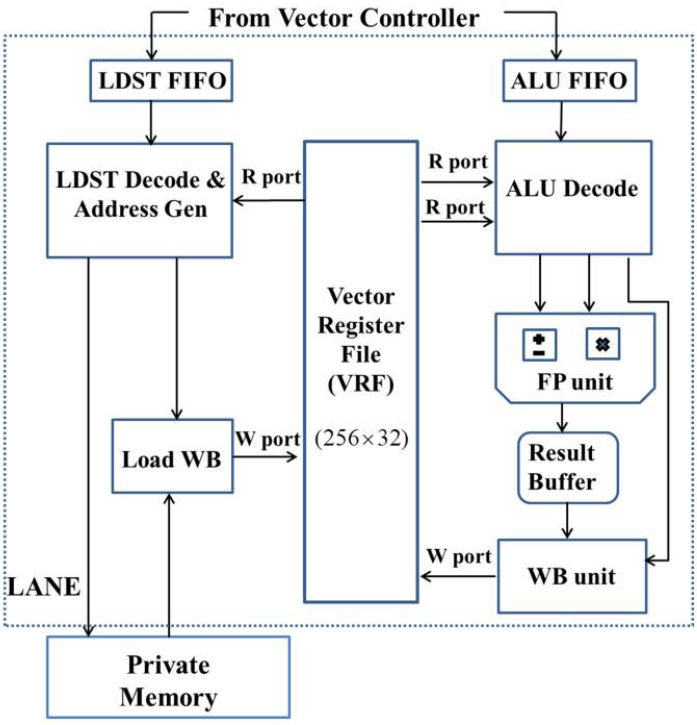

Figure 3.3 Lane architecture for VP in single host architecture.

and store (part of LDST) units require one write port each. In the case of contention, when different ports want to perform different tasks simultaneously on the same location in the VRF, the write first policy could be applied. The design results in one clock cycle latency for sending the output to related ports; it uses output enable ports to ease the reading task. Reading from the VRF is possible only when the output enables are triggered. The ALU decode unit requires two read ports when reading a pair of floating-point operands to realize vector-vector instructions. The ALU execution unit in the lane contains a floating-point adder/subtractor and a multiplier that were developed using open source code [Open cores, 2012]. This unit has six pipeline stages for addition and subtraction, and four stages for multiplication; it performs operations on 32-bit single-precision floating-point data. The results of the execution unit are sent to the WB block which connects to a write port of the VRF for writing one element per clock cycle in a pipelined fashion.

Absolute and indexed memory addressing are used to access the private memories. Absolute addressing may employ a non-unit stride. The LDST unit fetches the register content for a store instruction from the VRF and generates the destination address for the lane's private memory using the base address that arrived right after the instruction. It uses only one VRF read port. Each vector memory instruction issued to the lane has two 32-bit fields. The first field contains the source or destination register and the stride value, whereas the second one is a base address in the lane's private memory. Indexed addressing for the private memory is realized using the data shuffle engines. For load instructions, the WB unit writes the fetched memory contents into the proper register using a write port at the rate of one element per clock cycle. In the prototyped VP with four lanes, 1024 (i.e., 4 lanes *256 elements/lane) vector elements can reside in the VRF; the VRF is divided evenly among the four lanes so the VL must be a multiple of four. Hence, the VRF can be configured as 16 vector registers with VL=64, or 32 registers with VL=32, or 64 registers with VL=16. The location of register elements in the VRF depends on the VL value and the register ID. In the case of VL=64, for example, register "r0" contains all the elements of "r0" and "r1" for VL=32.

The ALU and LDST decode blocks in each lane include counters for synchronization when reading from the VRF and feeding the data to the next block; they are initialized based on the VL assumed by the instruction. Since the design avoids memory stalls by making a private memory available to each lane, all lanes remain synchronized in the full pipeline utilization mode where one element is processed every clock cycle in the lane. This synchronization flexibility allows dynamic changes of VL's value for any given instruction. For example, the vector-vector instruction "r2 <= r0+r1" for VL=32 can be

substituted by the two vector-vector instructions “r4<=r0+r2” and “r5<=r1+r3 for VL=16, and vice versa, within a thread or a loop since the corresponding registers include the same elements from the VRF (as per the preceding paragraph).

For memory access instructions without data shuffling, the shuffle engine adds no delay since the combinational crossbar is placed in the middle of the connection between the AXI4 shared bus and the memory port. Since both the shuffle engine and the MB use the same memory ports when accessing the private memory, only one of them can write to or read from the memory in any given clock cycle; the access decision is made by the shuffle engine and is realized via the crossbar. Each lane uses independent ports to access its private memory and the LDST unit can execute the next memory access instruction while data shuffling is performed as long as there are no data hazards. If there is an access contention on a memory bank while running a data shuffle instruction, the shuffle engine will apply the round robin scheduling policy. Indexed memory addressing also can be realized by the shuffle engine. The shuffle controller simultaneously provides to all four shuffle engines the information needed for shuffling (i.e., the source, destination and index register values).

3.1.2 Pipelined ALU and LDST Unit

The VC as well as the vector lanes are pipelined. The first block in the VP’s data path is the VC which has three pipeline stages for register renaming, hazard detection, and separating for forwarding the ALU and LDST instruction word components (e.g., base address or scalar operand), respectively. Two clock cycles are consumed in either FIFO to pass an instruction and its data to the VP. The ALU decode unit consumes four clock cycles for decoding, fetching operands and feeding them to the execution unit. The floating-point

execution unit consumes six clock cycles for processing and an additional cycle to receive an acknowledgment from the WB unit after writing a result into the VRF. Thus, the total latency for filling up the pipeline with ALU instructions is sixteen clock cycles (accounting for all delays in the lane and VC), as shown in Figure 3.4 (the first three stages are inside the VC).

Memory access instructions are decoded by the LDST decode unit which contains six pipeline stages for instruction decoding, data fetching from the VRF and address generation when executing store instructions. For a load involving the private memory, two more clock cycles are added representing a memory access and data latching by the WB unit, respectively. There is also one clock cycle delay between fetching consecutive vector instructions from either FIFO. This delay eases functional verification and instruction tracking through the data path during behavioral simulation, since it represents a high-impedance state ('Z') delineating consecutive instructions. The total latency for filling up the pipeline is 11 and 13 clock cycles for a store and a load instruction, respectively, as shown in Figure 3.4. For data shuffle instructions, the data path consists of the shuffle controller and shuffle engines. For a shuffle instruction, the shuffle controller accepts three addresses representing the location of the source, destination and index data in the vector memory. This controller will not initiate data transfers until all the required information for the desired permutation becomes available. After sending the information to the shuffle engines, four clock cycles are needed per element to fetch the data and the corresponding index from the memory, and at most four more clock cycles to apply round-robin scheduling upon data collision involving any of the four private memories.

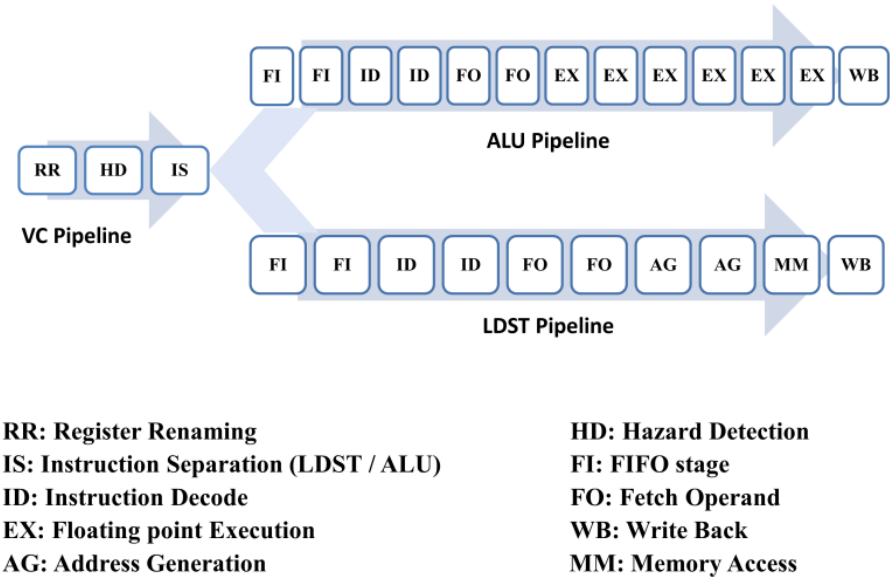

Figure 3.4 Pipelined structures in the ALU and LDST data paths.

3.1.3 Resource Utilization in Single Host System

Before demonstrating the proposed architecture's performance achievements, it is essential to know the silicon area occupied by this design. The architecture of Figure 3.2 was synthesized for the Virtex6 xc6vlx240t-FF748 FPGA device which is organized in columns and is built with a 40nm copper CMOS process. This Xilinx device includes 37,680 slices, where each slice contains 4 LUTs and 8 flip flops for realizing configurable logic. It also includes 768 DSP48E1 DSP modules, where each module contains an 25*18-bit multiplier, an adder and an accumulator. There are also 344 block RAMs (BRAMs) of 36 Kbits each which are used to realize memory components in digital designs. The overall resource consumption of our design is presented in Table 3.1. The MB system consumption in the table is without the VP and the connection interfaces of Figure 3.2. It can be

concluded that the VP accelerator consumes almost 11times as much area as the MB in an effort to speed up data-parallel applications. Also, the data shuffle engines do not consume many resources. It will be extrapolated further in Chapter 7 by investigating the dynamic energy consumption of these resources for a set of benchmarks.

Table 3.1 Resource Consumption for Single Host System.

Entity	Slice Registers (% Utilization)	Slice LUTs (% Utilization)	RAMB36E1s (% Utilization)	DSP84E1s (% Utilization)
Vector Processor	45212 (14.9%)	69127 (45.8%)	0	4 (0.5%)
Vector Memory	2 (0%)	296 (0%)	16 (3.8%)	0
Shuffle Engines	1320 (0.4%)	1228 (0.8%)	0	0
MB System	4947 (1.6%)	6183 (4.1%)	16 (3.8%)	3 (0.4%)

3.2 Multiple Hosts System Architecture

As shown in Figure 3.5, this prototyped system consists of two sub-systems, namely a heterogeneous component with five scalar processors and the VP. The scalar processors sub-system (SPS) runs system managing applications as well as the flow control part of vector applications, and sends vector instructions to the VP. The TLT, which provides hardware support for real-time VP register renaming, is also managed by the SPS. The interface between the SPS and the VP is pipelined, and the VP can read up to one 32-bit instruction/datum and three 6-bit physical register names from the SPS in each clock cycle. A detailed discussion of the SPS follows. More VP details follow in 3.4.

MB, a 32-bit RISC embedded soft processor provided by Xilinx, is the chosen architecture for the scalar processors, namely MB0 to MB4, in the SPS. The MB's Harvard architecture in the SPS interfaces a fast local memory (LM) that stores frequently used library functions. LM blocks can be initialized from the FPGA's flash memory upon power

up; the connections are omitted in Figure 3.5. The libraries can also be modified at runtime by attached MBs. In addition to regular load/store instructions which can access memory and I/O devices mapped within the 4GB address space, MB supports a special interface known as AXI4-Stream (AXI4-S). The MB's AXI4-S interface can be accessed using put/get instructions; each AXI4-S interface consists of one input and one output port, providing a low latency dedicated link to the processor's pipeline. Each MB can be configured with up to 16 AXI4-S interfaces. The AXI4-S interface is widely used in the developed multiple hosts system for inter-core and core to custom-hardware connections.

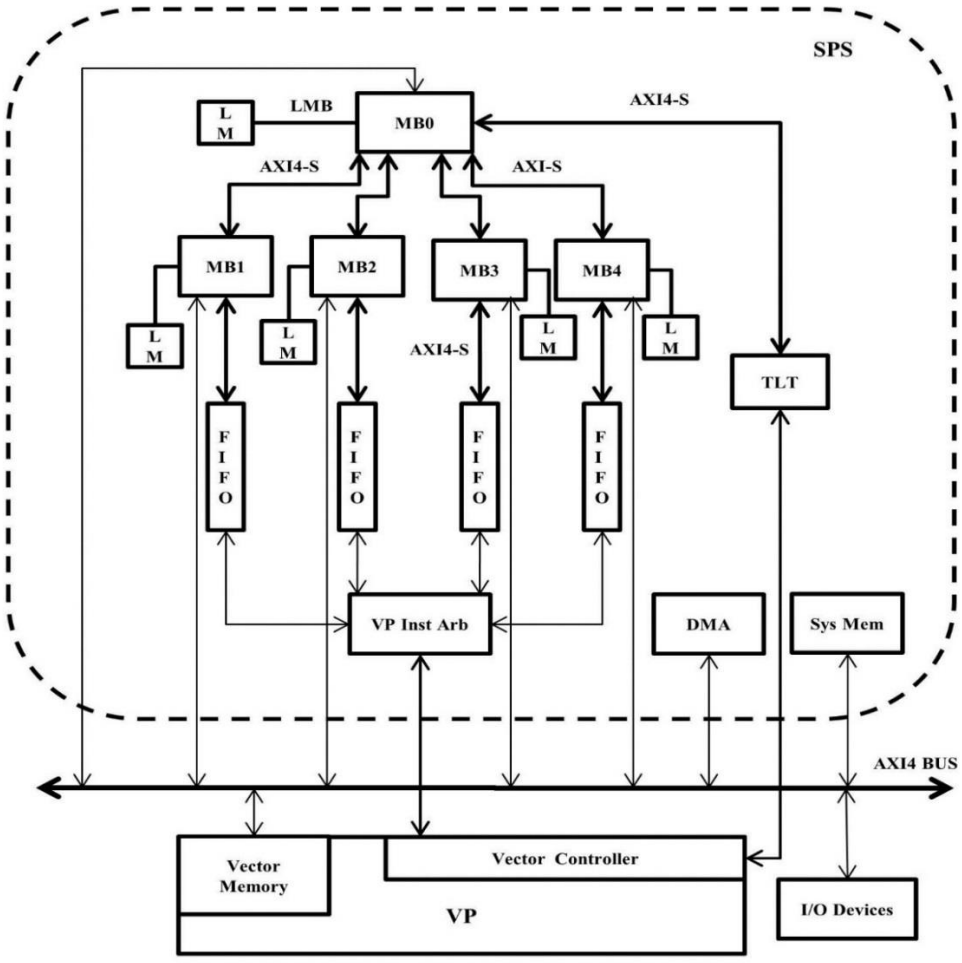

Figure 3.5 Multicore architecture for VP sharing (Instr Arb: vector instruction arbitrator).

Put/get instructions are of two types: blocking and non-blocking. Blocking instructions will stall the MB pipeline if the receiver/sender is not ready to receive/send data. On the other hand, for non-blocking instructions the processor will keep executing instructions without receiver/sender acknowledgments; a polled software flag indicates a successfully completed transfer. Both types of put/get instructions are used for reasons explained later in this section.

3.2.1 The System Core

MB0 is at the center of the SPS. It is connected to the other four MBs and the TLT using the AXI-S interface. MB0 performs the following tasks: i) It runs the register management algorithm that supports VRF virtualization. ii) It updates the TLT based on the mapping of virtual vector registers used by a thread to available physical registers in the VRF; the mapping is produced by the register management algorithm. iii) It estimates the VP utilization using information for tasks running on the VP and schedules new vector threads based on this estimate. iv) For simplicity without loss of generality, in the benchmarking MB0 notifies the application cores (MB1-MB4) about new tasks assigned to them. v) Finally, it polls MB1-MB4 for task completion before releasing VP resources.

MB0 is connected to the TLT using only the output port of its AXI-S interface. It uses a non-blocking put instruction since MB0 knows when the TLT is ready to be written. The connections between MB0 and the slave cores are bi-directional and facilitate non-blocking put/get instructions to free MB0 from slave acknowledgments while enabling the fine-grain monitoring of slave status. MB0 knows the state of every slave core and only assigns tasks to idle cores. Using a non-blocking get instruction, MB0 polls frequently each slave for task completion. A task completion flag written by the slave is checked by MB0

for avoiding the premature release of VP resources occupied by the task. MB0 is attached to a fast 32KB LM that contains the register management and thread scheduler codes. Since MB0 needs to run only integer code, an FPU is omitted for resource and power efficiency.

3.2.2 Application Cores

MB1-MB4 serve as application cores (ACs). Each AC runs applications that may contain function calls to vector kernels. These vector kernels are part of a library stored in the attached 16KB LM. For the sake of benchmarking the proposed VP virtualization technique, it is assumed here that the ACs receive commands from MB0 to execute vector kernels and then send an acknowledgment to MB0 upon successfully completing this task. This behavior of the ACs is represented by the finite state machine (FSM) of Figure 3.6. The AC, which starts in the wait state, executes an application after receiving an MB0 command. When the application finishes, the AC sends an acknowledgment that sets a flag which is periodically checked by MB0; the AC goes back to the wait state. Serving as slaves to MB0, the ACs use blocking put/get instructions to communicate with MB0 through the AXI-S interface. Blocking put/get instructions ensure that an AC trying to communicate with MB0 stalls its pipeline until a command or acknowledgement arrives from MB0.

Each AC is also configured with another AXI4-S interface that connects it to its dedicated vector instruction FIFO (see Figure 3.5). An AC running vector application kernels generates vector instructions which are forwarded to this FIFO. Vector instruction details are covered in Section 4.2.1. Each vector instruction goes through the VP instruction arbitrator before it reaches the VP for decoding and execution. The AC runs the serial code of the application.

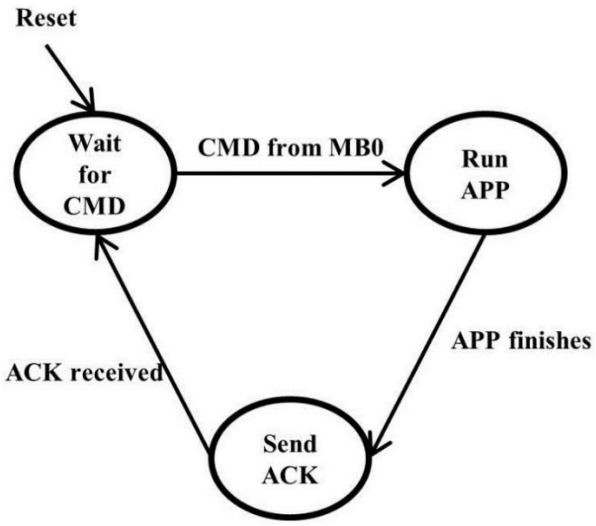

Figure 3.6 The FSM model for AC behavior (CMD: command; APP: application; ACK: acknowledgment).

3.2.3 Vector Instruction FIFOs and Arbitrator

The vector instruction FIFOs in the prototype are constructed using Xilinx IPs and are configured as First Word Fall Through (FWFT) FIFOs with a depth of 16 32-bit words. The arbitrator is custom hardware that is developed in VHDL. The FIFOs and arbitrator play important roles in system performance, especially when the VP utilization is relatively low for the following reason. To ensure that each vector instruction is received properly, an AC sends vector instructions or relevant data using a blocking put instruction that stalls the AC pipeline until the transaction is acknowledged. Due to VP sharing in this system, multiple ACs may send vector instructions at the same time. With the FIFOs added between the ACs and the VP, each AC can keep issuing vector instructions until its dedicated FIFO becomes full, which implies that the VP has saturated.

A smart round-robin arbitrator is implemented to share the VP resources equitably among all four ACs. To eliminate unnecessary clock wastage, only non-empty FIFOs are

polled. The FIFOs and pipelined arbitrator are carefully designed for high throughput. The arbitrator consists of two stages that realize arbitration and handshaking with the VP's receiving unit, respectively. The FIFO and arbitrator interconnects provide a bandwidth of one 32-bit instruction/data per clock cycle with the SPS and VP respectively, which suffices to sustain peak VP performance.

3.2.4 The System Bus

An AXI4 bus connects the five MBs with the system and vector memories. This 32-bit system bus is optimized for high performance with separate read and write channels; it also supports incremental bursts of up to 256 bus-wide data transfers. The 128 KB system memory is accessible by the five MBs and external I/O devices, while the vector memory is accessible by the MBs and the VP. Application data are initially stored in the system memory and are moved to the VM for VP processing. A DMA engine can expedite these 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 can be achieved between the VP and the VM compared to a system with a crossbar [Beldianu et al., 2013]. The other port of each bank is connected to the system bus in a 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 accesses. I/O devices attached to the system bus can support debugging, display or other data input/output capabilities. For VP benchmarking, data is initialized in the system memory and configure LEDs for debugging using general-purpose input/output (GPIO) channels.

3.3 VP Virtualization

Without loss of generality, the multiple hosts system prototype supports VP sharing with up to four threads running simultaneously, where each thread uses a VL from the set {16, 32, and 64}. To support VP sharing for achieving the highest thread throughput, a runtime VRF virtualization technique was invented that resolves register conflicts among competing active threads. Each vector thread is programmed with its own independent virtual register name space and at run time virtual register names are mapped to physical names based on the availability of VP registers. The VRF virtualization technique involves two components: (1) a register management algorithm run by MB0 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 the mapping process. Using a convenient programming interface for this prototype, which is supported by the VRF virtualization technique, applications have access to virtual vector registers 0 to 31 for VL=16 or 32, and 0 to 15 for VL=64; this choice matches the physical VRF size as discussed in the next subsection. It is not assumed the uncommon case of 64 vector registers with VL=16 since it will also increase unnecessarily the complexity.

3.3.1 The Vector Register File

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 3.7, **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 the VL of the instruction to physically locate a register in the VRF. In the VP design, each vector instruction contains a 2-bit thread ID, the 5-bit IDs of involved virtual registers, and the VL of the instruction encoded in a 2-bit field. The thread ID and the virtual register IDs are used to obtain physical register IDs from the TLT, as discussed in the following section.

3.3.2 The Vector Register Management Algorithm

The functional blocks of the register management module (RMM) and its TLT interface are shown in Figure 3.8. The vector register management algorithm is developed to support an independent/virtual space of 32 vector registers for each thread. The RMM receives as input a request to either allocate new registers, with the needed VL and number of registers, or release registers, with the ID of a vector thread that just completed execution. After

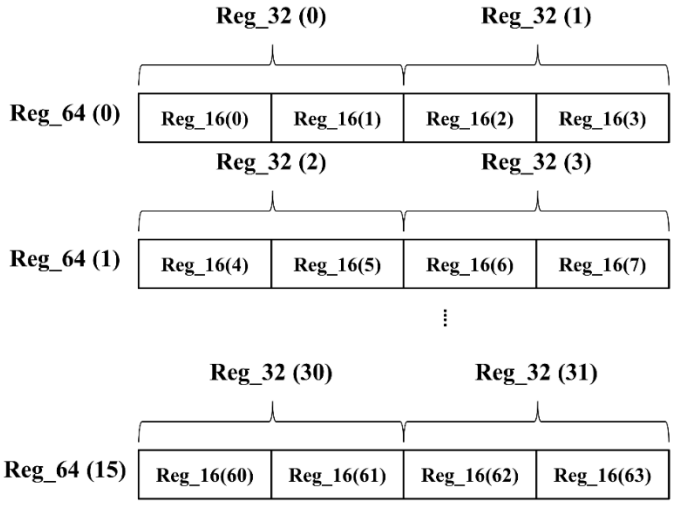

Figure 3.7 VRF structure in multiple hosts system.

properly processing the input command and updating the register and thread state accordingly, the RMM responds to the corresponding core by providing the assigned thread

ID. 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. More details follow later in this section. For our current benchmarking, the functionality of RMM is realized in software by MB0. A hardwired version of RMM is a future objective towards even higher performance and lower energy consumption.

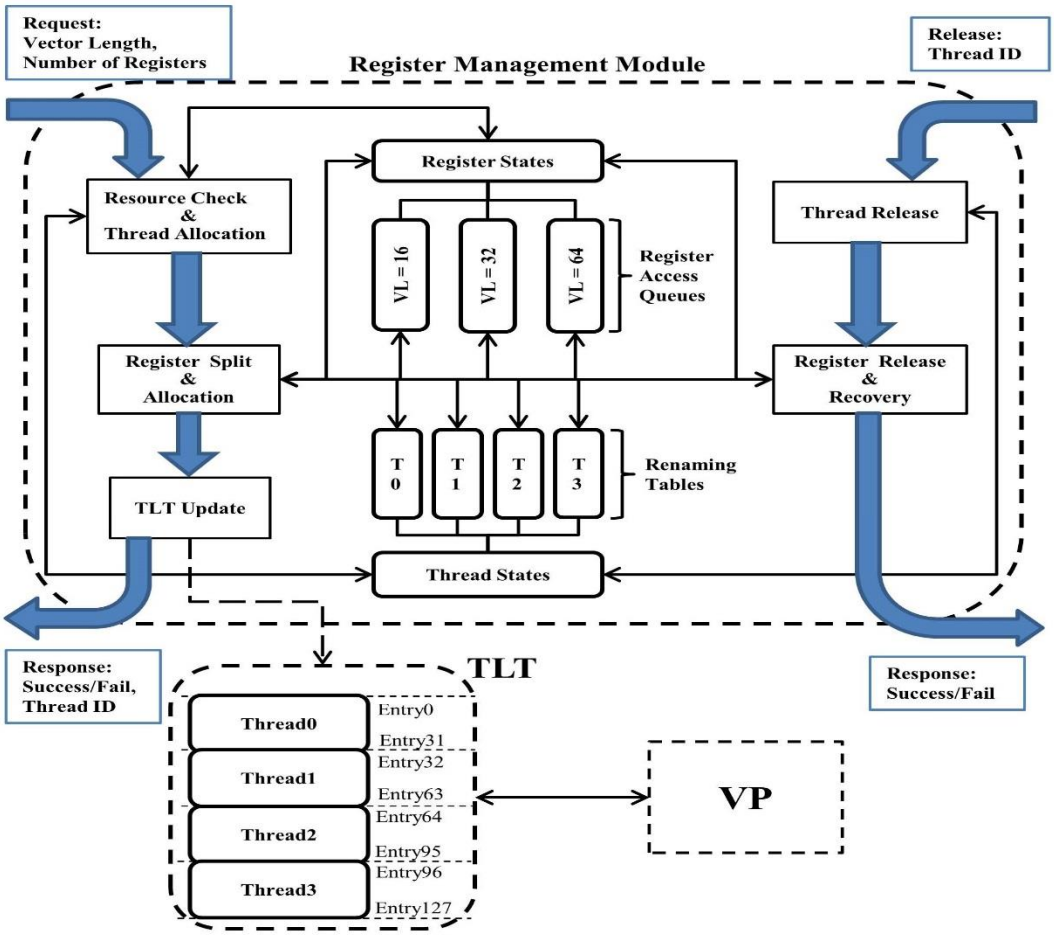

Figure 3.8 RMM (Register Management Module) and its TLT interface.

Figure 3.9 shows two data structures for VRF management. *Struct* **vp_control** contains data needed to manage VRF. 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 for access, **vp_reg** can also be accessed using the quick access queue. Inside **vp_reg**, field **rname** is the physical name of the register; it initializes to the index within the array. Field **in_queue** is set to '1' when a register is put into the fast access queue; it is available to be assigned to a thread or to be split for 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 used to implement the fast access queue (a doubly linked list).

```
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 3.9 Data structures used to manage the VRF.

The fast access queue 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 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; therefore, 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. Fields **in_que_64**, **in_que_32** and **in_que_16** are initialized to 16, 0 and 0, respectively.

3.3.3 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 will be 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** will later be used 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 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)**.

3.3.4 VRF Fragmentation Issues

The 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 algorithm, an experiment was performed involving random VP request/release calls. After each request/release call, the number of fragmented **reg_32** and **reg_64** are counted. The number of request failures are also counted due to register fragmentation. 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. This 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 3.10. 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.

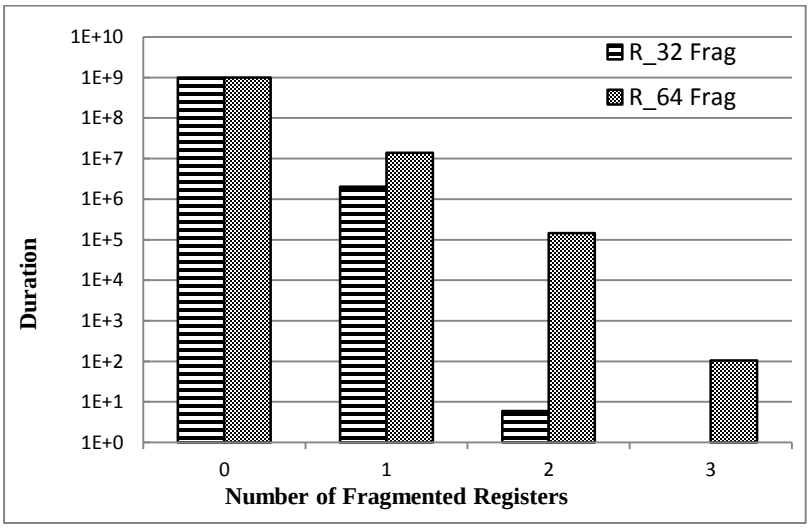

Figure 3.10 Duration of fragmented registers for VL=32 and 64.

3.4 VP Architecture in the Multiple Hosts System

The VP consists of a VC, a data hazard detection unit, the VRF containing 1024 32-bit elements, a VM of size 64KB, and four vector lanes; each lane has a LDST unit and a

FPU. The VM is divided into four low-order interleaved banks; each bank is a true dual-port RAM with one port connected exclusively to one of the vector lanes and the other port connected to the system bus shared with the SPS. Whereas each vector lane can only access its own dedicated memory bank, all scalar processors 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 the VM banks using either the DMA engine or one of the ACs. Figure 3.11 shows the detailed architecture of the VP prototype applied in multiple hosts system. Two types of vector instructions are used by VP which were described in Section 3.1.1. Since here also focus is proof of concept for the hardware design, an advanced compiler was not developed for the VP. Vector instructions are generated by ACs using macro definitions in C code, and are sent to the VP via the arbitrator interface.

The VP has the same pipeline stages as discussed in Subsection 3.1.2. The internal architectures for realizing the register renaming and hazard detection stages are completely different. The following sections describe how these two first stages work to support SMT performed on the multiple hosts system.

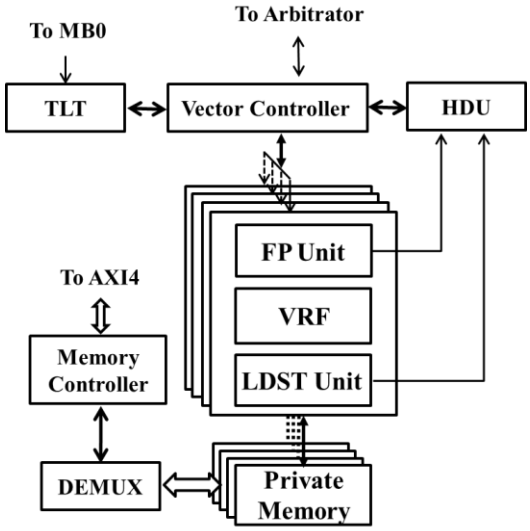

Figure 3.11 Detailed architecture of the VP applied in the multiple hosts architecture.

3.4.1 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 Section 3.3.2.

3.4.2 Hazard Detection

After updating the register name fields, instructions enter the Hazard Detection Unit (HDU). RAW, WAW and WAR data hazards are detected by this unit to provide control signals to the VC. The VC then resolves all potential data hazards by stalling instructions that are dependent on other instructions in the pipeline, to assure the proper order of register access. This prototype can process simultaneously four vector threads by assigning distinct IDs to threads. Since there is no data dependency across different threads, the HDU is only responsible for detecting data hazards within each thread. The modular HDU design is scalable to eventually support more simultaneously executing threads. Each HDU module has a temporary slot that buffers the previous instruction of a thread that entered the vector

lanes, and a counter that counts the number of remaining same-thread instructions in the lanes. A buffered instruction is a potential cause of hazard since the next incoming instruction may depend on it. The counter of instructions is incremented by one upon the VC issuing a new vector instruction from the same thread; it is decreased by one when an instruction from the thread completes execution. Involved lanes broadcast an acknowledgment with the thread ID to all the HDU modules when an instruction completes; the module with the matching thread ID will then update its counter. Due to the separate data paths for ALU and LDST instructions in the lane design, the counter may be decremented by two when two instructions simultaneously completing execution in the two data paths belong to the same thread. A counter value of zero means that there is no pending instruction in the lane for this thread, so there is no need to check the buffered instruction for hazards. When an instruction enters the HDU, the HDU module that corresponds to the instruction's thread-ID is chosen to perform hazard detection. The instruction is compared against the buffered instruction in the module; if a data hazard is detected, the instruction will be stalled from entering the lanes until the counter's value is reduced to 0.

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 design targets SMT assuming no dependencies among threads, the HDU's performance impact is almost negligible.

3.4.3 Vector Lane Structure in the Multiple Hosts System

The detailed lane architecture is depicted in Figure 3.12. The modular VP model can be easily extended to include more lanes. To reduce the complexity of the hardware design in order to track the progress of operations through the pipeline, relatively simple execution units are used in the vector lanes. Once a vector instruction has passed the hazard checking phase, it is broadcasted to all vector lanes for execution. The pipelines for LDST and ALU instructions are exactly the same as discussed in Section 3.1.1. The only difference is related to performing handshaking for hazard detection. In the tagging mechanism, only the TAG field of every instruction is sent back to the VC once the instruction has passed the safe point; in contrast, as discussed in Section 3.4.2 here the thread ID is passed.

For load instructions, the WB unit writes the fetched memory contents into the proper register using a write port at the rate of one element per clock cycle. Each lane is directly connected to its private memory in order to avoid contention when accessing memory banks, and therefore a high throughput LDST can be implemented since memory access will never be stalled due to arbitration. The need for arbitration often drags down performance in designs where all memory banks are accessible by all lanes. Since a VP lane only occupies one port and every VM bank is dual-ported, the other port can be dedicated to the AXI4 bus. With such a configuration, LDST instructions can be executed without affecting data transfers between the system memory and VM. The ALU and LDST decode blocks in each lane include counters for synchronization across different lanes, and the counter values are initialized based on the VL field contained in vector instructions.

Since each vector instruction contains its own VL information, the VP no longer needs to keep the VL state. Vector instructions with different VLs can coexist in the VP lanes, making the VP extremely flexible in handling applications with different VLs.

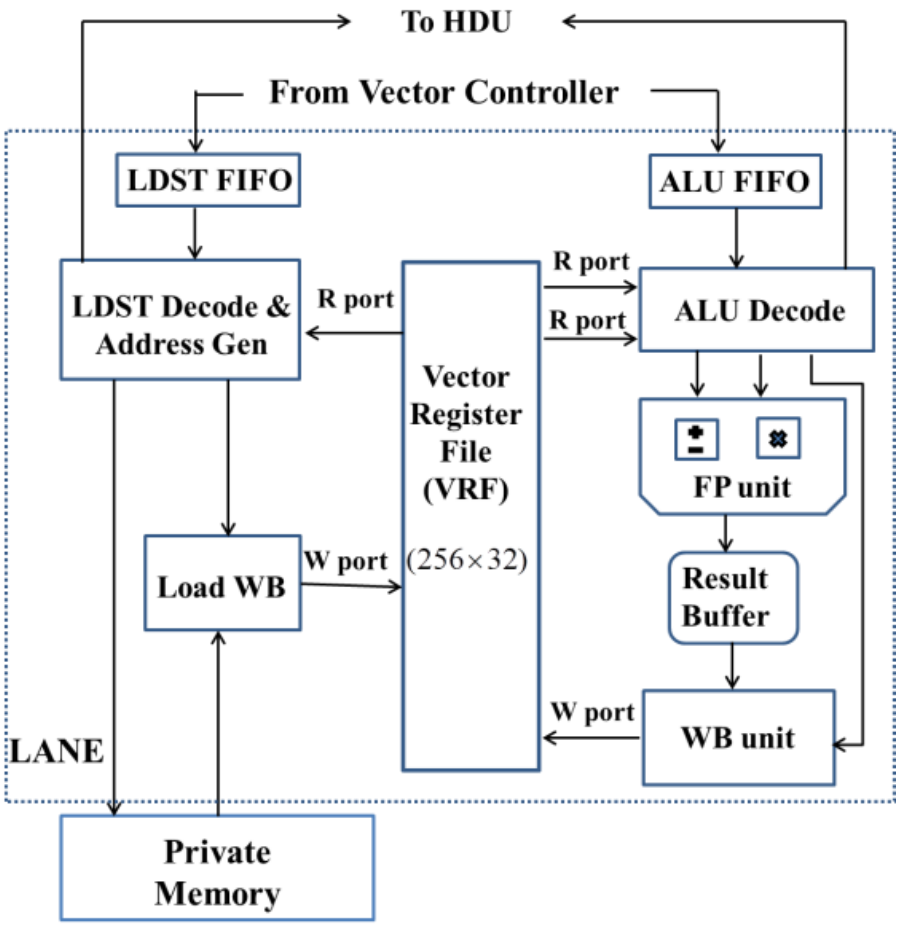

Figure 3.12 Vector lane architecture for the multiple hosts system.

3.4.4 Resource Utilization in the Multiple Hosts System

The multiple hosts system is prototyped on a Xilinx Virtex6 xc6vlx240t FPGA device. The entire VP, the vector instruction arbitrator, and the TLT are custom designed in VHDL. The rest of the system is constructed by connecting various IP cores provided in the Xilinx tool chain. The system is fully synthesized and routed, and the FPGA resource consumption

is shown in Table 3.2. The FPGA device 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 DSP slices (DSP48E1) contain a hardwired 25x18 two's complement multiplier/accumulator. The VP's FPUs are designed with custom logic for ASIC implementation, and therefore do not employ DSP slices. Only four DSP48E1s are used in the VP, one for each vector lane's synchronization counter. The entire 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 relies somewhat on the randomness of the routing process. Some registers and LUTs are simply used as wires and buffers to reduce critical path delays. Therefore, the actual minimum amount of resources required to implement the system is lower than that in the table.

The entire design flow relies on the Xilinx ISE design suite. For simulation efficiency, all performance results presented in Chapter 5 and Chapter 6 are based on cycle accurate behavioral system simulation. For highly accurate power measurements, the post-place-and-route simulation was performed on the VP at a fine detail, down to the switching of individual LUTs. The binaries for each benchmark were generated and used as testbenches to obtain Switching Activity Interchange Format (SAIF) files, which were used by the Xpower Analyzer to calculate the accurate power consumption.

Table 3.2 Resource Consumption for Multiple Hosts System.

Entity	Slice Registers (%Utilization)	Slice LUTs (% Utilization)	RAMB36E1s (% Utilization)	DSP48E1s (% Utilization)
1 Vector Lane (ALU+LDST+VRF)	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 4

BENCHMARKING

4.1 Benchmark Suite for the Single Host System

The FPGA-based simulation testbench was built using the Xilinx Project Navigator for the single host system. The chosen working frequency of 50 MHz for the VP is the result of the open source codes used to implement the ALU's FPU. However, critical path delay analysis shows that the VP's clock cycle could become as low as 7.01 ns (i.e., a frequency of 142.65 MHz) corresponding to the path delay in 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. A 50-MHz frequency was thus chosen for the MB and all the peripherals (e.g., memories, memory controllers and VP).

4.1.1 Vector Instruction in the Single Host System

Various vector-intensive benchmarks were employed to evaluate the developed design. Since MB is a soft core processor, the simulation of the executable file is performed using the developed RTL model. By performing behavioral simulation, all the ports, signals and memories in the system can be accessed. All the system components are integrated using the ISE project navigator and all the connections are made according to the architecture described in the previous chapter. The designed hardware is exported to the SDK tool so that the execution of developed application benchmarks can be driven by the scalar processor. The inline embedded macros for the VP are hand-coded to maximize its performance. Also, the drivers for the vector and shuffle controllers are developed manually using inline assembly coding. For a fair comparison with code run exclusively

on the MB, the MB's data and instruction caches are attached to the AXI4 memory; the time taken to transfer, via the DMA engine or the scalar processor, data from an external memory, such as DDR, to the data cache and private memories is the same since all connect to the same shared bus, and use the same clock signal and protocol. However, DMA is much faster in the burst transfer mode and should be used for preloading memories.

Although the time taken by data transfers in the performance comparison between the MB and the VP is excluded, the extra time is counted when data is moved between the cache and private memories. Since the private memories are connected independently to the AXI4 interconnect, all of them are accessible and addressable by both the MB and the DMA engine using low-order interleaving. Both the MB and DMA controller view the four private memories as a big vector memory with a single base address. Low-order interleaving is realized by the MUX block. The MB has access to all locations in the vector memory using its base address and the appropriate offset each time. In this prototype, each private memory is 16 Kbytes, so 64 Kbytes of vector memory are available to store application data.

Two distinct types of vector instructions, without (type 1) and with address (type 2) inclusion, are embedded in the C code run by the MB (i.e., using inline macro calls). The MB runs them as one or two store word (SW) instructions, respectively, targeting the VP's memory-mapped interface. Figure 4.1 shows how type 1 and type 2 instructions are defined using C functions, and how they are used to create macros that represent VP instructions. The type 1 __ADD instruction only needs 32 bits, whereas the type 2 __VLD (unit-stride load) and __VST (unit-stride store) instructions also carry a 32-bit address. The C code in Figure 4.1 loads two 16-element vectors from the VM and stores the summation

result back in the VM. The C structure that defines a vector contains two unsigned integer fields representing the vector's VL and a pointer to its first element in the VM. The latter field actually contains the offset of the first element which must be added to the base address of the VM.

To compile each benchmark written in the C language and containing inline macros for the VP, the MB GNU mb-gcc tool is applied twice, with and without optimization, respectively. Maximum optimization (O3) is invoked that involves inline functioning, loop unrolling and strict aliasing. This optimization causes code rearrangement in order to increase the rate of issuing vector instructions. Eight benchmark algorithms with three alternatives each for the VL value are tested. Therefore, results for 24 benchmark instantiations are presented for single host system. All the benchmarks are developed using the VP's instruction set architecture (ISA) shown in Figure 4.2.

4.1.2 Benchmark Applications for Single Host System

The first benchmark is the multiplication of square matrices with size 16*16, 32*32 and 64*64. Three benchmark algorithms are developed for matrix multiplication to run on single host system. Algorithms 1 and 2 calculate one element of the resulting matrix in each loop iteration. Only the location of additions differs between these two algorithms (details follow in the Section 5.1.1). Algorithm 3 improves the vectorization ratio (i.e., ratio of vector to scalar code) since all the elements of a row in the resulting matrix are calculated in each loop iteration.

For the sake of comparison, sequential C code for the MB scalar processor is also developed for matrix multiplication that represents the same number of operations with each of these algorithms. Two benchmark algorithms implement FIR (Finite Impulse

```
//functions to define two different kinds of instructions(with and without data)
#define V_instr_a(instr)  asm volatile("sw\t%0,r0"::"d"(instr), "d"(SCHEDULER_ADDRESS));

#define V_instr_b(instr,addr)  asm volatile("sw\t%0,r0""sw\t%1,r0"::"d"(instr),"d"(addr), \
"d"(SCHEDULER_ADDRESS));


//using defined instructions to define vector instructions as macros, X_SHIFT constant
//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(){
__VLD(0,adr1,16,0);     //loading from location adr1 to r0 for VL=16 and thread 0
__VLD(1,adr2,16,0);     //loading from location adr2 to r1 for VL=16 and thread 0
__VADD(2,0,1,16,0);     //r2<=r0+r1 for VL=16 and thread 0
__VST(2,adr3,16,0);     //storing from r2 to location adr3 for VL=16 and thread 0
};
```

Figure 4.1 Example of C code showing VP instructions implemented as macro calls for vector addition in single host system.

Target	Instruction	Description
Scheduler	__VpReq	Request to access VP
	__VpRel	Request to release VP
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)
Shuffle Engines	__VSHF	Vector shuffle
	__VLD_I	Vector load (index addressing)
	__VST_I	Vector store (index addressing)

Figure 4.2 ISA of the VP.

Response) digital filtering and use the outer product [Sung et al., 1987]. 16, 32 and 64 tap FIR filters are realized. One of these benchmarks (Algorithm 2, which is presented below), applies special memory initialization to maximize the vectorization ratio in a way that takes advantage of unrolling the loop four times. In this method the coefficient window slides 4 times over input sequence instead of once in every iteration. This was realized by initializing four copies of the input sequence in the VM with each first element of the sequence located in a different memory bank.

The next two benchmark algorithms implement FFT using a 16, 32 and 64 point decimation-in-time radix-2 butterfly algorithm [Cooley et al., 1965]. Shuffle instructions are executed in each stage. In each benchmark execution, the results of performing data shuffling by the scalar processor and the shuffle engine, respectively, are observed and compared. More details about the FIR and FFT benchmark variations follow in the Section 5.1.1.

The last benchmark is RGB2YIQ (RGB to YIQ color space) mapping. This benchmark, which is the most vectorizable, is run on 16*16, 32*32 and 64*64 pixel matrices. All these benchmark instantiations were also implemented on the MB using sequential code.

4.2 Benchmark Suite for the Multiple Hosts System

Various vector-intensive benchmarks were employed to evaluate the design in the multiple hosts system. Some benchmarks are already applied and explained in the single host system. Here the 100 MHz working frequency is selected for running benchmarks. However, as the single host system critical path delay analysis shows, the VP's clock cycle

could be as low as 7.01 ns (i.e., representing 142.65 MHz), corresponding to the path delay in the adder.

Although different frequencies are applied to benchmark the single host and multiple hosts systems, this may not cause any concern in the presentation of results due to the following reasons:

1. The only result which depends on the frequency is the execution time which, however, can be easily converted to a cycle count result for better evaluation.

2. The VP utilization in each scenario is completely independent of the working frequency.

3. The comparisons for both systems are based on the speedup achieved, which is independent of the working frequency.

4. Power and energy analysis for each benchmark is performed on the desired frequency and presented accordingly.

4.2.1 VP Instruction and Compilation for the Multiple Hosts System

As shown in Figure 4.3, two basic types of vector instructions are sent to the 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 4.3 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. It can be seen from the figure that different MB assembly instructions, namely "*put*" and "*cput*", are used here to develop vector instruction macros rather than "SW" which was the case in the single host system. This is because here the VP connects to the stream interface of the MB while in the single host system it was placed on the AXI4 shared bus. Since the AXI4 interconnect forms a memory mapped architecture, the store instruction was required to send data to the VP. The main function in Figure 4.3 loads two 16-element vectors from the VM and stores

the summation result back into the VM. To compile benchmarks, written in the C language, that contain macros and assembly code for vector instructions, the MB GNU mb-gcc tool without optimization (i.e., option o0) was applied. Five algorithms were evaluated on the multiple hosts system, each with three alternative VLs, for a total of fifteen distinct benchmarks.

```
// functions to define two different kinds of instructions(with and without 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))


//using defined instructions to define vector instructions as macros, X_SHIFT constant
//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(){
__VLD(0,adr1,16,0);    //loading from location adr1 to r0 for VL=16 and thread 0
__VLD(1,adr2,16,0);    //loading from location adr2 to r1 for VL=16 and thread 0
__VADD(2,0,1,16,0);    //r2<=r0+r1 for VL=16 and thread 0
__VST(2,adr3,16,0);    //storing from r2 to location adr3 for VL=16 and thread 0
};
```

Figure 4.3 Macros to define vector instructions in multiple hosts system.

The complete ISA of the VP, including all vector instructions as well as the control instructions developed for VP virtualization, are listed in Figure 4.2. Previously in the single host system discussed in Section 3.1.1 the __VP_REQ and __VP_REL instructions are sent to the hardware scheduler and a response is received accordingly by reading the scheduler handshake response from its memory mapped location. In the multiple hosts

system, these two instructions were implemented by software since the real scheduler in this system is the system core (MB0 in Figure 3.5) as described in Section 3.2.1. The control instruction __VP_REQ is implemented as a C function which takes the 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 a parameter the thread ID and releases all the 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 use it as the ID field for every VP instruction. When the application is completed, VP resources must be released using a __VP_REL call.

4.2.2 Benchmark Applications for the Multiple Hosts System

The first benchmark is matrix multiplication (MM) for square matrices of size 16*16, 32*32 and 64*64. Here only algorithm 3 for MM from the single host benchmarks is implemented. In this application, all elements on a row of the resulting matrix are calculated in each 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 (element 1 with row 1, element 2 with row 2, and so on) and appropriate partial products are summed up. All multiplications are performed using scalar-vector multiplication instructions, and additions are of the vector-vector type. Using an optimal approach, only two vector registers of size VL are needed in this benchmark. The results show that by increasing the dimensionality of the matrix and consequently the VL, the time needed to

generate one element in the product matrix decreases slightly (due to higher vectorization ratio).

The second benchmark is Finite Impulse Response (FIR) digital filter that uses the outer product. 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 length. This is algorithm 2 for the FIR benchmark which was discussed in terms of the single host system. A loop unrolling technique was used to expand the kernel four times and increase the vectorization ratio. This benchmark uses two vector registers of size VL.

The third benchmark is vector-dot product (VDP) with VL= 16, 32, and 64. A vector-vector multiplication instruction is followed by a couple of vector-vector addition instructions. Four VL-sized vector registers are used. The execution time of VDP is measured for an input of five pairs of arrays 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 to fixed-sized pixel blocks, like 8*8 or 4*4, following the same principle one-dimensional 8-point DCT on blocks of size 8*8 is performed. 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 last benchmark is RGB to YIQ color space mapping (RGB2YIQ). It has the highest portion of vector code among all five benchmarks and uses seven vector registers. The configurations of VL=16, 32, and 64 is used to perform the calculation on a 1024-pixel block. Since the input size is independent of VL, higher VLs lead to fewer loop iterations, and therefore shorter execution times.

CHAPTER 5

PERFORMANCE ANALYSIS

5.1 Single Host System Performance Analysis

In this chapter, performance results are presented for the benchmarking performed on the single host system architecture of Subsection 5.1.1. Detailed analysis for performance exploration focusing on ideal and practical systems is given in Subsection 5.1.2. A comprehensive comparison with other VPs benchmarked for the same applications is presented in Subsection 5.1.3. As discussed earlier in Section 4.2, the simulation frequency for both the VP and MB systems in the single host architecture is 50MHz. Hence, all the simulation results presented in this section are based on this frequency.

5.1.1 Simulation Results and Performance Analysis in the Single Host System

Table 5.1.a and b show results under various execution scenarios for matrix multiplication without compiler optimization and with maximum compiler optimization, respectively. The DMA is used to transfer input data to vector memory. For the sake of simplicity, the time taken in each case for producing one element in the resulting matrix is shown. Matrix multiplication Algorithms 1 and 2 multiply a row with a column and add the partial results. Both algorithms use the VP for multiplications. Algorithm 1 uses the MB for the addition of partial products whereas Algorithm 2 uses both the VP and the MB for this purpose. More specifically, partial products are loaded in four vector registers and vector-vector additions are then applied, which are followed by VL/4 MB additions to produce each element in the resulting matrix. Algorithm 3 uses a different technique that produces a single resulting row at a time. More specifically, the MB is only in charge of vector-

instruction control flow; all additions and multiplications are done by the VP. This algorithm multiplies a single element of the first matrix with all the 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 by the respective row in the second matrix (element 1 with row 1, element 2 with row 2, and so on) and appropriate partial products are summed up. All the multiplications are performed using scalar-vector multiplication instructions and additions are carried out via vector-vector additions.

Table 5.1 Performance Comparison for Three Multiplication Algorithms and Various VLs on the Single Host System. Algorithms 1 and 2 Use Both the VP and MB. Algorithm 3 Uses Only the VP. The Execution Time is Shown for Each Element Produced in the Product Matrix. (a) Without Compiler Optimization and (b) with Compiler Optimization.

Matrix Size Vector Length		Algorithm 1 Execution Time(us) (VP+MB)	Algorithm 2 Execution Time(us) (VP+MB)	Algorithm 3 Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
16*16	VL=16	75	28	4.5	160
32*32	VL=32	144	31	4.36	319
64*64	VL=64	279	34	4.32	630

(a)

Matrix Size Vector Length		Algorithm 1 Execution Time(us) (VP+MB)	Algorithm 2 Execution Time(us) (VP+MB)	Algorithm 3 Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
16*16	VL=16	69	21	1.5	139
32*32	VL=32	136	22	1.43	278
64*64	VL=64	268	23	1.368	552

(b)

The results show that by increasing the dimensionality of the matrix and consequently the VL, the time needed to generate one element in the product matrix increases for the element-wise Algorithms 1 and 2 while it decreases slightly for the row-wise Algorithm 3 (due to higher vectorization ratio). Compiler optimization demonstrates the most dramatic beneficial impact on the algorithm that uses the VP the most; this is because the improvement increases faster with the matrix size.

Table 5.2.a and b show performance results for FIR filtering under various scenarios and VLs without and with compiler optimization, respectively. 16, 32 and 64 tap FIR filters are implemented with the size of the input sequence being the same as the size of the filter; therefore, the resulting sequence has double the length of the input. The MB is used to transfer data between the global and vector memories. Two algorithms are developed. Algorithm 1 has no loop unrolling while Algorithm 2 uses special initialization of the vector memory customized for the four lanes by unrolling the loop four times to achieve higher VP utilization. The results in the table include the time to initialize the private memory by transferring data from the global memory. Similar to matrix multiplication, the effect of compiler optimization is more prominent when using only the VP.

Table 5.2 Performance Comparison for FIR Filtering with Various Filter Sizes in the Single Host System. The Times for Data Exchanges Between the Global and Private Memories are Included. The Times are for Calculating All the Output Elements. (a) Without Compiler Optimization and (b) with Compiler Optimization.

Filter Size Vector Length		Input Length	Output Length	Algorithm 1 Execution Time(us) (VP+MB)	Algorithm 2 Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
16 Tap	VL=16	16	32	414	149	4250
32 Tap	VL=32	32	64	1439	278	15615
64 Tap	VL=64	64	128	5331	536	68703

(a)

Filter Size Vector Length		Input Length	Output Length	Algorithm 1 Execution Time(us) (VP+MB)	Algorithm 2 Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
16 Tap	VL=16	16	32	114	52	3813
32 Tap	VL=32	32	64	370	94	14189
64 Tap	VL=64	64	128	1312	178	64102

(b)

Table 5.3.a and b depict the results for FFT without and with compiler optimization, respectively. 16, 32 and 64 point FFT are implemented using two algorithms. MB is in

charge of transferring input and output data between the global and vector memories. In Algorithm 1, the shuffling of data in the vector memory is realized by the MB via the AXI4 interconnect. In Algorithm 2, the distributed data shuffle engines implement shuffling needed in each stage of FFT. As intended, the data shuffle engines which are distributed across the vector lanes have a spectacular beneficial impact on FFT's execution time due to its hefty demand of data shuffling. The relevant speedup is 5.92 and 7.33 for the 64-point FFT without and with compiler optimization, respectively. Furthermore, the 64-point FFT speedup of Algorithm 2 against runs on the MB is 52.07 and 110.45 without and with compiler optimization, respectively. Also, similar to runs of the other benchmarks, the impact of compiler optimization becomes more prominent when the VP utilization increases. In general, the performance advances of our architecture become more manifested with increased VLs in applications.

Table 5.3 Performance Comparison for FFT of Various Sizes in the Single Host System. The Execution Time Includes the Overhead of Writing and Reading Between the Global and Vector Memories. The Numbers are for Calculating All the Output Results. (a) Without Compiler Optimization and (b) with Compiler Optimization.

FFT Size Vector Length		**Algorithm 1 Execution Time(us) (VP+MB)**	**Algorithm 2 Execution Time(us) (VP)**	**Scalar Execution Time(us) (MB)**
16 Point	VL=16	386	132	3850
32 Point	VL=32	814	190	7380
64 Point	VL=64	1783	301	15673

(a)

FFT Size Vector Length		**Algorithm 1 Execution Time(us) (VP+MB)**	**Algorithm 2 Execution Time(us) (VP)**	**Scalar Execution Time(us) (MB)**
16 Point	VL=16	175	42	2520
32 Point	VL=32	379	63	5605
64 Point	VL=64	762	104	11487

(b)

Table 5.4.a and b show performance results for the RGB2YIO benchmark without and with compiler optimization, respectively. DMA is used to move information about pixels to the vector memory. Although three sizes are chosen for the input pixel array, the results in the table are for calculating the values in a block of 8*8 pixels in the YIQ target space. Since each output pixel value depends only on the corresponding input pixel value, the MB time consumed for calculating an 8*8 block is the same in all three cases. Since this benchmark is highly vectorizable, the speedup of the VP over the MB is huge. Obviously, it increases even further via compiler optimization.

Table 5.4 Performance Comparison for RGB2YIQ with Various VLs in the Single Host System. The Time is for Calculating a Block of 8*8 Pixels in the YIQ Color Space. (a) Without Compiler Optimization and (b) with Compiler Optimization.

Vector Length	VP Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
VL=16	63.25	10932
VL=32	31.66	10932
VL=64	16.76	10932

(a)

Vector Length	VP Execution Time(us) (VP)	Scalar Execution Time(us) (MB)
VL=16	27.58	10572
VL=32	13.87	10572
VL=64	6.99	10572

(b)

To further underscore the need of the VP coprocessor, Figure 5.1.a shows its speedup compared to MB execution for matrix multiplication using Algorithm 3. The VP achieves a x400 speedup with VL=64 and compiler optimization. A main performance bottleneck in this testbench is the low rate of issuing vector instructions to the VP. Therefore, without optimization the VP is not fully utilized since the time between issuing

vector instructions to the VP is larger than the time needed for a vector instruction to be implemented. Using inline assembly-language macros for vector instructions rather than HLL function calls, this difference is reduced as much as possible. There are also two other approaches needed to be followed in the effort to minimize VP idle times. First, increasing the VL of the application algorithm will keep the VP busy for a longer time, hopefully till the next vector instruction is issued.

The second approach is to increase the vector instruction issue rate for the VP by applying code optimization. Both of these methods result in increasing the VP utilization when it is exclusively attached to a scalar core in the single host system. As it will be seen in the multiple hosts system result, VP utilization could also be increased through VP virtualization and share VP resources between different threads. Figure 5.1.b shows the speedup of the VP+MB system versus the MB for matrix multiplication under Algorithm 2. Two main differences can be observed between the two parts of Figure 5.1. The rate of speedup improvement for Algorithm 3 with an increasing VL is much higher, which implies a higher VP utilization. Also, the effect of compiler optimization is lower for Algorithm 2 because only part of the application is run on the VP.

Figure 5.2 depicts the speedup for FIR filtering under various filter tap sizes and VLs. Figure 5.2.a presents the speedup of the VP versus the MB for Algorithm 2. For 64 taps with compiler optimization, the speedup is higher than 350 and increases drastically when the VL increases due to higher vectorization ratios. Initializing the private memories of the four lanes helps this process.

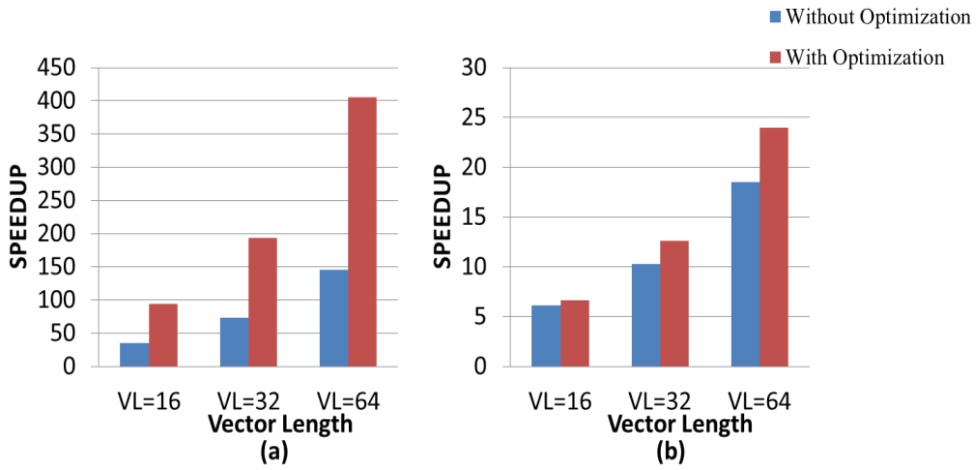

Figure 5.1 Speedup for matrix multiplication with and without optimization. (a) VP vs. MB for Algorithm 3 and (b) VP+MB vs. MB for Algorithm 2.

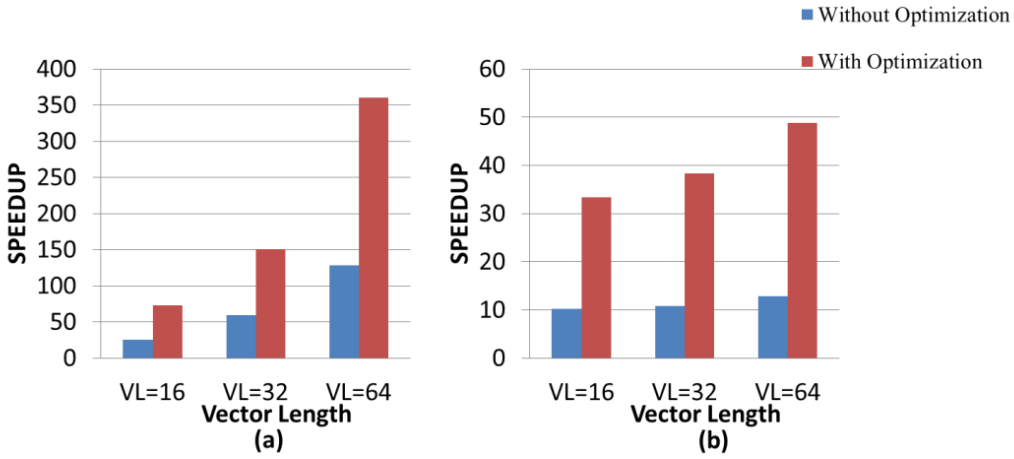

Figure 5.2 VP vs. MB speedup for FIR filtering with and without optimization. (a) Algorithm 2 and (b) Algorithm 1.

Figure 5.2.b shows the speedup for Algorithm 1. Without optimization, the speedup does not keep up with VL increases since this algorithm has a lower vectorization ratio; the MB is involved in each iteration that produces a new element of the result. However, compiler optimization increases the portion of the algorithm that runs on the VP which, in turn, improves acceleration.

Figure 5.3.a and b show the FFT speedup for various VLs, with and without the distributed data shuffle engines, respectively. The VP with the data shuffle engines and code optimization achieves a 110-fold speedup over the MB for the 64-point FFT. For Figure 5.3.b, the data shuffle instructions are implemented by the MB instead of the VP. Without the shuffle engine and without code optimization, the speedup degrades slightly with an increasing VL. Since data shuffling is the most time consuming process in each stage of FFT, performing it on the MB with an increased VL results in slight performance

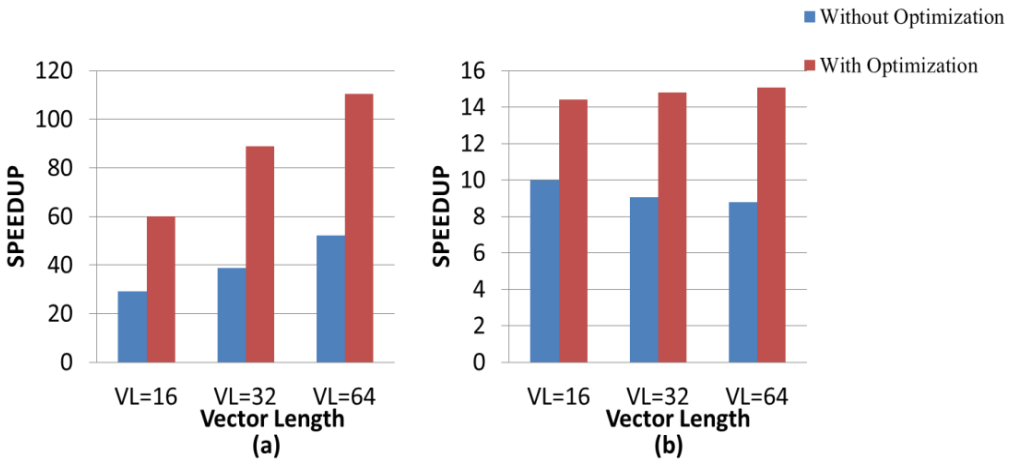

Figure 5.3 Speedup for FFT. (a) VP with the data shuffle engine vs. MB for Algorithm 2 and (b) VP+MB without the shuffle engine vs. MB for Algorithm 1.

degradation; however, compiler optimization can compensate by increasing the portion run on the VP.

Figure 5.4 shows the speedup for the highly vectorizable RGB2YIQ benchmark. The VP achieves an impressive 1500-fold speedup over the MB for VL=64 and with compiler optimization. This benchmark approaches the peak performance of the VP since most of the time the VP is fully utilized without waiting for a new vector instruction to be issued.

For the sake of comprehensive analysis, Figure 5.5 shows the performance/area ratio of the VP over MB execution of the benchmarks in the single host system for three VL alternatives, assuming maximum compiler optimization. The benchmarking shows that the VP supports scalability since the ratio generally improves with increases in the VL. Actually, the improvement is faster for a reduced number of data dependencies (i.e., RGB2YIQ), and slower or negligible for a very large number of data dependencies (i.e., FFT).

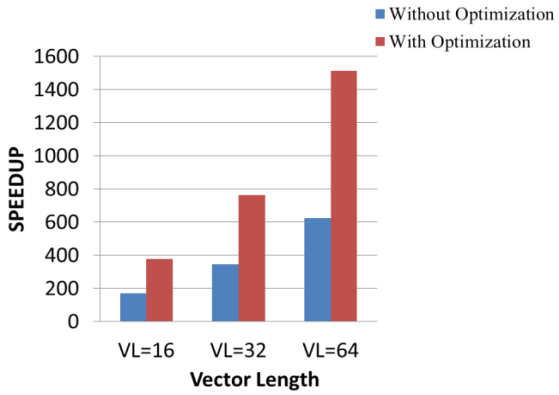

Figure 5.4 VP vs. MB speedup for RGB2YIQ in single host system.

5.1.2 Performance Exploration in the Single Host System

For a fair performance comparison with earlier works involving VP designs, it is needed to count the execution time of applications in number of clock cycles (e.g., since different target FPGAs used in prototyping support different clock frequencies, etc.). The main bottleneck in benchmarking is the MB core that adds delays to the process of issuing vector instructions to the VP; this decreases the utilization of the VP due to a lesser average density of vector instructions in each lane's ALU and LDST FIFOs. Compiler optimization may ease this problem depending on the application, as discussed in Section 5.1.1. For fair

VP comparisons independent of compilers and scalar cores, maximum VP performance should be targeted. Therefore, all the vector instructions in this section are placed in advance in the VC queue instead of being issued by the MB as encountered in the application code. Also, all private memories are initialized with the needed application data. Thus, the clock cycles really taken by the applications on the VP are counted.

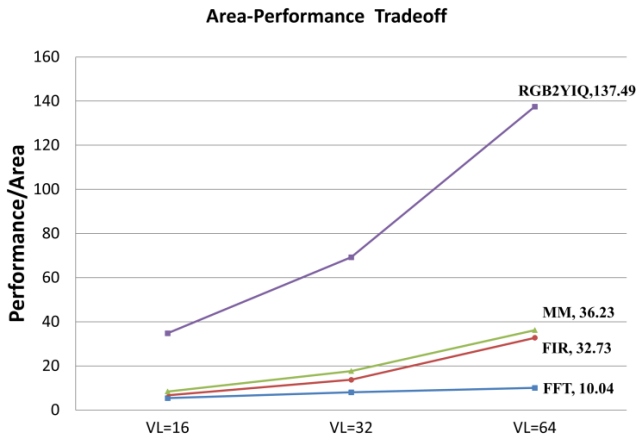

Figure 5.5 Performance/Area improvement for the VP over the MB in single host system.

To find the minimum number of clock cycles for executing each benchmark, its total number of instructions is calculated. For accuracy, the VP behavior in the case of hazard detection is taken into consideration. Whenever a data hazard is detected by the VC, issuing instructions to the FIFO is stalled and demand for a new vector instruction (VI) is delayed until the corresponding instruction is committed and the pipeline is back to normal operation. Another important issue is the capability of overlapping a data shuffle instruction with subsequent instructions as long as no data hazard is present.

Table 5.5 to Table 5.8 show performance results for executing each benchmark on the VP. “Practical” times are obtained from the already presented results by excluding the

times needed to transfer data between the global and private memories. “Ideal” times are obtained by removing any MB delay in issuing instructions to the VP. “Ideal without private memories” times are similar to ideal but, instead of having a private memory in each lane, each lane has access to all memory banks in the vector memory using a crossbar that connects lanes to memories (similar to the architecture [Beldianu et al., 2013]). Under the worst case scenario for vector load and store instructions, only one element per clock cycle can be transferred between the lanes and the vector memory. This, however, is the best case for the VP architecture due to the presence of the private memories that can transfer four elements per clock cycle.

Table 5.5 Matrix Multiplication Performance Comparison for Various VLs in the Single Host System.

Vector Length	**TI***	**EFPI****	**#of cycles**	**Stall Rate (%)**	**#of cycles per VI*****	**Average Utilization (%)**		
						LDST	**ALU**	
VL=16	784	10.44	6893	0	8.79	15.8	29.7	**Ideal**
VL=32	3104	21.11	40461	0	13.05	20.9	42.2	
VL=64	12352	42.44	252441	0	20.48	26.3	51.7	
VL=16	784	10.44	10157	32	12.95	10.7	20.2	**Ideal without private memory**
VL=32	3104	21.11	64845	35	20.89	13.1	26.3	
VL=64	12352	42.44	455309	44	36.86	14.58	28.6	
VL=16	784	10.44	19200	0	24.48	5.7	11.3	**Practical**
VL=32	3104	21.11	73216	0	23.58	11.3	22.3	
VL=64	12352	42.44	280166	0	22.68	23.7	46.6	

*TI=Total Instructions ***VI: Vector Instruction
**EFPI=Effective FLOPs Per Instruction

“Total instructions” represents the number of vector instructions issued by the MB, considering all loop iterations. The effective FLOPS per instruction are obtained by dividing the total number of ALU floating-point operations in the benchmark by the total number of instructions. The average number of clock cycles needed per instruction is then

Table 5.6 FIR Performance Comparison for Various VLs in the Single Host System.

Vector Length	TI	EFPI	#of cycles	Stall Rate (%)	#of cycles per VI	Average Utilization (%)		
						LDST	ALU	
VL=16	100	10.24	463	0	4.63	29.3	57.1	**Ideal**
VL=32	196	20.89	1419	0	7.23	37.1	73.2	
VL=64	388	42.22	5341	0	13.76	39.1	77.3	
VL=16	100	10.24	931	50	9.31	14.6	28.4	**Ideal without private memory**
VL=32	196	20.89	3003	52	15.32	17.5	34.6	
VL=64	388	42.22	11691	54	30.13	17.9	35.3	
VL=16	100	10.24	1450	0	14.5	9.3	18.2	**Practical**
VL=32	196	20.89	2850	0	14.54	18.53	36.5	
VL=64	388	42.22	5750	0	14.82	36.1	71.8	

Table 5.7 FFT Performance Comparison for Various VLs in the Single Host System.

Vector Length	TI	EFPI	#of cycles	Stall Rate (%)	#of cycles per VI	Average Utilization (%)			
						LDST	ALU	SHF	
VL=16	64	10	396	0	6.18	16	40.4	24	**Ideal**
VL=32	80	20	956	0	11.95	16.7	41.7	25.1	
VL=64	96	40	2280	0	23.75	16.8	42	25.2	
VL=16	64	10	672	40	10.5	9.4	23.8	14.1	**Ideal without private memory**
VL=32	80	20	1611	40	20.14	9.9	24.7	14.9	
VL=64	96	40	3820	40	39.79	10	25.1	15	
VL=16	64	10	1300	0	20.31	4.7	12.3	7.4	**Practical**
VL=32	80	20	1550	0	19.37	10.3	26	15.4	
VL=64	96	40	2500	0	26.04	15.4	38.4	23.1	

Table 5.8 RGB2YIQ Performance Comparison for Various VLs in the Single Host System.

Vector Length	TI	EFPI	#of cycles	Stall Rate (%)	#of cycles per VI	Average Utilization (%)		
						LDST	ALU	
VL=16	336	11.42	1437	0	4.27	26.7	66.8	**Ideal**
VL=32	672	22.85	5165	0	7.68	29.7	74.3	
VL=64	1344	45.71	19533	0	14.53	31.4	78.6	
VL=16	336	11.42	2493	42	7.42	15.4	38.5	**Ideal without private memory**
VL=32	672	22.85	9581	46	14.25	16	40	
VL=64	1344	45.71	37581	48	27.96	16.3	42.9	
VL=16	336	11.42	5500	0	16.36	6.9	17.5	**Practical**
VL=32	672	22.85	11100	0	16.51	13.8	34.6	
VL=64	1344	45.71	22400	0	16.66	27.4	68.6	

calculated for each benchmark. The LDST average utilization of a lane shows the number of vector elements sent to or received from the vector memory in 100 clock cycles. The ALU average utilization represents the number of elements produced by a lane's ALU in 100 clock cycles. The SHF utilization in Table 5.7 for the FFT shows the number of elements transferred by the shuffle engines in 100 clock cycles. It can be concluded that increasing the VL brings the practical time closer to the ideal one due to higher VP utilization and less idling between instructions issued by the MB.

5.1.3 Comparison with Prior Work

The presented 4-lane VP results are compared with those in [Beldianu et al., 2013] that assumed a VIRAM-like VP (like us) with eight lanes. A crossbar in the latter design connects the lanes to the global vector memory, and is also used to realize data shuffle and indexed memory instructions. Due to the lack of private memories, there is a high chance of stalls for memory instructions. They used a Xilinx block memory IP (BRAM) for the VRF and their total VRF capacity was 2 Kbytes/lane (ours is 1 Kbyte/lane). Since they used Xilinx IPs to build lane ALUs, many embedded DSP blocks were consumed. The Fast Simplex Link (FSL) point-to-point interface was employed to connect the VP with the MB. Each of the two MBs in their design uses its own VC and FSL slave and master interfaces to access the VP. To allow the two MB cores to share the VP, three scheduling policies were implemented for their scheduler. In coarse-grain temporal sharing (CTS), the two cores time share the entire VP. In fine-grain temporal sharing (FTS), both cores compete simultaneously for all the VP resources. This is equivalent to SMT with respect to VP usage and the two core threads use different vector registers. Vector lane sharing (VLS), finally, assigns distinct lanes to each MB upon demand, as decided by the VC.

As per Table 5.9, the obtained speedup for the FFT benchmark is smaller than that for FIR filtering and matrix multiplication since the former requires heavy data shuffling. However, the speedups are always higher than those in [Beldianu et al., 2013]. This observation becomes even more impressive considering that proposed VP platform here has four, instead of eight, lanes and one core, instead of two. For the FIR benchmark, the speedups of the VP and SODA is compared in reference to their individual host processors (i.e., MB and Alpha, respectively). SODA achieves speedups of up to 19 and 26 for 33- and 65-tap filters, respectively. The developed VP accomplishes much higher speedups of 150 and 350 for the 32- and 64-tap filters, respectively.

Also, there exist comparative results of FPGA and ASIC realizations involving various designs that make it possible to estimate the relative speedup and improved power dissipation, within an order of magnitude, when an FPGA-based design is moved into the ASIC realm (e.g.,[Beldianu et al., 2015][Kuon et al., 2007][Suresh at al., 2013]). However, an ASIC implementation of our design, a rather hectic and lengthy process, will be a future research objective.

Table 5.9 Speedups of the VP in Single Host System and the Design in [Beldianu et al., 2013] vs. the MB for VL=32.

Architecture	Matrix Multiplication	FIR	FFT
CTS [Beldianu et al., 2013], 8 lanes, 1 core	12.97	10.93	49.02
FTS [Beldianu et al., 2013] 8 lanes, 2 cores	25.89	21.83	86.76
VP in single host system, 4 lanes, 1 core	193.5	150.94	88.98

5.2 Multiple Hosts System Performance Analysis

All the simulation results presented in this section are based on a working frequency of 100MHz for all the MBs and the VP.

5.2.1 Simulation Results

In this section only, it is assumed that the VP runs simultaneously each time 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 VP has 16 registers of VL=64; it is assumed up to two threads 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 our 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. Table 5.10 to Table 5.14 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 in Section 4.2.2.

In this section, all simultaneous threads of the same application are homogeneous, but independent, and their flow control codes are executed on different MBs. All threads operate on different input data sets to increase the throughput.

In Chapter 6, the VP's simultaneous execution of heterogeneous threads with different VLs, coming from different MBs, and the implementation of different algorithms will be discussed. The tables show that the 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 the VL increases, the utilization of a thread increases up to a saturation point. As explained earlier (Section 3.1.2), there is a high impedance state of one

Table 5.10 Matrix Multiplication Performance in the Multiple Hosts System (Input and Output 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 5.11 FIR Performance in the Multiple Hosts System (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 5.12 VDP Performance in the Multiple Hosts System (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 5.13 DCT Performance in the Multiple Hosts System (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

clock cycle between issuing successive instructions; this state may decrease 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. Before saturation, a benchmark's performance is actually upper bounded by scalar core execution for the serial part of the vector application; thus, a VP shared by many scalar cores is recommended in this case. The total execution

Table 5.14 RGB2YIQ Performance in the Multiple Hosts System (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

time with multiple threads may be the same as the benchmark's native duration (the execution time when the VP is exclusively occupied by one thread of the benchmark) if each thread has a rather low utilization. When the total VP utilization with many simultaneous threads exceeds the VP's nominal maximum, all threads' execution will slow down proportionally due to resource competition. It is also important to realize that 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 provided for multiple hosts system, it can be seen that 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 that limits further throughput increases. Upon VP saturation, the slowdown amount is determined by the higher of the ALU and LDST utilizations.

Figure 5.6 shows the maximum ALU and LDST utilizations for various benchmarks, and the VL and core numbers. Running the RGB2YIQ benchmark with VL=64 on more than two cores is impractical since each benchmark instance needs seven vector registers whereas our VP contains 16 vector registers of VL=64. According to Table 5.14, the performance of RGB2YIQ with VL=64 saturates for two cores although the ALU utilization is not close to the nominal maximum of 94%. This can happen 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 the VP versus the scalar core.

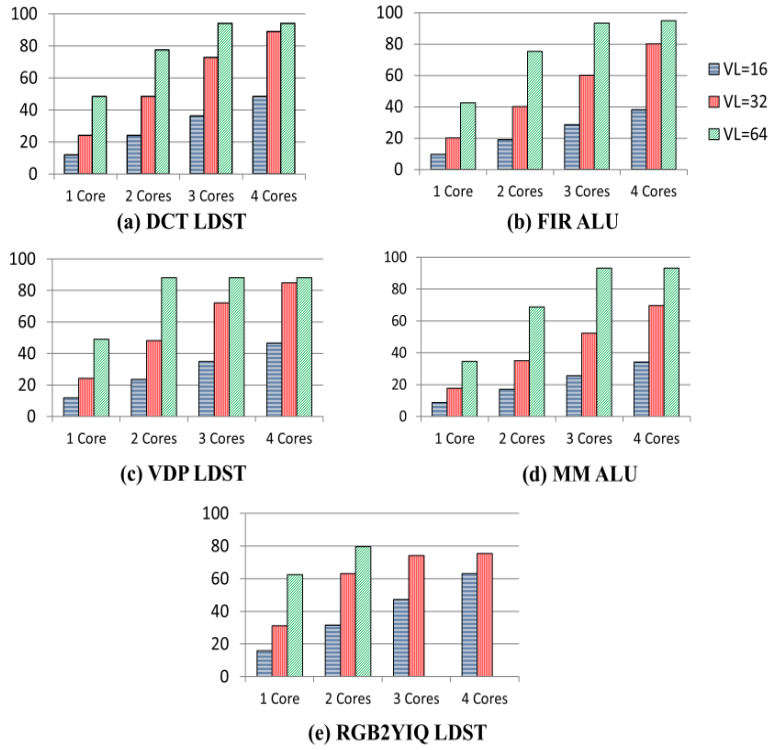

Figure 5.6 Maximum utilization of the LDST and ALU units.

5.2.2 Comparison with the Single Host System and Prior Works

To perform a fair performance comparison with the single host system and other previously published works that focused on VP sharing for multicores, a common reference point is chosen. Moreover, the chosen benchmark scenarios are similar (including the same values of VL). Since VP speedups against their host processors were listed in all these prior works, the same is applied for the multi host system. Table 5.15 shows comparisons with [Beldianu et al., 2013] that implemented an 8-lane VP shared by two scalar processors 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 VP sharing technique in the multi host system. The single host architecture [Rooholamin et al., 2015] uses a VP architecture that has many

similarities to the one applied in the multi host system. It utilizes a hardware scheduler and register renaming block to support VP sharing for two threads with identical VL; one host issues threads. It relies on compiler optimizations to increase the issue rate of vector instructions. The table shows that the proposed VP sharing technique always yields by far the best speedup compared to the single host system and others which have double the lanes.

Table 5.15 Speedup Comparison With the Single Host and Previously Shared VP .

SYSTEM \ BENCHMARK	MM	FIR	RGB2YIQ	VL
Single host system, 4 lanes,1 core	92.66	73.32	383.32	16
Multiple hosts system, 4 lanes, 4 cores	339.91	314.8	1432.53	
[Beldianu et al., 2013], CTS, 8 lanes, 1 core	12.97	10.93	NA	32
[Beldianu et al., 2013], FTS, 8 lanes, 2 cores	25.89	21.83	NA	
Single host system, 4 lanes,1 core	193.06	150.94	762.22	
Multiple hosts system, 4 lanes, 4 cores	693.53	612.31	1713.98	
Single host system, 4 lanes,1 core	403.50	360.12	1512.44	64
Multiple hosts system, 4 lanes, 4 cores	917.50	776.29	1808.80	

CHAPTER 6

SCHEDULING VECTOR THREADS

6.1 Scheduling Algorithm Implemented on the System Core

This chapter focuses on throughput-maximizing thread scheduling for a multi-host system. First, each application is profiled to determine its ALU and LDST utilizations, as well as its native duration based on the results in Section 5.2.1 (i.e., its execution time with exclusive VP access). The combinations of simultaneously executing benchmarks are evaluated from the set of 15 benchmarks (five benchmarks with three VL alternatives each) for: i) A closed system with a fixed number of threads in Section 6.2. ii) An open system with randomly arriving threads in Section 6.3.

As observed in Section 5.2.1, 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 3.1.2), the VP's saturation threshold is not 100% but a number from 80% to 94% depending on the active threads' VLs. A saturation threshold of 90% is assumed to design a scheduling algorithm that keeps the VP highly busy either with zero or minimum saturation.

In a closed system, all threads in a queue at a given time are scheduled. No new threads are added into the queue before all threads in the current queue have finished execution. The scheduler algorithm flowchart is given in Figure 6.1. Once a thread is picked by the scheduler, it keeps executing until the end, at which time its VP resources are released to any pending threads. Pending threads are arranged in descending order of

their native duration. The ALU and LDST utilizations as well as the VRF usage of pending threads are provided to the scheduler as input. The scheduler keeps picking pending threads for execution until the VP has four threads, or no other pending thread can be accommodated due to unavailable VRF resources. The scheduler searches down the queue

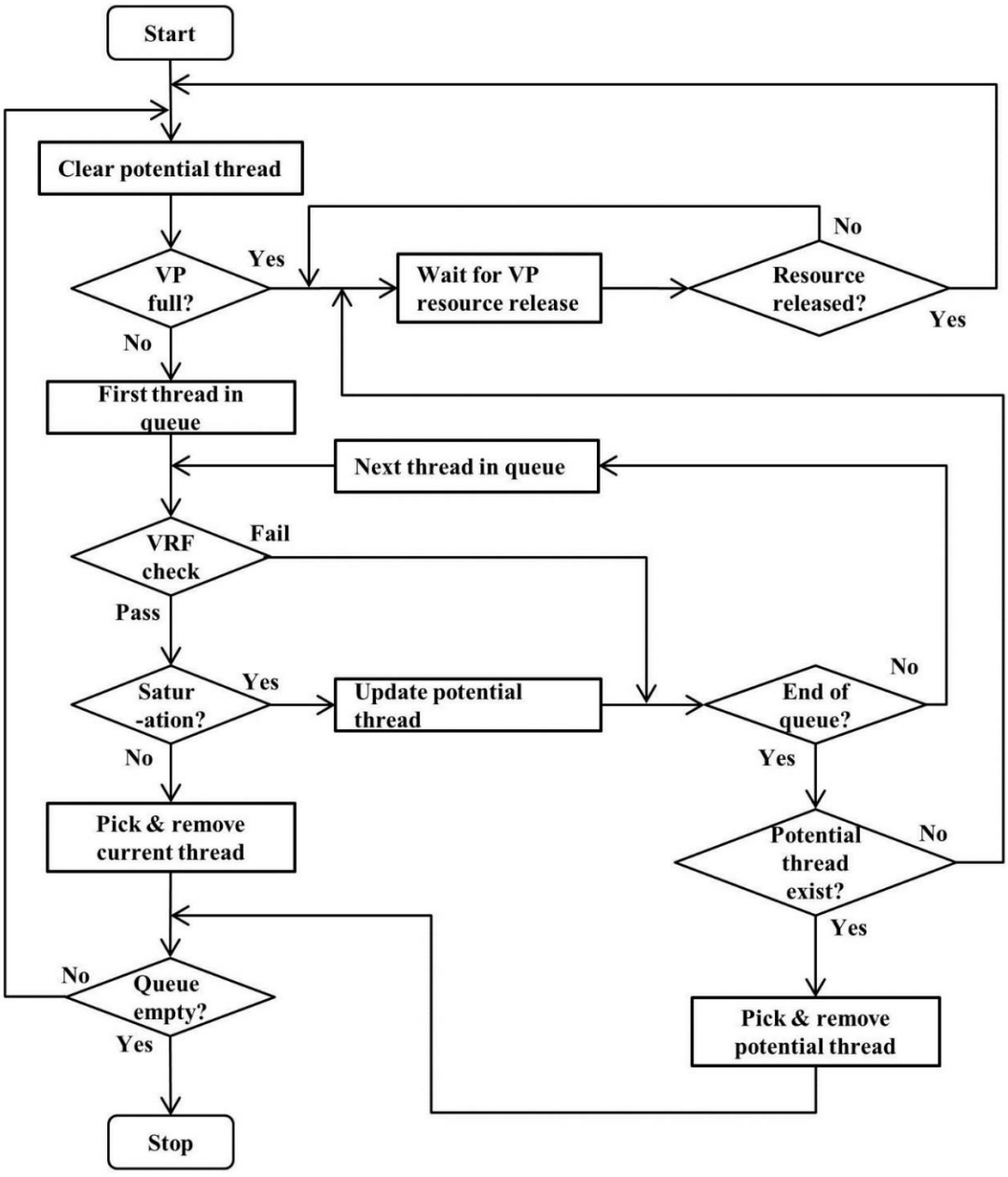

Figure 6.1 Scheduler flowchart.

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 searches down the queue only once to find a fitting thread that results in minimum saturation. The scheduler always starts investigation with the first pending thread of the longest native duration. If the available VRF resources are sufficient, utilization saturation check is performed to see whether this thread will lead to an ALU or LDST overall utilization higher than 90%. If no saturation can occur, this thread is scheduled. Otherwise, it becomes the "potential thread" for scheduling. When another thread in the queue is found to lead to utilization saturation, it is compared against the currently potential thread. If the former thread can 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 pending threads are either not fitting or lead to saturation, the currently potential thread is chosen for immediate scheduling.

6.2 Queues of Fixed Length

The scheduler is tested with thread queue lengths of 8 and 16 using each time six different thread combinations. Threads were chosen with equal probability from the list of 15 benchmarks of Section 4.2.2. The input data size for each thread was randomly picked, resulting in different native durations for the same benchmark across different thread combinations. The average result is shown in Figure 6.2. To compare with the optimal solution for the six scenarios with a queue length of eight, the best possible execution time is identified via exhaustive search (i.e., by calculating the total execution time of all possible scheduling orders using a C program). Compared to the optimal case, which cannot be implemented in practice at run time, the execution time is only 14.7% slower on

the average and optimality is achieved in one of the six scenarios. In the case of a queue length of eight, the scheduling algorithm results in an average speedup of 2.83 compared to the case without VP sharing; when the queue length increases to 16, the average speedup increases to 3.33. As the length of the thread queue increases, the speedup further increases to become close to four, which is ideal (since it matches the maximum number of threads that can share the VP). In this dissertation, only one of the six scenarios is chosen for each queue length to generate a table with detailed simulation information. Table 6.1 and Table 6.2 show details for all threads, including the critical times when threads are chosen for scheduling or complete execution.

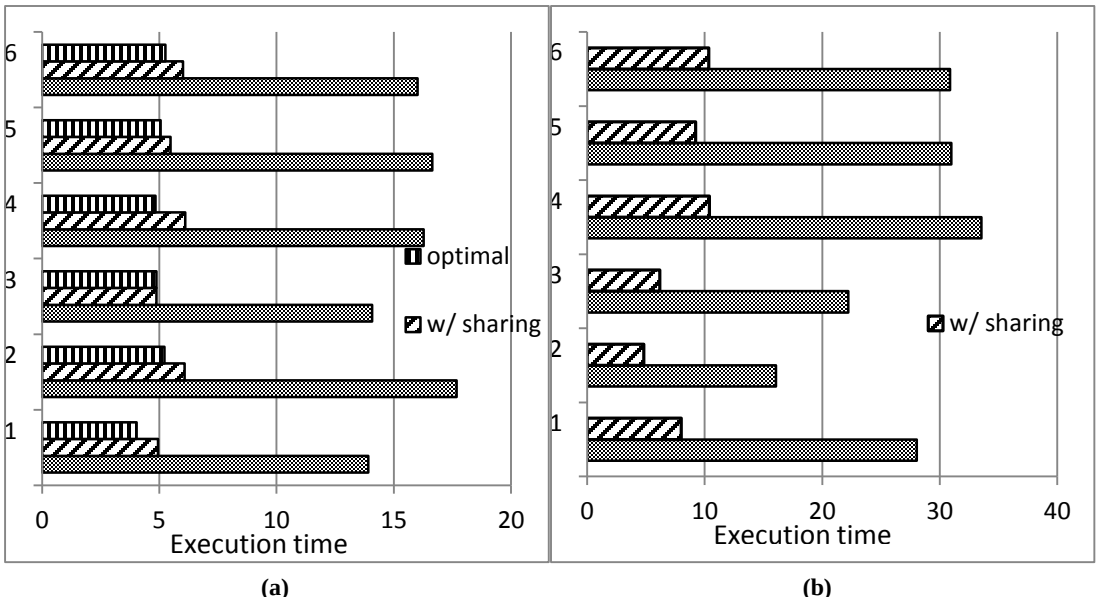

Figure 6.2 Execution time for thread queues of fixed length. (a) Length = 8. (b) Length =16.

Table 6.1 Detailed Results for a Schedule with Pending Thread Queue Length of 8.

Task ID	Application	VL	Native Duration w/o VP Sharing (us)	ALU Utilization (%)	LDST Utilization (%)	Issue Time (us)	Commit Time (us)	Actual Duration (us)
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 **Best issue order based on simulation of all permutations: 0,3,6,4,1,2,5,7** **Actual execution time = 6.083ms** **Total native duration w/o VP sharing= 17.677ms** **Speedup =2.91** **Optimal execution time =5.215ms**								

Table 6.2 Detailed Results for a Schedule with Pending Thread Queue Length of 16.

Task ID	Application	VL	Native Duration w/o VP Sharing (us)	ALU Utilization (%)	LDST Utilization (%)	Issue Time (us)	Commit Time (us)	Actual Duration (us)
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**								

6.3 Open System with Randomly Arriving Threads

To simulate an open system with randomly arriving tasks, all tasks arriving within 10ms time slices are scheduled. A fixed input size is chosen for each benchmark to create 15 distinct tasks. The characteristics of each task are listed in Table 6.3. Dynamic energy measurement is the focus of Chapter 7. 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 are generated and average values for the six schedules is calculated. Details of task arrivals and execution times are shown in Table 6.4 to Table 6.6. The average of the total execution time for all threads scheduled in a time slice is shown in Figure 6.3. 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 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 7.2.2).

Table 6.3 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

Table 6.4 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	.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	.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 6.5 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 6.6 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	2	1	1	0	0	1	0.83
1	MM_64	1	2	2	2	2	0	1.5
2	MM_32	1	0	0	1	0	1	0.5
3	RGB2YIQ_32	0	2	3	1	1	0	1.16
4	FIR_64	1	2	0	0	0	0	0.5
5	DCT_64	2	0	1	0	1	0	0.66
6	DCT_32	1	0	1	0	2	0	0.66
7	DCT_16	0	2	2	0	2	1	1.16
8	MM_16	2	3	3	1	0	0	1.5
9	RGB2YIQ_64	1	1	0	1	1	2	1
10	FIR_32	1	0	1	1	3	1	1.16
11	VDP_64	0	2	1	0	1	0	0.66
12	VDP_32	0	2	1	1	1	2	1.16
13	FIR_16	0	1	2	1	2	2	1.33
14	VDP_16	0	2	1	0	0	1	0.66
Total Native Duration (ms)		28.75	34.57	35.14	17.81	25.53	15.76	26.26
Actual Duration (ms)		8.44	10.29	9.81	6.17	7.83	5.53	8.01
Speedup		3.4	3.36	3.58	2.88	3.26	2.84	3.23

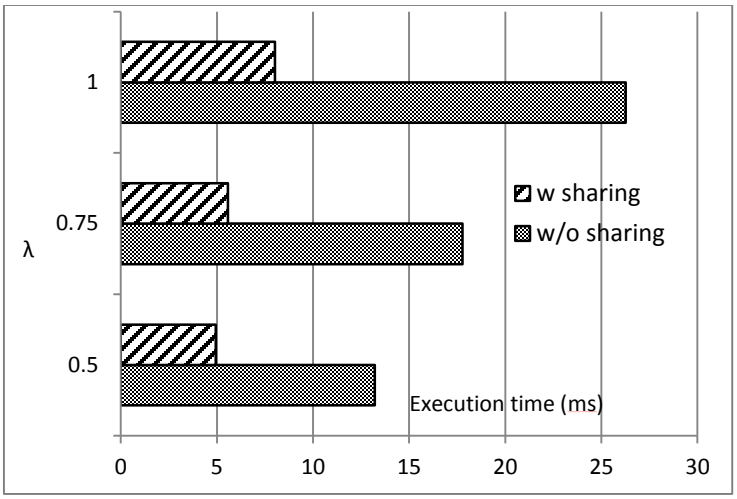

Figure 6.3 The average of the total execution time for all threads scheduled in a time slice, with and without VP sharing, for λ= 0.5, 0.75 and 1. (Time slice: 10ms.)

CHAPTER 7

POWER ANALYSIS AND ENERGY CONSUMPTION

In this chapter, a comprehensive power analysis and energy consumption for the single host system is presented (Section 7.1). For this system, as mentioned in Section 4.1, all the results are based on a frequency of 50MHz for the VP and VM. Results are given for the benchmarks suit presented in Section 4.1.2 for the single host system. In Section 7.2, the power analysis is presented for the multi-host system which is based on a frequency of 100MHz for the VP and VM running the benchmark suit of Section 4.2.2. In this section it is also discussed how proper scheduling and a power gating technique can reduce significantly the energy consumption.

7.1 Power Analysis for the Single Host System

For high accuracy in the estimation of the VP's power and energy consumption, the Xilinx Power Analyzer (XPA) is employed which can determine the power when the activity rate for each signal and net in the hardware are specified. Power dissipation has two major components. Static power dissipation is due to current leaking through the transistors, even without any activities. Dynamic power depends on the design's activities [Xilinx INC, 2011].

Estimating the dynamic power of a design requires knowledge of the activity rates for all signals and nets in the hardware. This information is available in the Xilinx SAIF (Switching Activity Interchange Format) and VCD (Value Change Dump) files which are generated after performing timing simulation of the design. Timing simulation for each benchmark instantiation is performed using the ISE simulator (ISim) tool that generates an

SAIF file that shows the exact activity rates in the placed-and-routed (RAP) design. The design is first synthesized, translated and mapped to the target platform before RAP. The SAIF file is generated between desired intervals during simulation and includes information for the interval. The SAIF, NCD (Native Circuit Description) and PCF (Physical Constraint) files are imported into XPA to obtain accurate estimation of the power consumption before the configuration bit-stream is generated and downloaded into the FPGA.

To find the maximum dynamic energy dissipation, maximum VP performance is targeted. To this end, we focus on the kernel of each benchmark that involves in the calculations the VP, private memories and shuffle engines. The respective vector instructions are then applied directly to the VP instead of being issued by the MB (i.e., there is no delay between consecutive VP instructions). For each kernel, first behavioral simulation is performed to obtain the interval for SAIF generation. The start point is the moment that the first vector instruction from the kernel enters the VC whereas the end point is when the last vector element is written back to the VRF or private memories. After determining the desired interval, the post-RAP simulation is performed for the benchmark and the SAIF file is generated for the desired interval. The device configuration and environmental parameters are set to their default values (e.g., the ambient temperature is 50^0C and the airflow is 250LFM). The static power remains unchanged at 2.878W for all benchmarks since the same target device is used (FPGAs do not support power gating).

For matrix multiplication with Algorithm 3, which is the most vectorizable in Table 5.1, the innermost loop that involves three instructions is considered as the target kernel. It is repeated VL times until one row of the product matrix is generated. This kernel

includes one load instruction, one vector-scalar multiplication and one vector-vector addition. Table 7.1 shows the measured values for this benchmark under this maximum power dissipation scenario that assumes no delay between issuing consecutive vector instructions. "Kernel duration" shows the length of the chosen interval. "Application duration" is obtained from the "ideal" numbers in Table 5.5. It can be seen that the clock distribution network dominates the dynamic power, which is in agreement with earlier results. In fact, as it will be seen in Section 7.2, the clock power is technically part of the static power. The dynamic power does not include the clock distribution network power dissipation.

Table 7.1 Power and Energy Consumption for Matrix Multiplication (f=50 MHz).

VL	Kernel Duration (ns)	VC+4lanes+Memories Dynamic Power (mW)			Application Duration (us)	Application Dynamic Energy (uJ)	Kernel Dynamic Power (mW)
		Clock	Signal & Logic	BRAM & IO			
16	550	106.8	71.96	4.16	137	25.06	182.92
32	710	106.8	87.96	5.28	809	161.83	200.04
64	1030	106.8	104.2	6.84	5048	1099.66	217.84

For FIR filtering with Algorithm 2, which is the most vectorizable in Table 5.2, the target kernel for power estimation is the internal loop that 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 that consist of four load, four vector-scalar multiplication and four vector-vector addition instructions, and maximizes the VP utilization for this benchmark. The power and energy values are presented in Table 7.2.

For FFT with Algorithm 2, which is the most vectorizable in Table 5.3, a single stage of FFT calculation forms the target kernel. Each stage involves sixteen vector

Table 7.2 Power and Energy Consumption for FIR Filtering (f=50 MHz).

VL	Kernel Duration (ns)	VC+4lanes+Memories Dynamic Power (mW)			Application Duration (us)	Application Dynamic Energy (uJ)	Kernel Dynamic Power (mW)
		Clock	Signal & Logic	BRAM & IO			
16	1150	106.8	93.52	5.36	9.2	1.89	205.68
32	1790	106.8	102.6	6	28.3	6.09	215.4
64	3070	106.8	109.52	6.52	106.8	23.80	222.84

instructions which include two data shuffle, six vector-vector multiplication, two load, two store, two vector-vector addition and two vector-vector subtraction instructions. Table 7.3 contains the results. Since data shuffling does not interfere with internal VP operations, the "ideal" numbers from Table 5.7 are assumed for the time. It can be seen that, when the ratio of ALU to LDST instructions issued increases in the kernel (e.g., FFT), the dynamic power of the signals and logic increases since more results are produced in the FPUs.

Table 7.3 Power and Energy Consumption for FFT (f=50 MHz).

VL	Kernel Duration (ns)	VC+4lanes+Memories Dynamic Power (mW)			Shuffle Engines+ Crossbar Dynamic Power (mW)	Application Duration (us)	Application Dynamic Energy (uJ)	Kernel Dynamic Power (mW)
		CLk	Signal & Logic	BRAM & IO				
16	1350	106.8	107.36	5.52	22.12	7.9	1.91	241.8
32	2090	106.8	133.12	6.48	22.12	19.1	5.13	268.52
64	3690	106.8	144.16	7	22.12	45.6	12.77	280.08

For the RGB2YIO benchmark, the chosen kernel with the maximum VP utilization converts the color space for one row of the input block. This kernel consists of 21 vector instructions and includes three load, nine scalar-vector multiplication, six vector-vector addition and three store instructions. The "ideal" numbers from Table 5.8 are used. The results are shown in Table 7.3

Table 7.4 Power and Energy Consumption for RGB2YIQ (f=50 MHz).

VL	Kernel Duration (ns)	VC+4lanes+Memories Dynamic Power (mW)			Application Duration (us)	Application Dynamic Energy (uJ)	Kernel Dynamic Power (mW)
		Clock	Signal & Logic	BRAM & IO			
16	2350	106.8	82.96	5.72	28.7	5.6	195.48
32	4230	106.8	93.16	5.64	103.3	21.2	205.2
64	7590	106.8	94.2	5.44	390.6	88.6	206.4

7.2 Power Analysis for the Multiple Hosts System

In this section, the energy consumption for the benchmarks of Section 4.2.2 is investigated. Based on the power dissipation of individual benchmarks, a projection is made of the total energy consumption for the dynamic schedules of Subsection 6.3. In a more accurate categorization, power consumption has three components: device static, design static and design dynamic [Beldianu et al., 2015]**.** The device static power, also known as leakage power, is a device specific constant not related to resource utilization or the switching activity. Under our simulation conditions for an ambient temperature of 50^0C and an airflow of 250LFM (linear feet per minute), the leakage power for our 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 VP actually uses, the presented power model adds the design's static and dynamic powers to estimate the total dissipation.

7.2.1 VP Dynamic Power for the Multiple Hosts System

To reliably estimate the dynamic power, the VP design was fully implemented and all signal switching activities of each system node were used as input for power calculation. As for the single host system, the VP is fully implemented (i.e., synthesized, translated, placed and routed) using the Xilinx ISE tool chain, and performed PAR ISE simulations. The binaries of the vector instructions of each benchmark were generated to estimate the dynamic power. The power measurements 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, the average power consumption for one iteration of each vector kernel is measured. For matrix multiplication and FIR filtering, the same kernels as previously discussed for the single host system in Section 7.1 are applied here. For MM, the innermost loop that involves three vector instructions is considered as the target kernel. 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. 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 7.5 to Table 7.9. The energy numbers shown are based on the input data sizes of Section 5.2.1. Using the measured power, the total dynamic energy consumption of each benchmark for various native durations can be calculated; this approach aids the estimation of the energy consumption in dynamic environments. The dynamic energy results for the predefined tasks of Section 6.3 were included in Table 6.3. Using a task's average number of arrivals per time slice, its average dynamic energy consumption per slice can be produced. Figure 7.1 shows that the dynamic energy consumption is related almost linearly to the task arrival rate.

Table 7.5 Power and Energy Consumption for MM (f= 100MHz).

VL	Kernel Duration (ns)	VC+4Lanes+Memories Dynamic Power (mW)		Kernel Dynamic Power (mW)	Application Duration (us)	Application Dynamic Energy (uJ)
		Signal & Logic	BRAM & IO			
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

Table 7.6 Power and Energy Consumption for FIR (f= 100MHz).

VL	Duration (ns)	VC+4Lanes+Memories Dynamic Power (mW)		Dynamic Power (mW)	Duration (us)	Dynamic Energy (uJ)
		Signals & Logic	BRAM & IO			
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

Table 7.7 Power and Energy Consumption for VDP (f= 100MHz).

VL	Duration (ns)	VC+4Lanes+Memories Dynamic Power (mW)		Dynamic Power (mW)	Duration (us)	Dynamic Energy (uJ)
		Signals & Logic	BRAM & IO			
16	765	136.26	11.24	147.5	12	1.77
32	1235	187.4	19.04	206.44	15	3.09
64	2275	243.28	24.92	268.2	18	4.82

Table 7.8 Power and Energy Consumption for DCT (f= 100MHz).

VL	Duration (ns)	VC+4Lanes+Memories Dynamic Power (mW)		Dynamic Power (mW)	Duration (us)	Dynamic Energy (uJ)
		Signals & Logic	BRAM & IO			
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

Table 7.9 Power and Energy Consumption for RGB2YIQ (f= 100MHz).

VL	Duration (ns)	VC+4Lanes+Memories Dynamic Power (mW)		Dynamic Power (mW)	Duration (us)	Dynamic Energy (uJ)
		Signals & Logic	BRAM & IO			
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

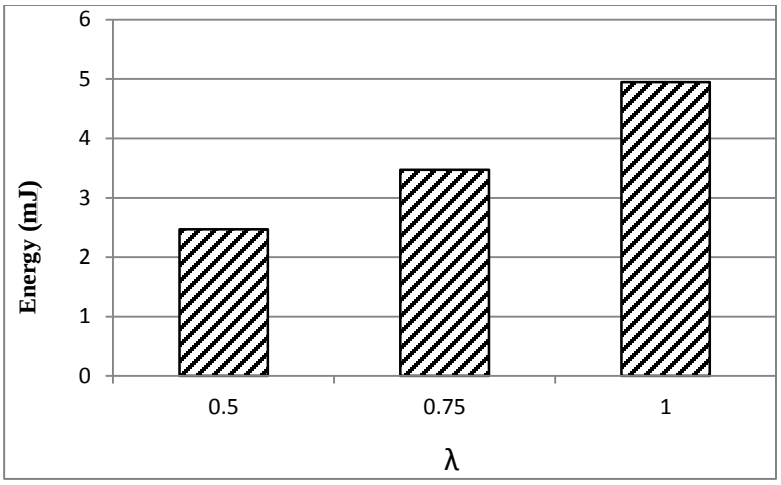

Figure 7.1 Average total dynamic energy consumption per time slice for λ=0.5, 0.75 and 1.

7.2.2 Total Energy Consumption in the Multiple Hosts System

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 et al., 2015].

Although commercial FPGAs currently lack PG support, PG in association with proposed dynamic scheduler of Section 6.3 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 static power measurements, the assumption of a 85% static power reduction with PG and the measured average execution time in Figure 6.3, the VP's average static energy consumption is projected per time slice for a given task arrival rate. Combining the results with the dynamic energy of Figure 7.1, Figure 7.2 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 our very substantial performance improvements.

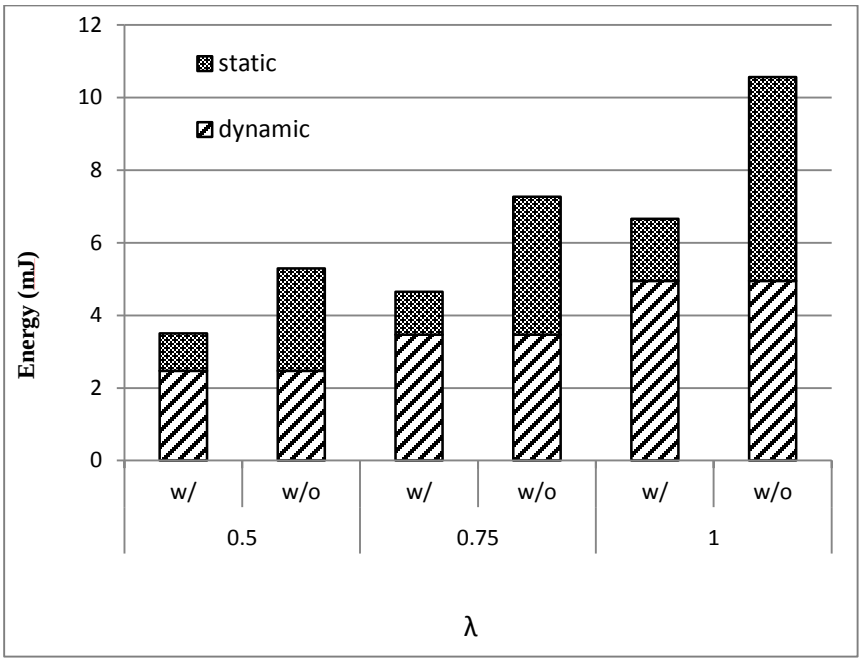

Figure 7.2 Total energy consumption with (w/) and without (w/o) VP sharing, and with power gating, for λ=0.5, 0.75 and 1.

CHAPTER 8

VIRTUALIZED SMT VP AND OPTIMIZATION VIA THREAD FUSION AND LANE CONFIGURATION

In this chapter, architectural VP modifications are described in Section 8.1. Some of the improvements and modifications are made to increase VP utilization, while others result in a virtualized SMT VP. Resource utilization and performance benchmarking results for the virtualized SMT VP are included in Sections 8.2 and 8.3, respectively. An accurate power dissipation model for this new VP is introduced in Section 8.4, and energy optimization scenarios and scheduling processes using this power model are discussed in Section 8.5.

8.1 Virtualized VP Architecture

Subsection 8.1.1 covers those modifications which result in increasing VP performance and utilization. Subsection 8.1.2 introduces a novel architecture for performing the data shuffle instruction. The detailed architectural implementation of this pipelined structure is discussed and its performance evaluation is presented. Subsections 8.1.3 to 8.1.5 cover hardware modifications applied to the VP to increase flexibility and realize VRF and VM virtualization processes in the VP data path. Subsection 8.1.6 describes how the new VP architecture can be exploited in the fused mode.

8.1.1 Increasing the VP Saturation Level

As mentioned earlier in Section 5.2, the developed VP has a limitation on its maximum achievable utilization level, namely its saturation level. The main reason contributing to this problem is having an idle clock cycle between fetching consecutive instructions in

either the LDST or ALU data path from the instruction FIFOs. Although this idle cycle eases the process of functional verification, it also puts an undesirable bound on VP utilization. This causes the maximum theoretical utilization of the VP to be 80% for VL=16, 88% for VL=32 and 94% for VL=64. By a slight modification of the ALU and LDST decoder state machines in Figure 3.3, the idle high impedance state between successive instructions is removed. Saving this precious clock cycle can results in a theoretical maximum utilization of 100% for the VP resources. As a result, the performance of the VP for single thread execution is also slightly improved, which can be noticed from the benchmarking of Section 8.3.

Using open source code for FPU ALU units causes another bottleneck in VP performance. Using RTL code for developing the FPU adder and multiplier is beneficial when the VP is designed to be prototyped on ASIC platforms. Since our design is prototyped on an FPGA platform (for proof of concept), the Xilinx IPs replace open source FPUs. As a result, the VP uses more Xilinx DSP modules rather than registers and LUTs (as presented in Section 8.2) and is no longer limited by the critical path delay in open source blocks (it results in 15% improvement in the critical path delay).

8.1.2 VP Pipelined Data Shuffle Network

The data shuffle engine structure which was used in the single host system of Section 3.1 has some limitations. First of all, it is not pipelined. It uses a host side VM port to read the source, and destination address as well as shuffle index for each element. Since the VM bandwidth is bounded by one element per clock cycle, a total of eight clock cycles are needed per element in the lane to perform shuffling. Three clock cycles are needed per element to fetch the source, destination and corresponding index address from the memory,

one for generating the physical address and four more clock cycles to apply round-robin scheduling between memory banks to avoid any possible contention. All the shuffle engines work concurrently. So, for example, for VL=32 with 8 elements in each lane the shuffle engine architecture takes 64 clock cycles to complete shuffling. Secondly, shuffle instructions use their own controller. Having a dedicated controller removes the unnecessary stall of the VP during shuffling. As a drawback, it makes the synchronization of threads complex under SMT since each thread has two interfaces to the VP.

To alleviate these problems, a new data shuffle network is designed and implemented. This design is completely pipelined and works on elements that reside within the VRF rather than the VM. The shuffle network shuffles data among lanes. A data shuffle instruction is issued like a regular ALU instruction through the VC. The previous synchronization problem no longer exists since all requests are issued to the VP through a single interface (VC). The total clock cycles needed for shuffling is equal to the number of elements per lane since the design is fully pipelined (e.g., 8 clock cycles for VL=32).

The new data shuffle network consists of four stages of pipelining. No extra read or write port is needed on the VRF to perform shuffling. The ALU decoder reads the source data and the shuffle index from the VRF and sends them to the shuffle unit. In each pipeline stage, the shuffle unit registers both the data and index per element, and sends them to the next level on a clock edge. The targeted next level can be located in the lane shuffle unit or in the next neighboring lane's shuffle unit. So each pipeline stage in a lane shuffle unit can accept data and an index from its own higher stage or from the higher stage of the previous lane's shuffle unit. The off lane passing decision is made based on the corresponding input index (I_indx) and lane index (L_indx). If (I_indx mod 4= L_indx),

then that data is located in the correct lane and no longer needs to be passed to the neighboring lane. Since there are four lanes in the developed VP, at most four clock cycles are needed for an element to reach its destination lane. The result of the shuffle unit (both the data and index), which is ready after four clock cycles, is buffered in the WB unit. The data will be written in the correct destination in the VRF based on the index and destination information provided to the WB unit by the decoder. Figure 8.1 gives an example of how the pipelined shuffle network works. The input for level 1 is provided by the lane's ALU decoder. The index numbers in the figure represent (I_indx mod 4) values. The elements "A" to "D" represent four successive elements in the vector register. Since the VRF has a low order interleaved structure, elements reside in the four different lanes.

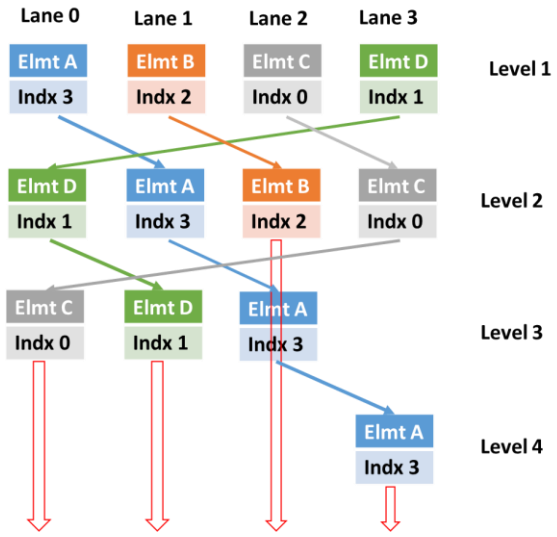

Figure 8.1 Data shuffling example for the pipelined shuffle network.

It can be seen from Figure 8.1 that when an element's index matches the lane index (I_indx mod 4= L_indx), there is no need to pass the element to the next lane. This can happen only after four stages. Since the shuffle network consists of four stages, three levels

of passing elements will be needed between stages. The overall architecture of the pipelined data shuffle network is depicted in Figure 8.2.

It can be observed from Figure 8.1 that, in the last stage of the shuffle unit, four elements may need to be written into the VRF. This can happen only if four elements are to be written into four different lanes' VRFs. This fact imposes a limitation on the proposed pipelined structure. Based on this limitation, only certain shuffle patterns could be realized in the pipelined architecture and the rest may require the help of a scalar host processor. If four successive indexes in the shuffle index register target four different lanes' indexes, then the pattern can be realized directly with the pipelined structure. Fortunately this is the case for most of the desired shuffling patterns in a practical application such as FFT.

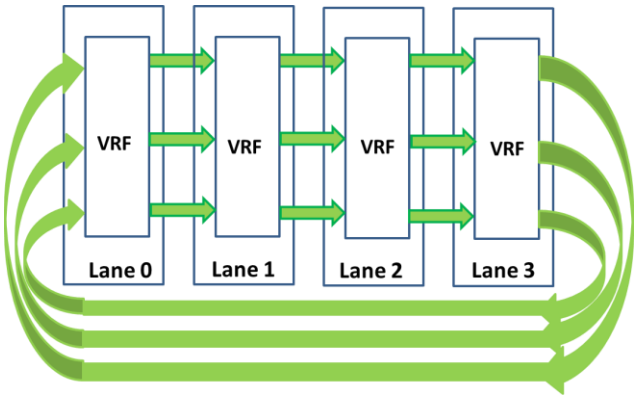

Figure 8.2 Overall architecture of the pipelined data shuffle network.

To evaluate the new shuffle architecture, the FFT benchmark is rearranged to be run on the single host system with a modified VP architecture that includes the new pipelined shuffle network. Comparisons in Table 8.1 are against the previous architecture for FFT. It shows that the new architecture can accelerate FFT around 2-3 times, compared to the old VP with the unpipelined shuffle engine network.

Table 8.1 FFT Performance and Utilization Comparison Between the Previous VP with the Shuffle Engines and the Modified VP with the Pipelined Data Shuffle Network (f=100MHz)

VL	Platform	ALU (%)	LDST (%)	SHF (%)	Execution Time (ns)
16	Pipeline shuffle	40.2	20.1	N/A	3980
	Shuffle engines	4.7	12.3	7.4	13000
32	Pipeline shuffle	65.04	31.22	N/A	6150
	Shuffle engines	10.3	26	15.4	15500
64	Pipeline shuffle	78.81	36.78	N/A	12180
	Shuffle engines	15.4	38.4	23.1	25000

8.1.3 Virtualized VM Address Space

Each vector lane in the VP contains an ALU unit as well as a LDST unit that interfaces the VM. As discussed in detail in Section 3.1.1, the VP 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. 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.

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 runtime 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 by the hardware under disparate VP configurations. To facilitate address virtualization, the host-to-VM mux and the VC are modified 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.

8.1.4 Configurable Components

Figure 8.3 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 8.3.a) and four-lane (Figure 8.3.b) 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 $M \geq N$. 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.

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 8.3.b, 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.

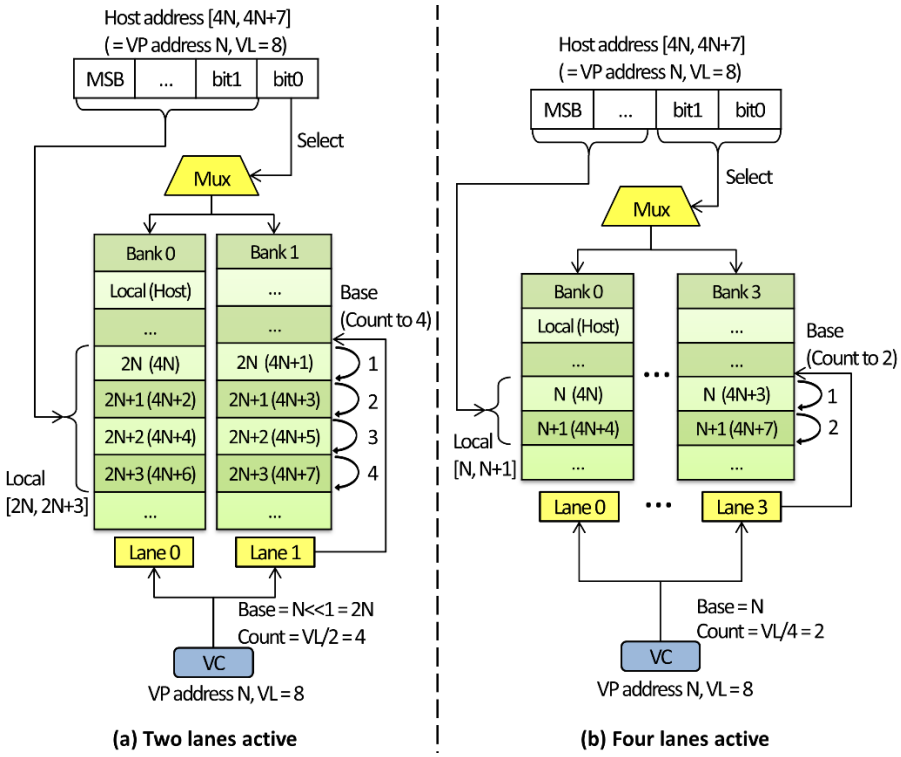

Figure 8.3 Mapping of VL, host-to-VM address and VP-to-VM address via virtualization.

When the host dynamically deactivates two VP lanes and their attached VM banks, only two banks remain 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.

8.1.5 VRF and VM Virtualization Under SMT

The VP is originally designed to support vector-based SMT and sharing among many processors. To achieve true SMT where instructions from multiple threads can coexist inside the VP pipeline without interference, both the VRF name space and the VM space are virtualized on a per instruction basis. With SMT virtualization, one SMT capable VP appears as multiple logical VPs (LVPs) to multiple hosts/cores. Shown in Figure 8.4 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 TLP via thread fusion. The thread ID for each instruction 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.

As shown in Figure 8.4, 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. 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 simplicity, an FPGA-based prototype capable of executing two threads simultaneously is built. However, the max number of simultaneous threads can be easily increased by increasing the number of instruction FIFOs and modifying the arbitrator's state machine. VRF virtualization for more than two threads can be supported by using a VRF renaming algorithm presented in section 3.3.2 which dynamically maps the threads' virtual VRF names to physical names while minimizing register fragmentation. Virtual VM for multiple threads can be implemented by using a memory management unit.

8.1.6 Fusion of Similar Threads

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 8.5 shows how two DCT operations are accelerated by fusing the threads. Without fusion (Figure 8.5.a), the two operations will be executed sequentially. When two threads are fused (Figure 8.5.b), 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 8.4 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. 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.

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.

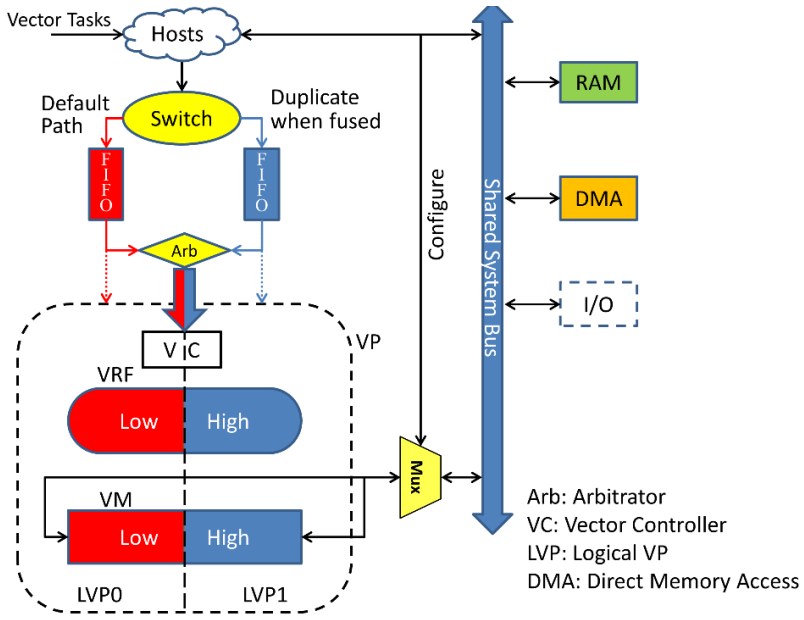

Figure 8.4 System architecture of a fusion capable VP of degree two

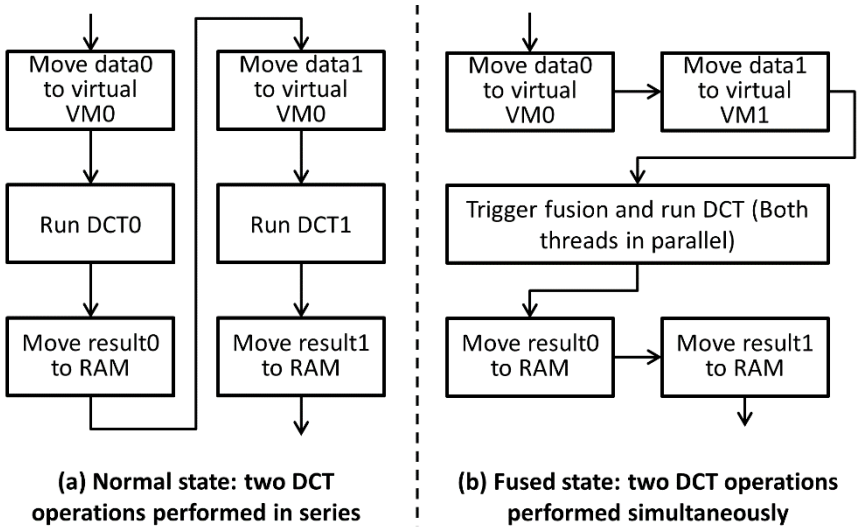

Figure 8.5 Fusion of two DCT operations

8.2 System Architecture and FPGA Implementation

To evaluate the two proposed techniques, a dual-threaded modified VP interfaced with a hosts system is prototyped on a Xilinx XC7Z045-1fbg676 FPGA. The system architecture

is similar to that in Figure 8.4, with the hosts system replaced by a MB processor that issues vector threads. Various vector kernels are stored in MB 16KB local memory. 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. I/O components on the bus are used for debugging purposes and we implemented an 8-bit LED 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. 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 and it is improved by 15% compare to old VP as open source FPUs are substituted with Xilinx IPs; for simplicity, a 100MHz system is implemented. The resource consumption breakdown for the VP is shown in Table 8.2.

Table 8.2 Resource Consumption and Utilization Percentage

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%)

8.3 Benchmarking the Virtualized VP

Four vector applications, which were introduced in Chapter 4, are picked for proposed system benchmarking. The applications are DCT, FIR , RGB2YIQ and VDP. Since the current VP supports three different VLs, each picked application is evaluated using all supported VLs, creating a total of 12 benchmarks. Each benchmark is characterized by its ALU utilization (UALU) and LDST utilization (ULDST). Each benchmark is executed under various configurations, measured the corresponding runtime, and calculated the utilization the pipelines. The utilization is defined as Ototal /O4lanes, where Ototal is the total number of operations for an application and O4lanes is the maximum number of operations that can be performed by the four lanes during the application's runtime.

Table 8.3 to 8.5 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 8.3 Performance Profile Data for 4Lanes Unfused VP

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
RGB2YIO	**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

The native utilization *(U)* and native runtime *(T)* are called an application's figures under configuration **a**. 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 are not affected due to lane deactivation.

Table 8.4 Performance Profile Data for 4Lanes Fused VP

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
RGB2YIQ	**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 8.5 Performance Profile Data for 2Lanes Unfused VP

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

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 8.1.

if (U_{ALU} <50 and $U_{LDST} < 50$) then

$$U'_{ALU\_2lanes} = U_{ALU};\ U'_{LDST\_2lanes} = U_{LDST};\ T'_{2lanes} = T$$

else if ($U_{ALU} > U_{LDST}$) then

$$U'_{ALU\_2lanes} = 50;\ U'_{LDST\_2lanes} = \frac{U_{LDST}}{U_{ALU}}50;\ T'_{2lanes} = \frac{U_{ALU}}{50}T \quad (8.1)$$

else

$$U'_{ALU\_2lanes} = \frac{U_{ALU}}{U_{LDST}}50;\ U'_{LDST\_2lanes} = 50;\ T'_{2lanes} = \frac{U_{LDST}}{50}T$$

The maximum utilization achievable is 50% when two lanes are deactivated. The Equation 8.1 agrees with the measurements shown in

Table 8.3 to Table 8.5. 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 8.4. The runtime scheduler will use the utilization information to choose the optimal number of active lanes based on the scheduling policy. The scheduling policy will be discussed in Section 8.5.

8.4 Power Model

A highly accurate VP power consumption model is needed for optimization purposes. By combining the VP's NCD file with the testbenches of different scenarios, the detailed SAIF for various VP utilizations is obtained. By using testbenches that issue instructions to the VP at various rates, the VP's static and dynamic power under various utilizations is measured. Figure 8.6 shows dynamic power results.

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 8.2. Each coefficient K is the power per utilization in mW/% for each corresponding component. On the other hand, the 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 8.3.

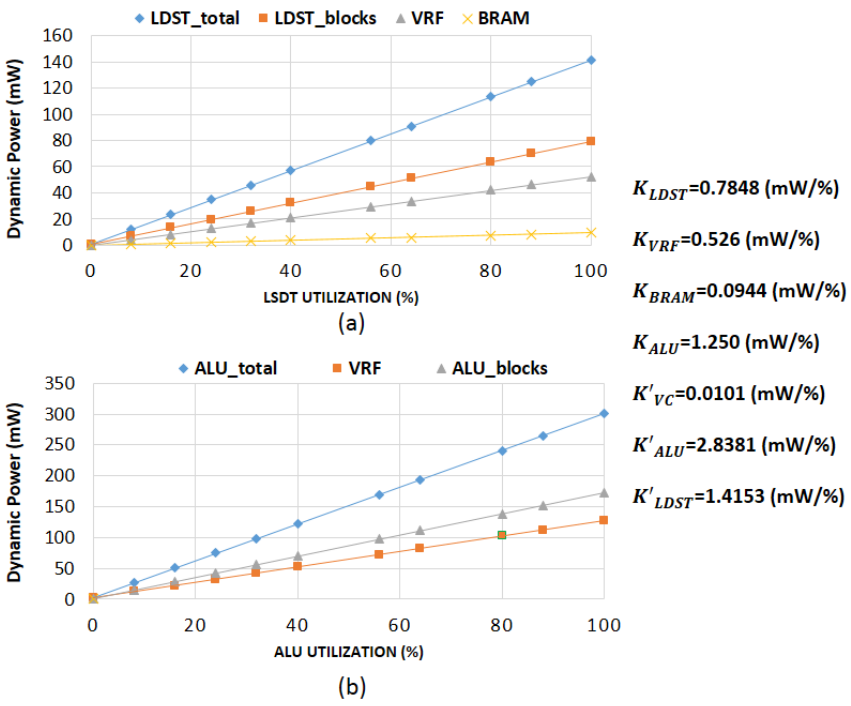

Figure 8.6 Dynamic power vs. utilization for both ALU and LDST data paths.

$$P_{ALU\_dynamic} = K_{ALU}U_{ALU} + K_{VRF}(3*U_{ALU}) \qquad (8.2)$$
$$P_{LDST\_dynamic} = K_{LDST}U_{LDST} + K_{VRF}U_{LDST} + K_{BRAM}U_{LDST}$$

$$P_{VC} = K_{VC} * IR = K_{VC} * (\frac{(U_{ALU} + U_{LDST}) * 4}{VL}) = K'_{VC}(U_{ALU} + U_{LDST}) \qquad (8.3)$$

By adding together the terms in Equation 8.2 and Equation 8.3, the Equation 8.4 is obtained the VP's dynamic power as a simple linear function of U_{ALU} and U_{LDST}.

$$\begin{aligned} P_{VP\_dynamic} &= K_{LDST}U_{LDST} + K_{ALU}U_{ALU} + K_{VRF}(3*U_{ALU} + U_{LDST}) \\ &\quad + K_{BRAM}U_{LDST} + K'_{VC}(U_{ALU} + U_{LDST}) \\ &= K'_{ALU}U_{ALU} + K'_{LDST}U_{LDST} \end{aligned} \qquad (8.4)$$

This power model matches the measurements of the VP's dynamic power vs. U_{LDST} with idle ALU (Figure 8.6.a), and power vs. U_{ALU} with idle LDST (Figure 8.6.b). From the measured data the coefficient K for each component is extracted; the most important are for the ALU and LDST units: K'_{ALU} = 2.838mw/%, and K'_{LDST} = 1.415mW/%.

The VP's total power is given by Equation 8.5. 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 [Beldianu et al., 2015]. P_{static} is 108.2mW, 63.15mW and 40.63mW for the 4, 2, and 1 lane configuration, respectively. In Equation 8.5, U_{ALU} and U_{LDST} are the applications' actual utilizations under various situations. Combining Equation 8.5 with Equation 8.1,

Equation 8.6 is obtained that describes the relation of an application's power consumption with two active lanes and its native utilizations. A similar equation can be derived with one active lane.

$$P_{total} = P_{static} + K'_{ALU} U_{ALU} + K'_{LDST} U_{LDST} \tag{8.5}$$

$$\begin{aligned}
&\text{if } (U_{ALU} < 50 \text{ and } U_{LDST} < 50) \text{ then} \\
&\quad P_{total_2lanes} = 63.15 + K'_{ALU} U_{ALU} + K'_{LDST} U_{LDST} \\
&\text{elseif } (U_{ALU} > U_{LDST}) \text{ then} \\
&\quad P_{total_2lanes} = 63.15 + 50K'_{ALU} + 50\frac{U_{LDST}}{U_{ALU}} K'_{LDST} \\
&\text{else} \\
&\quad P_{total_2lanes} = 63.15 + 50K'_{LDST} + 50\frac{U_{ALU}}{U_{LDST}} K'_{ALU}
\end{aligned} \tag{8.6}$$

8.5 The Scheduling Policy

So far a vector application's P_{4lanes}, P_{2lanes} and P_{1lane} are obtained as function of their native utilizations. The execution times T_{2lanes} and T_{1lane} are also related to T_{4lanes}, and the example for T_{2lanes} is shown in Equation 8.1. The set of P and T values form two-dimensional matrices with U_{ALU} and U_{LDST} as indexes. Two different scheduling policies using P and T are proposed. 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}, the utilization boundary for optimal configuration can be determined. Figure 8.7.a 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 which minimizes the product of an application's execution time and energy consumption can be obtained; it is shown in Figure 8.7.b.

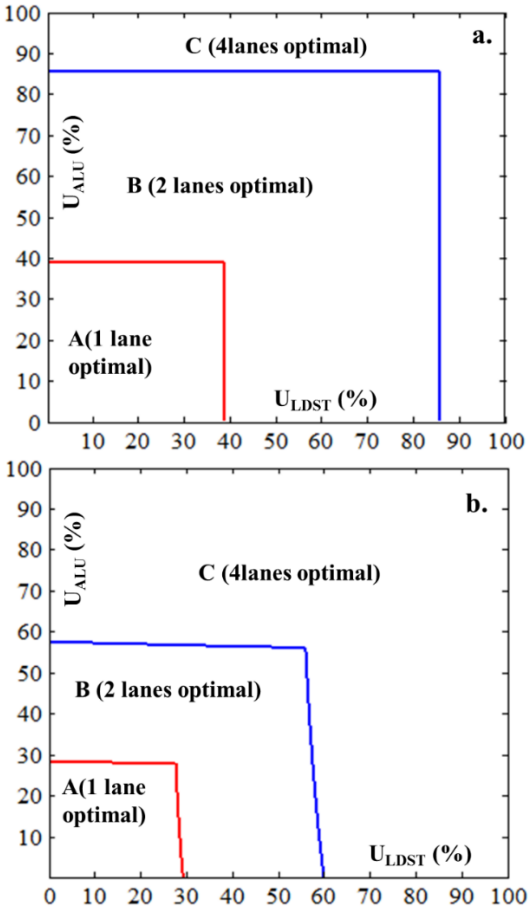

Figure 8.7 Optimal utilization boundaries for a. minimum energy b. minimum energy-execution time product

The two scheduling policies (E_{min} and ET_{min}) were 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 easily identifies fusable tasks. The

tasks are those in Section 8.3. For each optimization policy, every task has two optimal execution configurations: unfused and fused modes. All configurations can be obtained by combining each task's U_{ALU} and U_{LDST} with the results shown in Figure 8.7. As mentioned previously, the scheduler will treat a fused task as a new task with its own U_{ALU} and U_{LDST}.

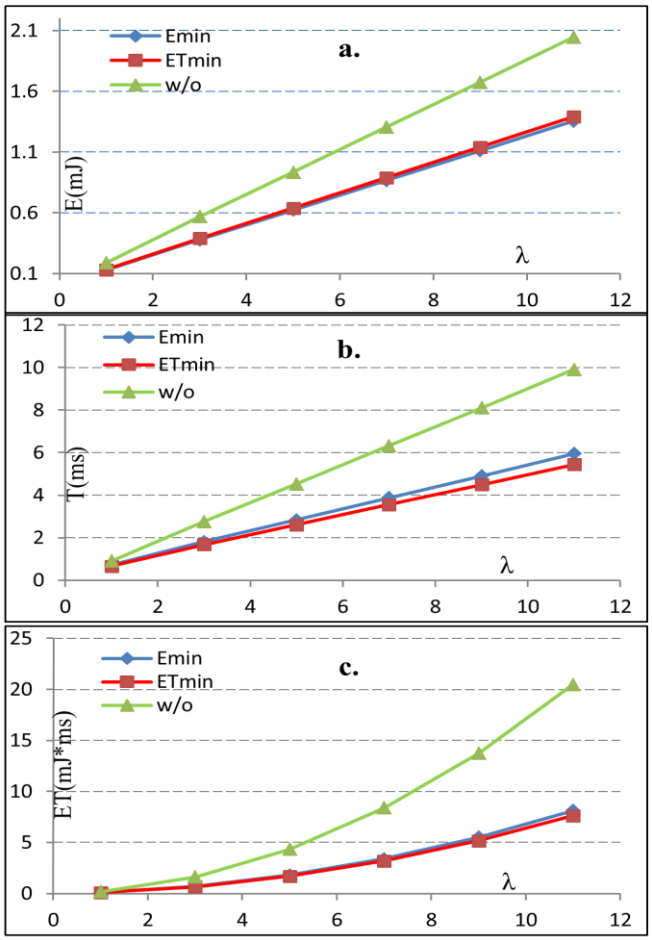

Figure 8.8 Comparison of the E_{min}, ET_{min} policies against a VP w/o fusion and lane configuration over the average of 1000 time slices. a. energy b. runtime c. energy-runtime product.

The task queues for 1000 time slices are generated using the MATLAB random number generator, and calculated the average parameters for the two scheduling policies

and also for the VP without the proposed techniques. As shown in Figure 8.8, 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 $\lambda = 11$ is close to 10ms and the system is about to overflow.

CHAPTER 9

COCNCLUSION AND FUTURE WORK

9.1 Conclusion

This dissertation presented a multi-lane VP architecture as a high-performance coprocessor for data-parallel applications in multicore/multithreaded processors. More specifically, the main motivation of this work was to introduce a multithreaded VP framework realizing SMT and eventually resource virtualization. This coprocessor is applied to three system architectures.

In the single host system, the VP is exclusively dedicated to a scalar processor and improves system performance via exploiting DLP. The proposed VP has a VIRAM-like architecture with dedicated data paths in each lane for LDST and ALU instructions. This data path separation makes the VP capable of exploiting ILP as well. Assigning a private memory to each vector lane and specifying one set of memory ports exclusively for transactions between that memory and the corresponding lane increases the speedup for memory-based vector instructions. Data shuffling and index addressing are realized using distributed data shuffle engines and a crossbar which is placed between the private and global memories. A benchmark suite to evaluate the system performance is introduced which shows an up to 1500-fold speedup over a scalar processor; the area is increased 11-fold. Detailed performance and power dissipation results for each benchmark are provided. The results prove the viability of our approach.

In the multiple hosts system, the scalar cores share the VP resources via VP virtualization that improves the aggregate utilization and performance with SMT. Virtualization can be applied in a multicore environment where the VP is shared by

multiple cores via a bus or in a unicore environment where the core is designed to support SMT. An easy-to-use interface makes VP sharing transparent to application programmers while improving the throughput many-fold. More specifically, the proposed VP can simultaneously execute multiple threads of similar or disparate vector lengths to improve VP throughput. The virtualization technique is prototyped for a multi-core processor embedded in an FPGA as a multiple hosts system. Under the dynamic creation of threads with diverse needs for vector sizes and types of operations, benchmarking results show impressive VP speedups of up to 333% and total energy savings of up to 37% with proper thread scheduling and power gating compared to a single host system that allows VP access to just one thread at a time. Finally, the performance improvements compared to the single host system and other prior works for VP sharing that did not support VP virtualization further prove the viability of our approach since the obtained speedups are impressive.

Subsequently, the VP architecture is improved to increase its functionality and configurability as well as its throughput. The new version is called virtualized SMT VP. By combining the proposed dynamic lane configuration and fusion techniques in the design of a shared virtualized SMT VP, the VP's energy consumption and energy runtime product are improved substantially under two proposed optimization policies. As VPs scale up in the number of vector lanes, fine-grain power management provided by lane configuration becomes more critical. The benefit of the fusion technique will also be amplified when the fusion degree grows above two.

9.2 Future Work

The VP pipeline can be improved to support more operations, such as the square root and negation. Adding a data reduction instruction to the VP for fully associative operations

(e.g. adding all the elements in a vector register) would also be extremely beneficial. These kinds of instructions were always performed on the scalar host for all previously proposed VPs and resulted in performance degradation for several practical applications.

Developing a distributed, rather than a centralized system, where each scalar core has access to more than one virtualized SMT VP can further improve parallelism. In this system, instruction fusion will be applied on top of VP virtualization to increase DLP exploitation. In fact, with instruction fusion the effective rate of issuing vector instructions will be doubled, while the total dynamic energy consumption due to vector application flow control on the host processor will be reduced by 50%. In such a system, each host can request fusion in order to achieve higher performance by exploiting more parallelism. In the case of request granting, the host can take advantage of many vector logical threads to run an application. Similar copies of a vector instruction will be sent to different virtualized vector logical cores to perform the same function on different resources. These virtualized vector cores can be located within the same or different physical vector coprocessors. A very abstracted overall architecture of such a distributed system is presented in Figure 9.1. Each virtualized SMT VP in this system will provide two logical cores to perform SMT.

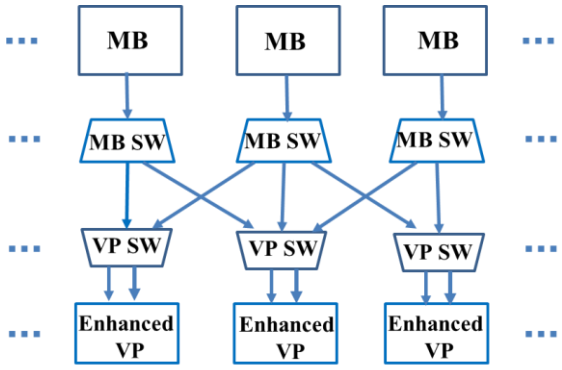

Figure 9.1 Abstracted architecture of a distributed system with enhanced VPs.

Having all the profiled data regarding performance and power analysis, a comprehensive high level model of the system can be developed in C. This model will be used for further exploration.

REFERENCES

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.

Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90), pp. 297-301.

Espasa, R., & Valero, M. (1997). Simultaneous multithreaded vector architecture: Merging ILP and DLP for high performance. *4th IEEE International Conference on High-Performance Computing*, pp. 350-357.

Hagiescu, A., and Wong, W. F. (2011, February). Co-synthesis of FPGA-based application-specific floating point SIMD accelerators. *19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 247-256.

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.

Kennedy, K., & McKinley, K. S. (1992, August). Optimizing for parallelism and data locality. *6th international conference on Supercomputing*, pp. 323-334.

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.

Kozyrakis, C., & Patterson, D. (2002, November). Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. *35th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 283-293.

Kuon, I., & Rose, J. (2007, February). Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 26 (2), pp. 203-215.

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.

Lu. Y, Rooholamin. SA, & Ziavras. S.G. (2015, Oct). Vector Processor Virtualization for Simultaneous Multithreading, *ACM Transactions on Embedded Computing Systems,* Accepted for publication.

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.*

Open Cores. 2012. http://www.opencores.org/projects (accessed on Sept. 10, 2013).

Rakvic, R., González, J., Cai, Q., Chaparro, P., Magklis, G & A. Gonzalez, A. (2010) Energy efficiency via thread fusion and value reuse. *IET Computers Digital Techniques*., 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.

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. *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.

Suresh, S., Beldianu, S. F., & Ziavras, S. G. (2013, June). FPGA and ASIC square root designs for high performance and power efficiency. *24th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 269-272.

Xilinx INC. 2010. MicroBlaze Processor Reference Guide, http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf (accessed on Oct. 10, 2013).

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 Oct. 10, 2013).

Xilinx INC. 2011. Power Methodology Guide, http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/ug786_PowerMethodology.pdf (accessed on Jan. 15, 2014).

Yang, H., & Ziavras, S. G. (2005, September). FPGA-based vector processor for algebraic equation solvers. *IEEE International Conference on System on Chip*, 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. *16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pp. 222-232.