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 COPROCESSOR SHARING TECHNIQUES FOR MULTICORES: PERFORMANCE AND ENERGY GAINS

by
Spiridon Florin Beldianu

Vector Processors (VPs) created the breakthroughs needed for the emergence of computational science many years ago. All commercial computing architectures on the market today contain some form of vector or SIMD processing.

Many high-performance and embedded applications, often dealing with streams of data, cannot efficiently utilize dedicated vector processors for various reasons: limited percentage of sustained vector code due to substantial flow control; inherent small parallelism or the frequent involvement of operating system tasks; varying vector length across applications or within a single application; data dependencies within short sequences of instructions, a problem further exacerbated without loop unrolling or other compiler optimization techniques. Additionally, existing rigid SIMD architectures cannot tolerate efficiently dynamic application environments with many cores that may require the runtime adjustment of assigned vector resources in order to operate at desired energy/performance levels.

To simultaneously alleviate these drawbacks of rigid lane-based VP architectures, while also releasing on-chip real estate for other important design choices, the *first part* of this research proposes three architectural contexts for the implementation of a shared vector coprocessor in multicore processors. Sharing an expensive resource among multiple cores increases the efficiency of the functional units and the overall system throughput. The *second part* of the dissertation regards the evaluation and

characterization of the three proposed shared vector architectures from the performance and power perspectives on an FPGA (Field-Programmable Gate Array) prototype. The *third part* of this work introduces performance and power estimation models based on observations deduced from the experimental results. The results show the opportunity to adaptively adjust the number of vector lanes assigned to individual cores or processing threads in order to minimize various energy-performance metrics on modern vector-capable multicore processors that run applications with dynamic workloads. Therefore, the *fourth part* of this research focuses on the development of a fine-to-coarse grain power management technique and a relevant adaptive hardware/software infrastructure which dynamically adjusts the assigned VP resources (number of vector lanes) in order to minimize the energy consumption for applications with dynamic workloads. In order to remove the inherent limitations imposed by FPGA technologies, the *fifth part* of this work consists of implementing an ASIC (Application Specific Integrated Circuit) version of the shared VP towards precise performance-energy studies involving high-performance vector processing in multicore environments.

VECTOR COPROCESSOR SHARING TECHNIQUES FOR MULTICORES: PERFORMANCE AND ENERGY GAINS

by
Spiridon Florin Beldianu

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 Computer Engineering

Department of Electrical and Computer Engineering

May 2012

.

APPROVAL PAGE

VECTOR COPROCESSOR SHARING TECHNIQUES FOR MULTICORES: PERFORMANCE AND ENERGY GAINS

Spiridon Florin Beldianu

Dr. Sotirios G. Ziavras, Dissertation Advisor — Date
Professor of Electrical and Computer Engineering, NJIT

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

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

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

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

BIOGRAPHICAL SKETCH

Author: Spiridon Florin Beldianu

Degree: Doctor of Philosophy

Date: May 2012

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2012

- Master of Science in Electrical Engineering,
  "Gheorghe Asachi" Technical University, Iasi, Romania, 2002

- Bachelor of Science in Electrical Engineering,
  "Gheorghe Asachi" Technical University, Iasi, Romania, 2001

Major: Computer Engineering

Presentations and Publications:

Beldianu, S. F., and Ziavras, S.G., "Performance-Energy Optimizations for Shared Vector Accelerators in Multicores" submitted to IEEE Transactions on Computers, 2012.

Beldianu, S. F., and Ziavras, S.G., "ASIC Implementation of a Shared Vector Processor," to be submitted.

Beldianu, S. F., and Ziavras, S.G., "Multicore-based Vector Coprocessor Sharing for Performance and Energy Gains," accepted for publication, ACM Transactions on Embedded Computing Systems, 2012.

Beldianu, S. F., Dahlberg, C., Steele, T., and Ziavras, S.G., "Versatile Design of Shared Vector Coprocessors for Multicores" re-submitted to Elsevier Microprocessors and Microsystems: Embedded Hardware Design after a minor revision.

Beldianu, S. F., and Ziavras, S.G., "On-chip Vector Coprocessor Sharing for Multicores," Parallel, Distributed and Network-Based Processing (PDP), 19th Euromicro International Conference on, pp. 431-438, 9-11 Feb. 2011.

Beldianu, S.F., Rojas-Cessa, R., Oki, E., and Ziavras, S.G., "Scheduling for input-queued packet switches by a re-configurable parallel match evaluator," Communications Letters, IEEE , vol. 14, no. 4, pp. 357-359, April 2010.

Beldianu, S.F., Rojas-Cessa, R., Oki, E., and Ziavras, S.G., "Re-Configurable Parallel Match Evaluators Applied to Scheduling Schemes for Input-Queued Packet Switches," Computer Communications and Networks, (ICCCN 2009) Proceedings of 18th International Conference on , pp. 1-6, 3-6 Aug. 2009.

To my Family, with Love and Gratitude.

Familiei mele, cu multă dragoste și recunoștință.

ACKNOWLEDGMENT

In the first place, I would like to express my deepest appreciation to my adviser, Dr. Sotirios Ziavras for being my mentor throughout my PhD research process. As my adviser, he guided and motivated me to find and pursue my research topic. Through long and daily talks in his office during the first semesters I was able to identify an original and innovative research direction. His extraordinary skills to inspire new and original ideas, his in depth knowledge in a vast number of areas and ability to provide up to date valuable references guided me throughout my entire research. Also, I would like to thank him for his time, energy and patience in reviewing my entire text work.

I would like to thank Dr. Roberto Rojas-Cessa for serving in my dissertation committee and also for giving me the opportunity to work on another research topic (scheduling in packet switches). Also, I would like to extend my special thanks to Dr. Durga Misra, Dr. Edwin Hou and Dr. Alexandros V. Gerbessiotis for serving as members in my dissertation committee. Dr. Misra's VLSI courses and Dr. Hou's Computer Algorithms course have been of great value to me.

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

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

Additionally, I would like to thank my friends, Vlad, Roxana, Ciprian and Viorica for all the great and unforgettable moments we shared together during these years.

Finally, I would like to express my deepest gratitude to my Family - for their support and understanding. I am deeply grateful to my wonderful parents, Georgeta and Spiridon, for their unconditional support and for always being there for me, and to my sister and my brother, Mariana and Liviu for being so close to me (geographically and, the most important, emotionally). My final and special acknowledgment goes to my love and best friend, my wife Oana for her patience and support throughout these years.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF TABLES
(Continued)

LIST OF TABLES
(Continued)

Table **Page**

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
CMP	Chip Multi-Processor
CPU	Central Processing Unit
CTS	Coarse-grain Temporal Sharing
DLP	Data Level Parallelism
DMA	Direct Memory Access
DSP	Digital Signal Processor
EPI	Energy Per Instruction
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
HPC	High Performance Computer
ILP	Instruction Level Parallelism
IPC	Instructions Per Cycle
MB	MicroBlaze
MC	Memory Crossbar
MM	Matrix Multiplication
PLB	Processor Local Bus
QoS	Quality of Service
SaaS	Software as a Service
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SMT	Simultaneous Multithreading
SPM	Scratch Pad Memory
SpMV	Sparse Matrix Vector multiplication
TLB	Translation Lookaside Buffer

VC	Vector Controller
VLIW	Very Long Instruction Word
VLS	Vector Lane Sharing
VLSI	Very Large Scale Integration
VM	Vector Memory
VP	Vector Processor
VRF	Vector Register File

CHAPTER 1

INTRODUCTION

1.1 Multithreading and Multiprocessing

The two important techniques for throughput-oriented computing are multithreading and multiprocessing.

Multithreading is used to increase the instruction level parallelism (ILP) handled by superscalar processors since it stalled more than a decade ago. Due to the difficulty of further speeding up an ILP-constrained single thread or program most computer systems actually multi-task multiple threads or programs. This technique improves the overall system throughput by increasing the average number of executed Instructions Per Cycle (IPC). The basic hardware multithreading scheme, namely *coarse-grain*, consists of switching one stalled thread with another one that is ready to execute [Kurihara et al., 1991; Agarwal, 1992]. The thread switch takes less than a few clock cycles (usually one) and the active thread does not share the functional pipeline with any other thread. The extra hardware cost is the replicated program registers and some control registers (that form the context). Quick context switching can potentially hide long latency stalls and increase the overall throughput and utilization of a processor's resources. *Interleaved multithreading* (fine-grain multithreading) takes advantage of the relative independence between threads and allows switching processor's context in any cycle [Horowitz et al., 1994]. In a given cycle a processor issues instructions from one of the threads, and in the next clock cycle it switches to a different thread context and issues instructions from the new thread. The primary advantage of interleaved multithreading is that it can better tolerate short latency stalls and increase the overall throughput. In addition to coarse-

grain needs, hardware support consists of labeling each instruction with a thread ID, increasing the number of registers, and also incorporating larger caches and Translation Lookaside Buffers (TLB) in order to minimize the conflicts between different threads.

The most efficient type of multithreading, which is currently deployed in most of the desktop and server microprocessors, is Simultaneous Multithreading (SMT) [Tullsen et al., 1995; Eggers et al., 1997]. SMT alleviates limited per thread instruction level parallelism by allowing superscalar processors to issue instructions from multiple threads in every CPU cycle. The extra hardware support is rather minimal as compared with interleaved multithreading, and shared resources such as L1/L2 caches and TLBs have to be adjusted appropriately to accommodate larger numbers of active threads. Most of the current processing units are either high-end chip multiprocessors with SMT cores (e.g., Intel i3/5/7, with 2 or 4 threads per core; AMD Opteron series with 2 to 4 threads per core; IBM Power7 with 4 threads per core) or embedded single-core SMTs (e.g., Intel Atom Z series with two threads per core). A more aggressive approach, i.e., Thread-Level Speculation [Oplinger et al., 1999; Wang et al., 2002], allows the compiler to optimistically generate parallel threads even if the threads are not eventually proved to be independent. Minimal hardware support is needed to track at runtime data dependences between speculative threads, to buffer the speculative state and to recover from a failed speculation [Colohan et al., 2007].

Multiprocessing refers to the use of multiple central processing units (CPUs) coupled together in a computer system. There are many variations on the definition of multiprocessors. If not explicitly stated otherwise, the one that this work refers to is multiple CPUs on a single die, i.e., Chip Multi-Processors (CMPs) in a single VLSI chip.

Instead of focusing on super-scalar processors, processor designers have recently increased core counts for CMPs. The emergence of multicores is caused mainly by:

(i) The fast evolution of VLSI technologies, such that the ever increasing number of transistors per unit area, made it possible to accommodate multiple cores on a single die (Moore's Law).

(ii) *Memory Wall*: The CPU performance has increased much faster than the memory performance, and now the memory performance becomes the bottleneck in many applications. Traditional Symmetric Multi-Processor (SMP) systems share the memory bandwidth among processors, further reducing the performance. For these systems, the traditional way to improve the memory performance by incorporating many levels of even larger caches has reached the point of diminishing returns.

(iii) *Frequency Wall*: To accommodate more threads and keep the frequency high, SMT requires deeper pipelines. Increasing the length of the pipeline increases the chances of resource conflicts in the instruction stream that will stall the pipeline or will cause a high cost for missed branches, thus reaching the point of diminishing returns [Chishti and Vijaykumar, 2008].

(iv) *Power Wall*: As the SMT processor tries to accommodate more threads and increase the frequency, the power consumption per operation increases dramatically as compared with CMP. More threads in SMT require a larger register file, larger data and address caches (TLBs) and more complex control logic. This comes at the cost of dynamic power and, more recently substantially increased leakage power (due to larger area), with no substantial performance improvement as the number of threads increases. It has been shown [Sasanka et al., 2004] that as the number of simultaneous threads per core increases, the Energy Per Instruction (EPI) at the same performance point gets higher than the EPI for a CMP. The main cause is contention for limited resources among threads that produces extra cache and TLB misses and, thus, more energy consumption for the same IPC. Also, increasing the operational frequency in a SMT processor increases the power consumption due to at least two factors: (a)

a higher frequency requires either increasing the voltage supply level (V*dd*) or decreasing the threshold voltage (V*th*); as the dynamic power is proportional to the square of V*dd*, the active power will eventually increase quadratically; (b) the static power also increases linearly with V*dd* and decreases exponentially with V*th* [Butts and Sohi, 2000]. Some decisions can be made that minimize the amount of interaction between threads. This minimization is accomplished by choosing threads that access different regions of the cache or different computational resources [Kihm et al., 2005].

(v) Small time to market pressure and reduced cost requirements necessitate the reuse of off-the-shelf uni-processor IPs when building multiprocessors. The new IP for multicores consists of the glue logic (interconnection) and minimal verification primarily focusing on the interconnection logic. It is much easier to replicate already tested cores than just improving a single out-of-order superscalar core.

(vi) The emergence of the Software as a Service (SaaS) paradigm [Wang et.al, 2011], is now deployed in datacenters. Amdahl's Law is often replaced by Gustafson's law [Gustafson, 1988] which states that problems with large and repetitive data sets can be efficiently parallelized (they have a high DLP or data level parallelism).

Since the mid 2000's designers have increased the number of cores per chip rather than focusing on single-core performance. However, a new limit on multicore scaling will soon make this approach less useful, thus creating a transistor utility economics wall in relation to underutilized resources (called dark silicon). A recent study, that takes into consideration the device, core and CMP scaling models, showed that regardless of chip organization and topology, a large area of the chip will have to be powered down [Esmaeilzadeh et al., 2011]. For example, at 22 nm (to be available soon), the study suggests that 21% of a chip must be off, and this number grows to more than 50% with 8 nm. Moreover, according to their unified model, in the next decade only an average

speedup of eight will be possible for common parallel workloads; this will yield a substantial gap (up to 24) between the expected and actual performance; ideally, each newer generation silicon technology node is currently expected to double the performance.

Scaling the performance and energy could be achieved by:

(i) Scaling the off-chip memory bandwidth capacity and the overheads associated with the process of moving data [Rogers et al., 2009].

(ii) Reducing the energy overheads associated with useful operations [Horowitz et al., 2011]. This further requires reducing the energy overheads on the instruction path and the instruction memory hierarchy.

The first requirement could be addressed by using a heterogeneous memory hierarchy, that is, by employing memories that are not fully cacheable but rather explicitly managed. This category includes Scratch Pad Memories (SPM) or Local Stores (LS) [Flachs et al., 2005]. SPM reduces the energy consumption by almost 40% and the area by 34% for applications with regular memory accesses [Banakar et al., 2002; Milidonis et al., 2009]. Unlike caches, it is the programmer's responsibility (possibly with the help of the compiler) to explicitly manage data transfers between the main memory and the SPM. The applications that can fully use SPM are scientific and multimedia (streaming) applications where data movement could be managed explicitly and uniformly between off-chip memory and stream processors. Applications that have a low degree of parallelism could be mapped to scalar cores such that the memory transfers can make use of the cache [Kudlur et al., 2008].

The second problem could be addressed by using more specialized cores for each task based on heterogeneous computing. This category includes ASIC custom designs

specialized for a single application, like video compression and encryption engines, Vector Processors (VP) operating in the Single Instruction Multiple Data (SIMD) mode, Graphical Processing Units (GPUs) operating in the Single Instruction Multiple Threads (SIMT) mode, and Digital Signal Processors (DSPs).

1.2 Related Work on Vector Processors

Vector code offers a compact, predictable, single-threaded programming model, with the possibility for loop unrolling to be performed directly at the hardware level under branch prediction. Moreover, the already compiled vector object code can directly benefit from new implementations even if some rescheduling is required for optimal performance on new SIMD micro-architectures. In recent years, SIMD extensions have become ubiquitous. Even scalar processors on the market today contain them in some form. Since the focus of this work is the vector processor architecture, an overview of this architecture is presented in the following sections. Section 1.2.1 presents the architecture of a modern vector processor, Section 1.2.2 presents an overview of high performance applications and vector processors used in supercomputers and Section 1.2.3 presents the emerging SIMD architectures targeting embedded applications.

1.2.1 Modern Vector Processor Architectures

Vector Computers created the breakthrough needed for the emergence of computational science. The vector architecture was first fully exploited with Cray-1 in 1976 [Russell, 1978]. Cray had a register file with eight vector registers which held 64 64-bit words each and achieved a peak performance of 240 MFLOPs. In the 1980s NEC introduced its first vector system (SX-2) which was an improved version of Cray-1. The vector processor

simultaneously performed mathematical operations on multiple data elements from an array, called *vector*, by instructions named vector instructions. A modern vector processor falls into the SIMD category, and usually consists of a scalar unit and a vector unit as shown in Figure 1.1. The scalar unit is similar to an ordinary pipelined scalar processor which executes scalar instructions for control functions, the unvectorizable part of the operating system and application code. The vector unit consists of vector registers, pipelined arithmetic unit(s) and a pipelined Load/Store unit (L/S). Most of the modern vector processors implement a register-bank scheme for the vector register file (VRF) [Asanovic, 1998]. By interleaving vector register storage across multiple banks, the number of ports required on each bank can be reduced. A separate interconnection network connects banks and arithmetic pipeline ports. In effect, all of these bank partitioning schemes reduce the connectivity between element storage and arithmetic unit ports. As depicted in Figure 1.1, in a lane based modern Vector Processor, a single vector register (VR0) with length VL can be low interleaved across M lanes resulting into VL/M elements from each vector register in a single lane. A vector lane is an independent vector subunit containing its own bus interfaces, processing units and vector registers; during its operation it does not compete for resources with any other lane, except for external accesses going to the same memory modules. As a note, the maximum number of elements to be held in a vector register, Maximum Vector Length, is 64 in Cray-1 and 256 in the NEC SX systems. The pipelined arithmetic units from each lane usually implement Add, Multiply, Divide, Logical and Shift operations in a pipelined fashion, in which the vector data are input from vector registers and the results are output every clock cycle into the vector registers.

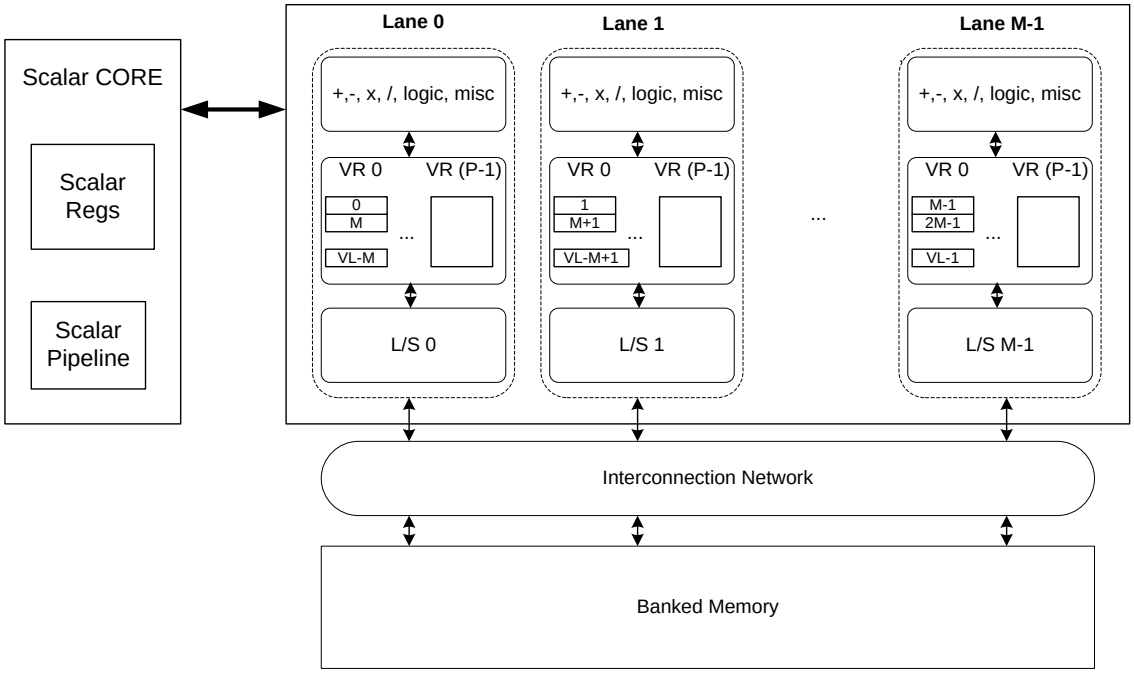


Figure 1.1 Lane based modern Vector Processor architecture.

Vectorization is the process of converting a computer program to a sequence of vector instructions for executing on a vector processor. Figure 1.2 shows a `for` loop in C code and the produced vector instructions. The vector length is 256 and each vector instruction processes 256 elements.

```
for (i=0; i<256; i++) {
     A(i)=B(i)+C(i)*D(i);
}
```

```
VLD VR0,C           ; VR0 <- (C)
VLD VR1,D           ; VR1 <- (D)
VMUL VR2,VR0, VR1   ; VR2 <- VR0*VR1
VLD VR3, B          ; VR3 <- (B)
VADD VR4, VR2, VR3  ; VR4 <- VR2*VR3
VST VR4, A          ; VR4 -> (A)
```

Figure 1.2 Source code and the produced vector instructions (VL is 256 and the scalar instructions are omitted for simplicity).

1.2.2 Vector Processors for High Performance Computing (HPC)

Computer modeling and simulations of physical phenomena and engineered systems have become widely spread in supporting theory and experimentation. High Performance Computers (HPC) are used in weather and climate research, bioscience, energy, military, automotive and many other engineering fields.

Introduced in 2002, the Cray-X1 vector supercomputer has a hierarchical design with the basic building block being the multi-streaming processor (MSP), which is capable of 12.8 GF/s for 64-bit operations (or 25.6 GF/s for 32-bit operations). Each MSP contains four single-streaming processors (SSPs), each with two 32-stage 64-bit floating-point vector units and one two-way super-scalar unit. The SSP uses two clock frequencies, 400 MHz for the scalar core and 800 MHz for the vector units. Each SSP is capable of 3.2 GF/s for 64-bit operations [Dunigan et al., 2005]. The NEC SX-9 processor runs at a frequency of 3.2 GHz and has 8 vector pipes (or lanes), each having two multiply units and two addition units; this results in a peak vector performance of 102.4 GF/s [Kobayashi et al., 2009]. For non-vectorized code, there is a scalar processor that runs at half the speed of the vector unit, i.e., 1.6 GHz. The NEC SX family is the only classic vector architecture which is still deployed in current supercomputers. The other major vendors (Cray, Fujitsu, Hitachi) have discontinued their (dedicated) vector product lines and adopted commodity scalar-based multiprocessors. Most of the Cray supercomputers are using AMD Opteron cores, Fujitsu adopted a SPARC architecture for its fastest supercomputer in the world as of November 2011 (K supercomputer with SPARC 64 VIIFX cores), and Hitachi adopted IBM POWER7 cores in its latest SR1600 supercomputer. Instead of improving a vector architecture with high time, design and

verification costs, supercomputer vendors started to use widely spread CMPs as the basic building block. The main reasons for CMP-based supercomputing are generality, scalability, low time to market and cost effectiveness. However, it has been reported that there is an increasing gap between the theoretical peak performance and the sustained system performance for High End Computing systems of major US high-end computing centers [Federal HPC Rep, 2004]. In other words, the commodity-based scalar systems have difficulty obtaining the high computation efficiency in the execution of real scientific and engineering applications. And, on top of that, the energy efficiency (MFLOPs/Watt) has been decreased dramatically. On the other hand, vector supercomputers achieve high sustained performance and high computation efficiency in various scientific and engineering applications [Oliker et al., 2008; Musa, 2009].

As a consequence, two distinct supercomputer architectures have emerged recently. The first one is hybrid, and one of its incarnations is the IBM Roadrunner**.** The hybrid design has in each node an IBM PowerXCell 8i attached to an AMD Opteron CMP [Barker et al., 2008]. The IBM PowerXCell architecture comprises one general purpose core (PPE), and eight special streaming processor elements (SPE) for floating point operations. The vectorized code is mapped to SPEs and the scalar part of the applications runs either on the Opteron or the PPE. This is an example of reducing the gap between sustainable and peak performance in modern supercomputers. Also, the IBM Cell-based supercomputers have been reported to be some of the most power efficient supercomputers [Green 500 List, 2011]. The second architecture contains heterogeneous CPU-GPU nodes, that is, a low latency scalar-based architecture (Intel i7, Intel Xeon, AMD Opteron) combined with a high processing throughput SIMT

architecture (NVIDIA or ATI Graphic Processor Units - GPU) [Nickolls and Dally, 2010]. The second fastest supercomputer in the world and the one in the 13th place in the green list (as of Nov 2011) belongs to this category. As its building block, it contains an Intel Xeon X5670 2.93 GHz processor and an Nvidia Tesla M2050 general purpose GPU (GPGPU) as the accelerator for high intensive parallel tasks; it reaches a maximum peak performance of 2.5 PetaFlops with around 4MW power consumption [Top 500 List, 2011].

As a conclusion, the supercomputing architectures are going back to a form of SI(MD/MT) in order to support high performance throughput with low power and area costs.

1.2.3 Emerging SIMD and Vector Architectures

In a SMT superscalar, more and more area and, obviously power, is consumed by complex structures required to support speculative, out-of-order superscalar execution (for instruction fetch, decode, register renaming, and control of the instruction window components, including speculation recovery). More area/power budget assigned to the instruction data-path and less to the processing core (functional units) leaves less room and power budget to the integer and floating processing data paths. On the contrary, the vector processor approach has more resources allocated to functional units which prove to be more efficient in terms of performance and energy for a broad class of applications [Lemuet et al., 2006; Who et al., 2008]. It has been shown recently that SIMD-based accelerators can handle regular and irregular DLP efficiently and still retain programmability [Krashinky et al. 2008; Lee et al., 2011]. Also, SIMD architectures are

more area and energy efficient than multi-scalar based microarchitectures, even for fairly irregular DLP.

VIRAM [Kozyrakis and Patterson, 2003b], SODA [Lin et al. 2006] and AnySP [Woh et al., 2010] are single-chip vector microprocessors, and their instruction sets support a comprehensive set of vector operations. Vector microprocessors have been shown to be more effective in embedded media applications than superscalar and VLIW processors [Kozyrakis and Patterson, 2002]. Also, a 2-dimensional (matrix-oriented) SIMD extension was developed in [Sanchez et al., 2005]. Due to recent advances in programmable devices that have increased substantially their logic cell densities, some Field-Programmable Gate Array (FPGA) based soft vector processors have been proposed as well [Yang and Ziavras, 2005; Cho et al. 2006; Yiannacouras et al. 2008; Yu et al. 2009]. An automated co-design tool chain in [Hagiescu and Wong, 2011] produces SIMD hardware accelerators and appropriate software for performance and energy gains. The VEGAS soft vector architecture in [Chou et al., 2011] is attached to a single soft Nios II/f Altera processor. It comprises a parameterized number of vector lanes, a scratchpad memory and a crossbar network for shuffle vector operations.

Multimedia (MMX) and streaming SIMD extensions (SSE1-4, AVX) are currently popular in commercial microprocessor architectures. They have been shown to provide a significant boost for a few key multimedia applications without requiring much silicon area. The most advanced SIMD extension, the Intel AVX [Yuffe et al., Intel, 2011], increased the vector length to eight elements (from four in previous implementations) for the single precision floating point format. However, these extensions have several disadvantages compared to a more comprehensive vector

approach: (i) the vectors are short; therefore, each instruction caries a small amount of data-path work. This requires high instruction issue bandwidth in order to keep the SIMD pipelines busy and consumes additional energy on the instruction path; (ii) memory load/store operations have to be aligned to 16 (32 in Intel AVX) bytes boundaries, and (iii) the data transfers between extended registers and the main memory are performed via a cache. In low-end mobile devices the most common architecture is ARM NEON [Rintaluoma and Silven, 2010]. The SIMD data path width is 128 bits (four single precision floating point elements) and it is designed to provide acceleration for mobile multimedia applications with low power budgets.

A multithreading technique for a vector processor architecture was first introduced by Espasa and Valero [Espasa and Valero, 1997] assuming a Cray C3400 vector machine model. The work shows that multiple threads can increase the utilization of memory ports and the overall throughput while also and hiding long memory access latencies. However, since their baseline vector architecture is modeled after the Convex C3400, the model assumes one vector pipeline only (i.e., one lane) and restrictive memory model. In [Rivoire et al., 2006] a Vector Lane Threading (VLT) architecture is introduced. It partitions the vector lane space among multiple threads and is suitable for applications with small vector lengths (a VL that is less than the number of lanes) that cannot take advantage of a wide Vector Processor. The finest granularity is one vector thread per lane, i.e., the vector length is one. Each thread (in a lane partition) requires separate control signals with substantial overheads that can become unbearable as the number of threads increases. Also, in any single lane only one thread (context) exists at any time.

In [Sasanka et al., 2007] an All Levels of Parallelism (ALP) model is presented; it is based on a conventional superscalar model with SMT and SIMD (MMX and SSE2 extensions) capabilities. The SIMD architecture is enhanced with SIMD vectors and streams (SVectors/SStreams). The application model builds a new data structure on top of the SIMD word, namely it creates a record or Stream Vector. Stream Vectors are implemented directly in the L1 cache allowing compute SIMD instructions to directly access them using existing data paths, without additional loads and stores. Even if this architecture applies well to an existing SSE extension it still retains the drawbacks of SIMD instructions with short vector lengths for compute operations and the usage of a cacheable memory in streaming applications.

1.3 Motivation and Objectives

Many high-performance and embedded applications dealing with streams of data cannot efficiently utilize dedicated vector processors for various reasons. Firstly, individual programs often display limited percentage of vector code due to substantial flow control or involved operating system tasks. The utilization of an available VP is then proportional to the vectorized part of the code; therefore, the rest of the time the VP will be idle [Azevedo and Juurlink, 2009]. Secondly, even with substantial vector code, the needed vector length may often vary across applications or within a single application, as in multimedia [Woh et al., 2010]. Thirdly, several applications have many data dependencies within sequences of instructions, a problem exacerbated further without loop unrolling or other compiler optimization techniques [Gerneth, 2010]. And, finally, as the computational intensity (the ratio of arithmetic operations to memory references) decreases, the utilization of the functional units goes down. The computational intensity

depends on the application and the memory hierarchy [White et al., 2005]. Moreover, for a given application, as the number of nodes increases in a scalable system, the computational efficiency per node decreases and also its utilization. Fewer computational resources are utilized and the need arises to adaptively adjust the active resources. Such limitations deter efforts to sustain high SIMD utilization, especially for superpipelined floating-point units (FPUs).

For example, Cray X1 achieves a sustained performance of 30% of its peak performance for sparse matrix based applications, 65% for dense matrix multiplication based applications and almost 50% for FFT based applications. The main cause is the limited off-chip memory bandwidth [Cray X1 Rep., 2004]. Also, a more recent vector supercomputer, the shared-memory NEC SX-9 vector system achieved a sustained rate of 68.8% of its peak performance for Earthquake (dense MM kernel), 55% for Turbulent Flow and Antenna (Fast Fourier Transform), and around 17% for LandMine, Turbine and Plasma (sparse kernels with irregular memory accesses) applications [Soga et al., 2009]. To sustain a low bandwidth requirement per flop, NEC SX-9 adds a software controllable on-chip cache, the Assignable Data Buffer (ADB), similar to SPMs. In 1993 vector supercomputers occupied 67% of the positions on the TOP500 list; however, as stated in Section 1.2.1, the number of vector supercomputers has decreased over the years, and GPP-based (AMD Opteron, Intel Xeon, IMB Power) clusters have dominated the TOP500 list. In systems with standard and scalar cores, it is even harder to achieve high efficiency (sustained performance over peak performance ratio) for SIMD pipelines, especially for non-unit stride and irregular scatter/gather operations. For scientific applications, these processors demonstrate a very low utilization of functional units.

Thus, either the SIMD units that reside in scalar processors or the vector pipelines controlled by scalar cores are highly underutilized.

Therefore, actual SIMD/vector architectures need:

(i) High utilization of the SIMD/vector pipelines. This could be achieved in two ways: (a) from the software perspective: a level of ILP and/or DLP parallelism that can provide a level of SIMD instruction throughput which will produce high utilization of the vector units, and (b) from the hardware perspective: to *share* expensive resources, such as VP lanes, between multiple cores in order to aggregate the SIMD instruction streams and produce high throughput on the data path. Allocating silicon area to an SIMD resource which is tightly coupled to a single core leaves less room for dynamic scheduling options in a multicore system while also consuming substantial leakage power as a percentage of the core's power budget.

(ii) Flexible vector length (VL) as per application (kernel) needs; that is, dynamic VL per thread transparent to the programmer. As stated previously, issuing the same instruction multiple times to perform identical jobs is not efficient since this consumes power on the instruction path and also requires frequent branch implementation with its associated overheads. Therefore, there is a requirement to adjust the vector length to the application needs.

(iii) Quality of Service (QoS) at the hardware level. Sharing expensive SIMD units among multiple threads requires QoS such that each thread that utilizes the units gets the desired level of throughput.

(iv) Performance-power tradeoff. Some vector applications may require low energy consumption with no time constraints; others may have performance as the first priority. Thus, there is a need to create a framework that adjusts the used vector computing resources based on given performance-energy constraints.

(v) Reduced impact of the static power on the total energy budget. Static power due to leakage currents will become an even larger source of power consumption in future technologies. The shrinking of transistors yields increased static power

contribution to the total energy consumption [Keating et al., 2007]. Particularly, for the 45nm technology generation and beyond, leakage power consumption catches up with, and sometimes dominates dynamic power consumption. Thus, as the number of resources/cores working in parallel increases, the consumed static energy increases almost linearly. However, the actual performance does not scale correspondingly and, as a consequence, the contribution of the static energy to the total energy budget increases. Additionally to static power, there is another power component that is consumed even when the device does not perform any useful operation; i.e., clock network power. Combined, these two components form the standby power (also known as idle power or no-load power).

(vi) VP sharing designs for multicores that can facilitate runtime resource and power management involving a good balance of performance and energy consumption. In contrast, a dedicated VP per core leaves much less room for runtime power management. Such management for shared VPs should introduce small timing and energy overheads. The objective should be the development of efficient power-gating techniques in relation to VP lane sharing. Existing rigid SIMD architectures cannot tolerate efficiently dynamic application environments with many cores that may require the runtime adjustment of assigned vector resources in order to operate at desired energy/performance levels that change frequently.

To simultaneously alleviate these drawbacks of rigid lane-based VP architectures while also releasing on-chip real estate for other important design choices, *the first objective* of this research is to propose three architectural contexts for the implementation of a shared vector coprocessor in a multicore environment. Sharing an expensive resource among multiple cores will increase the efficiency of functional units and the overall throughput. As presented is Figure 1.1, the baseline VP architecture is lane-based with a banked vector register file. *Coarse-grain temporal sharing* (CTS) consists of temporally

multiplexing sequences of vector instructions ideally arriving from different threads. However, providing a per-core exclusive access to the vector resources does not maximize their utilization. Derived from GPP-based SMT architectures, *Fine-grain Temporal Sharing* (FTS) consists of spatially multiplexing individual instructions issued by different scalar processors in order to increase the utilization of the vector units. Finally, *Vector-Lane Sharing* (VLS) consists of simultaneously allocating distinct vector lanes or collections of them to distinct scalar cores.

A second objective regards the evaluation and characterization of the three shared vector architectures from the performance and power perspectives. The performance and energy consumptions for these coprocessor sharing contexts are evaluated by implementing several floating-point applications on an FPGA-based prototype. A performance model for these coprocessor sharing contexts is presented as well as a power estimation model based on observations deduced from experimental results. These models suggest several techniques to increase the performance or reduce the energy consumption:

(i) Increase the data-level parallelism by increasing the vector length.

(ii) Increase the instruction-level parallelism at compile time by loop unrolling or other techniques.

(iii) Use multiple threads in a multiprocessor environment to increase the vector coprocessor utilization.

(iv) If none of the above is possible, adjust the VP resources in order to minimize a given energy/performance metric.

The analysis shows that technique (iii) can be superior to the former two combined. Therefore, the lack of adequate data-level parallelism in an application can be overcome by sharing the coprocessor resources among many cores.

The results show the necessity to create a HW/SW framework that adaptively adjusts the size of the vector processor in order to minimize the total energy consumption on a modern vector processor that runs applications with dynamic workloads. Therefore, the *third objective* is to develop an energy consumption estimation model and, based on this model, a hybrid fine-to-coarse grain power gating (PG) technique and relevant adaptive HW/SW support. Two approaches are possible: (i) at static time, apriori information about the application that needs to run (utilization, level of data/instruction parallelism, etc.) could be used by the SW to estimate the number of lanes for which the energy is minimized; and, (ii) at runtime, using embedded performance and/or energy counters that monitor the utilization of the lanes, a decision on how to shrink/enlarge the VP (i.e., adjust the number of lanes) has to be taken as fast as possible and with minimal energy impact. The energy metric can be used when the device is battery powered and there is no constraint on performance. However, this metric does not allow trade-off between power and delay. The energy delay product favors performance over energy and also measures the quality of a CMOS design [Sengupta and Saleh, 2007; Martin et al., 2001]. Therefore, a performance-energy tradeoff mechanism which gives priority to performance at the expense of more energy consumption is also introduced as part of this objective.

Finally, the *fourth objective* of this work is to implement the VP hardware design in ASIC using Synopsys design flow tools. Moving the entire FPGA-based design to an ASIC implementation will face a few challenges:

(i) Changing the proprietary IP cores, such as BRAMs and floating point units, with SRAM blocks and custom IPs.
(ii) Optimizing the ASIC design for speed and power in a given technology.
(iii) Evaluating different design options and the impact of their static energy and other standby components on the total energy budget.
(iv) Modeling performance and power.

This thesis is organized as follows. Chapter 2 presents the VP sharing architecture in detail and its implementation on an FPGA device. Chapter 3 describes the software development process and presents popular vector-dominant floating-point applications used to test out VP architecture. Performance, power and energy results are presented in Chapter 4 and are followed by a comparative analysis. Chapter 5 describes performance and power models, and introduces the opportunity to trade the energy and performance. Chapter 6 introduces two energy minimization techniques and a performance-energy trade-off mechanism. The ASIC implementation and relevant results are presented in Chapter 7. Finally, Chapter 8 draws conclusions and presents future work objectives.

CHAPTER 2

VECTOR COPROCESSOR SHARING

The main difference of this work from [Kozyrakis and Patterson, 2003b; Woh et al., 2010; Yu et al., 2009] consists of introducing (a) an architecture for lane based vector coprocessor design that can integrate mechanisms for the coarse-grain and fine-grain mixing of threads issued by one or multiple cores, (b) configurable vector lanes that can be grouped for assignment to distinct cores in a manner that eliminates internal resource conflicts, as well as (c) configurable vector register length. The main objective, as compared with all previous aforementioned works where just one thread can use the entirety of the VP resources, is to provide a hybrid VP architecture framework for sharing the vector coprocessor among multiple scalar cores. This architecture is even more suitable for shared-bus multicores, the current focus of commercial multicore technology.

The rest of the chapter is organized as follows. Section 2.1 presents the details of three basic vector-sharing architectures. Section 2.2 presents VP sharing architecture in detail, and Section 2.3 presents resource consumption and synthesis frequency figures.

2.1 VP Sharing Techniques

In order to increase the overall utilization and throughput of a VP embedded into a multicore chip, a mechanism must be developed for its simultaneous sharing by multiple cores. The terms scalar processor and core processor will be used interchangeably from now on. Sharing could also support multithreading inside the VP with the threads coming from one or more applications. Unlike VP architectures for single cores which are

designed with a fixed SIMD width (i.e., vector register size) aiming to service one application at a time, this work proposes adaptive VP sharing for multicores in order to support multiple-SIMD execution relying on thread-level parallelism (TLP). This design approach can maximize the VP utilization and throughput for two reasons:

(i) Different cores often handle different vector lengths, thus not being able to individually utilize dedicated VP resources fully. Also, applications have different natural vector lengths. Actual general purpose SIMD machines provide low vector length (4 for Single Precision Floating Point (SFFP) on ARM Neon and 8 for Intel AVX on the Sandy Bridge architecture [Rintaluoma and Silven, 2010; Yuffe et al., Intel, 2011]). Issuing multiple instructions to perform the same job is not efficient since this consumes power on the instruction path and also introduces unnecessary branches. Therefore, a better way is needed to adjust the vector length to the application needs is a requirement.

(ii) Different vector kernels in the same or different applications often have diverse VP-based computation needs [Woh et al., 2010].

To simultaneously alleviate these drawbacks of rigid VPs while also releasing on-chip real estate for other important design choices, this thesis proposes adaptive VP sharing for multicores that integrates three basic VP sharing architectures, namely *coarse-grain temporal* (CTS) *sharing*, *fine-grain temporal sharing* (FTS), and *vector lane sharing* (VLS) [Beldianu and Ziavras, 2011a]. This paper investigates power/energy consumption, and does not present any performance and power estimation models that could be used by the runtime system to fine-tune VP sharing at runtime (based on the needs of individual applications, or collections of them simultaneously competing for VP resources). VP system is implemented in the SystemVerilog high-level language and only performance benchmark results were recorded. In [Beldianu and Ziavras, 2011b] an improved VP sharing integration is presented, that, besides several architectural

improvements and new vector instructions, is implemented on an FPGA and is synthesized in VHDL. The implementation of various benchmarks on the target Xilinx FPGA device yields accurate figures for performance and power, thus leading to important conclusions about such versatile VP sharing systems. Also, a highly accurate performance and power estimation models is introduced. The rest of the chapter introduces the proposed VP sharing techniques and presents the details of VP sharing architectures.

2.1.1 Coarse-grain Temporal Sharing (CTS)

CTS sharing consists of temporally multiplexing the execution of sequences of vector instructions or threads containing them. A scalar processor takes exclusive control of the entire VP, and then releases it by executing a lock and unlock instruction, respectively. It runs a thread to completion or until it stalls due to a resource conflict (e.g., DMA access conflict). Such a stall forces thread switching for the VP. Figure 2.1 (a) shows how the CTS is performed; at any given time all lanes are processing only SIMD instructions issued by one scalar processor. CTS can alleviate the low utilization in a VP environment with an exclusive scalar core in cases where long sequences of scalar code are interleaved with long sequences of vector code (e.g., parallel programs which contain critical sections that need to be run on CPUs). No duplication of the vector register file is needed and only simple scheduling is required. Average utilization will be improved but the instantaneous utilization will not; thus, programs that need VP resources most of the time will not take advantage of this technique. Note that this context may be required also for kernels (programs) that need to run at full speed with no interference from other instruction streams coming from the other CPUs.

2.1.2 Vector Lane Sharing (VLS)

VLS lane sharing assumes a divisible VP consisting of independent vector lanes with their own execution units. A vector lane is an independent vector subunit containing its own bus interfaces, processing units and vector registers; during its operation it does not compete for resources with any other lane, except for external accesses going to the same memory modules. VLS facilitates the simultaneous allocation of distinct vector lanes, or collections of them, to distinct scalar processors for seamless processing. Based on the chosen set of vector-lane allocation and scheduling policies, a hardware scheduler external to the lanes determines at runtime how to group together vector lanes to meet the requirements of applications running on the cores. Therefore, if multiple cores simultaneously share the VP space, each core can use exclusive lanes forming a small-sized VP (as compared to the full-sized VP that comprises all of the lanes, say M). This technique is somehow similar with Vector Lane Threading from [Rivoire et al., 2006]. However, the main difference is that presented architecture does not provide a separate control bus for every lane partition (or sub-VP); instead, the lane is controlled by issuing the appropriate instructions to the assigned lanes. Figure 2.1 (b) shows one example of the VLS context. A VP with eight lanes in the figure is split into two VPs with four lanes each. Similar to CTS, at any given time a lane processes only instructions coming from a single CPU. However, an increase in the utilization is expected by increasing the number of elements per vector register.

Assuming a VP with *M* lanes, a fixed number of *K* elements in the VRF of each lane, and a vector length *VL* in an application that uses all the lanes, Equation 2.1 shows the number of vector registers (*VREGs*) available to the application.

$$VREGs = K\frac{M}{VL} \tag{2.1}$$

Since for a given architecture the number K of VRF elements in each lane is fixed, by reducing the number of lanes assigned to an application in the VLS mode either the number of vector registers or the VL has to be reduced. For programs requiring a substantial number of registers VL may need to be decreased. VLS proves useful when the degree of vectorization in an application running on a core is moderate, thus not requiring the full VP coprocessor space, or when the vector length required is less than the total number of available lanes. Also, VLS could be extended to cases where a VP subset simultaneously handles multiple threads issued by the same or different cores.

2.1.3 Fine-grain Temporal Sharing (FTS)

FTS sharing involves spatial (i.e., resource-based) multiplexing of vector instructions coming from different threads running on the same or different scalar processors. In the former case, the scalar runs in the SMT mode. A scalar issues an SIMD instruction in a given VP clock cycle according to a chosen arbitration scheme, the simplest one being round robin. The benefit of this approach is that the VP instantaneous utilization will be increased since data hazards do not exist between instructions issued by different threads or processors, and the VP resource idle times due to data transfers are eliminated or reduced. In FTS, vector instructions coming from different cores or threads can simultaneously execute in the same VP using the same pipelined resources (e.g., adder, multiplier, LDST unit). As shown in Chapter 4, this type of VP sharing provides the best performance and energy savings. Figure 2.1 (c) shows an example of two instructions issued by different CPUs coexisting inside the lane pipelines. One CPU issues

instructions with VL=32 and the other one with VL=16. According to the Equation 2.1, since two register contexts have to exist in each lane, the VRF resources have to be increased in order to allow two threads to run simultaneously.

2.2 VP Sharing Architecture

In order to validate the FTS, CTS and VLS vector-sharing contexts, the VP system is prototyped on a Xilinx FPGA device. Initially the design targeted a Virtex-5 XC5VLX110T FPGA device and later on it is ported to a Virtex-6 XC6VLX130T device. In order to avoid confusion, for all the subsequent chapters and sections, the appropriate device will be mentioned explicitly. The design consists of two scalar processors, an 8-way data-path partitioned VP with an 8-way vector memory load/store unit for parallel data memory accesses, a VP-memory interconnecting crossbar, and an 8-bank low-order interleaved on-chip vector memory. MicroBlaze, a 32-bit embedded RISC soft core provided by Xilinx, forms each scalar; it employs the Harvard architecture and uses the FSL interface to connect with up to eight coprocessors [Xilinx Inc., 2010b]. Instructions issued to VP use a 32-bit FSL bus. Since the Xilinx EDK (Embedded Development Kit) tool kit limits the operating frequency of MicroBlaze to 125 MHz, without loss of generality the entire design is optimized for this target frequency.

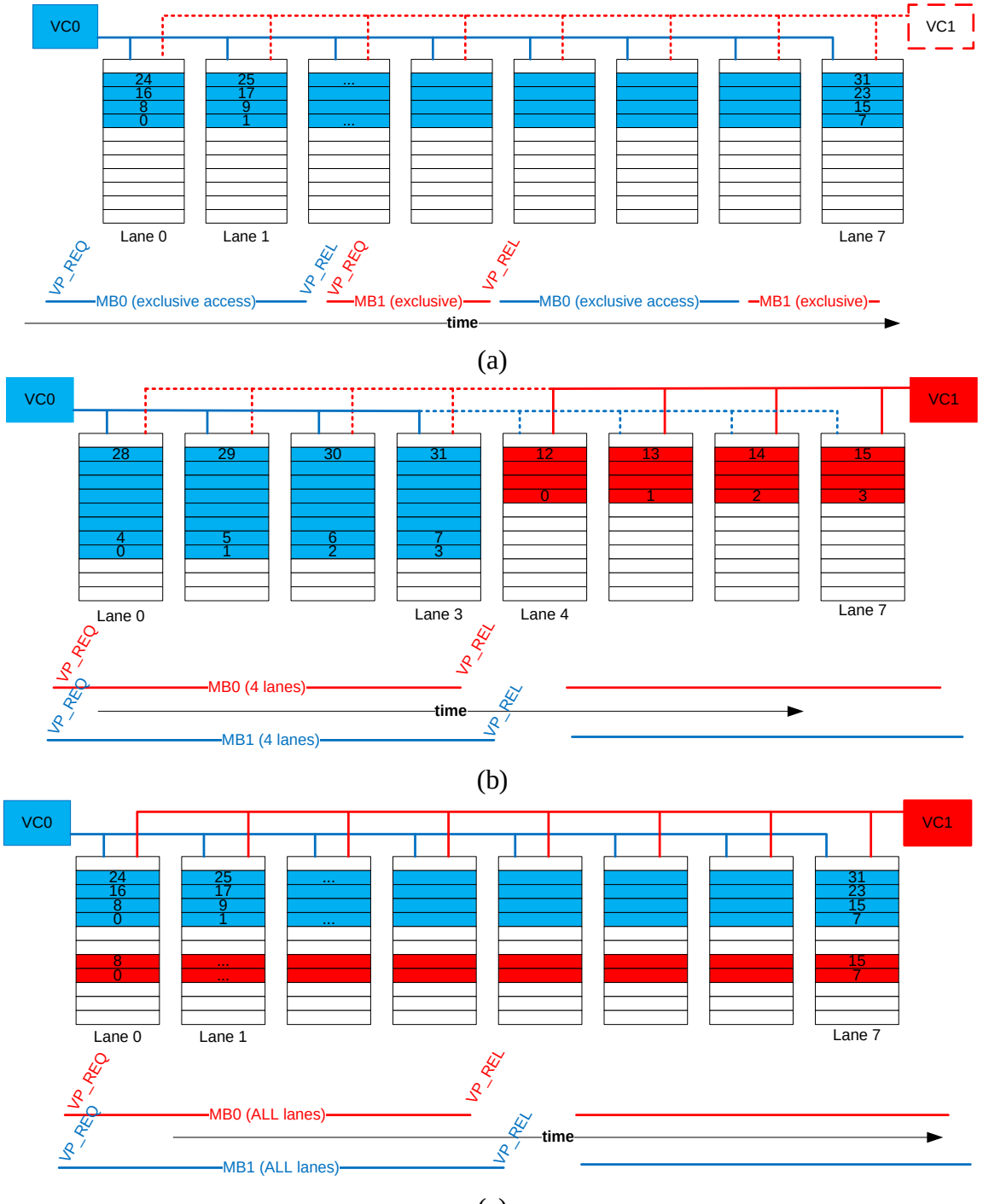


Figure 2.1 VP sharing contexts: (a) Coarse-grain temporal (CTS) sharing; (b) Vector lane sharing (VLS); and (c) Fine-grain temporal sharing (FTS). Each lane contains a fixed number of pipeline stages; colored boxes show the busy pipeline stages in each lane and white boxes are unused pipeline stages (pipeline bubbles).

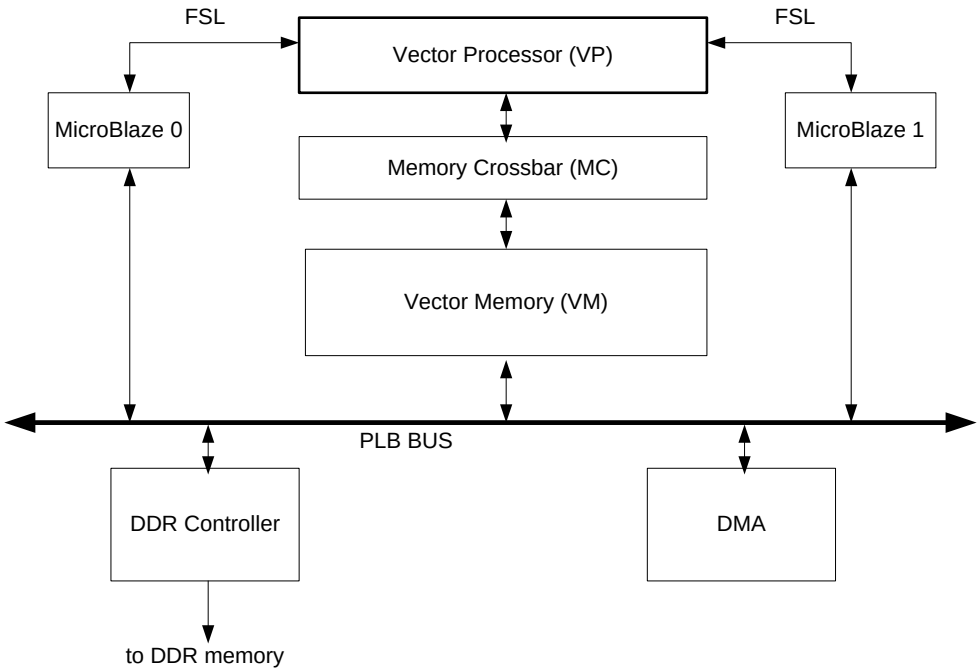


Figure 2.2 Architecture of the FPGA-based VP sharing prototype (PLB: Xilinx Processor Local Bus, used mostly for data transfers via DMA control; FSL: Xilinx Fast Simplex Link).

Figure 2.2 presents the complete system prototype that is implemented on this Virtex-5 FPGA using the Xilinx ISE tools. The Vector Processor (VP), Memory Crossbar (MC), Vector Memory (VM) and Vector Memory Controller (VMC) are custom IPs modeled in VHDL, and the rest of the system is generated using the Xilinx EDK tool, version 12.3. The VP basic structure conforms to the VIRAM lane-based architecture [Kozyrakis and Patterson, 2002; 2003a; and 2003b] that is proposed to connect to a single core. The vector lane space in the design can be partitioned among multiple cores, as needed. This adaptable structure can be used to assign varying numbers of vector lanes to the cores throughout execution based on individual application needs, as per the VLS design choice. Each vector lane contains a subset of the elements from a larger vector register, one FPU and a memory load/store (LDST) unit.

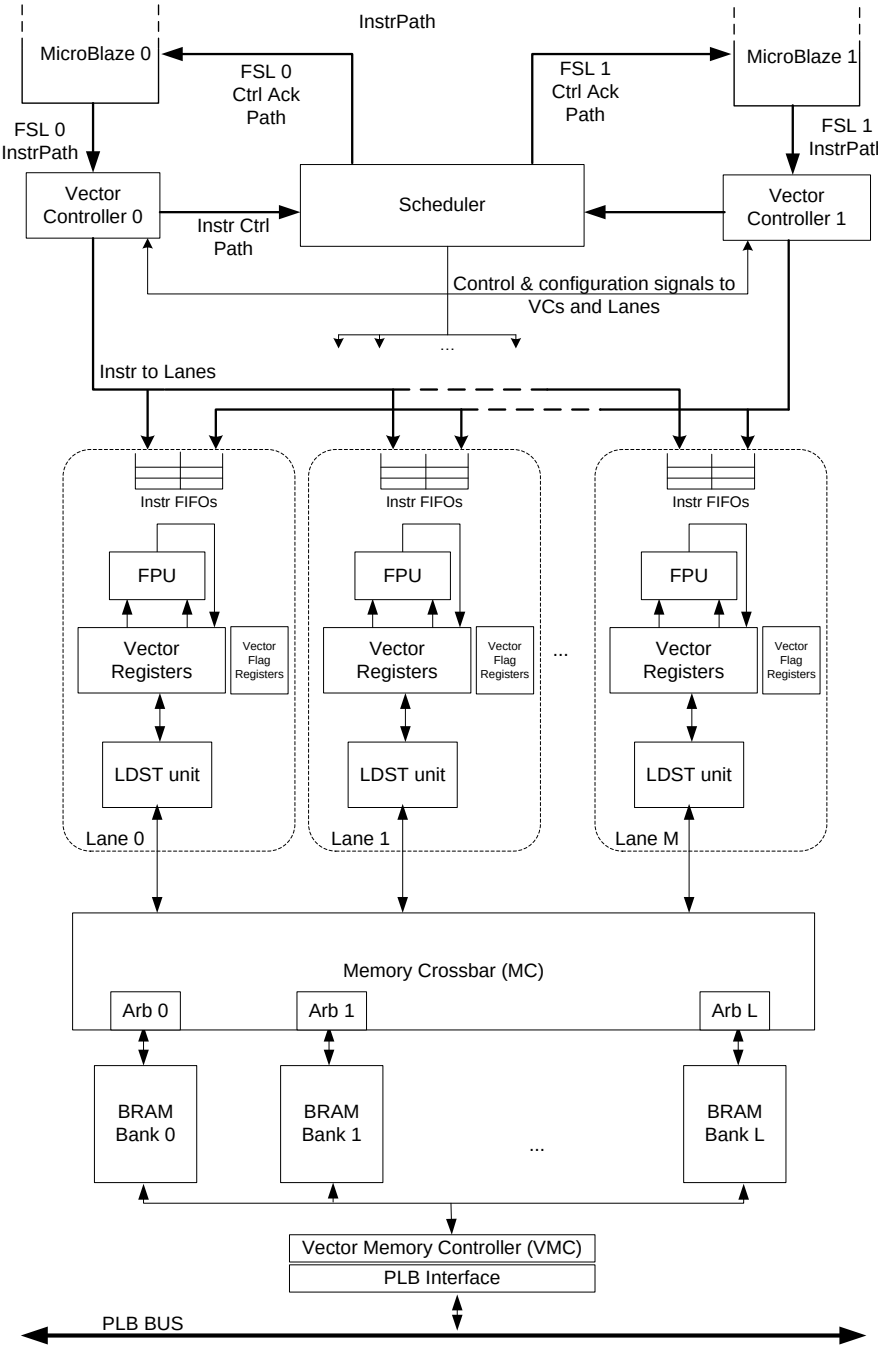


Figure 2.3 M vector lanes shared between two MicroBlaze processors (FSL serves as the instruction path between a MicroBlaze and its associated Vector Controller, through the Scheduler; BRAM: Xilinx Block RAM; each MUX in the figure is part of the respective lane).

Figure 2.3 shows the overall structure for vector lane sharing. Initially the FPGA-based prototype had M=8 lanes and L=8 memory banks (the whole section assumes this configuration). The LDST unit from each lane can operate with or without a vector stride, and can also carry out indexed memory accesses using the crossbar going to the memory. The crossbar allows for concurrent accesses from LDST units to distinct memory banks and also provides round-robin arbitration when many LDST units are accessing the same memory bank.

A distinct Vector Controller (VC) is attached to each scalar processor from which it receives instructions. Such instructions can be of two types:

(i) Vector instructions to move and process data, which are forwarded to vector lanes, and control instructions which are forwarded to the Scheduler.

(ii) Control instructions are used for communications between scalar processors and the Scheduler, for purposes such as acquiring VP resources and the current status of the VP.

The scalar processor always receives an acknowledgement word in response to a control instruction. The VC forms a pipeline with two clock cycles latency, where the first stage is used for decoding, and the second stage is used for hazard detection and register renaming. All three types of data hazard (i.e., RAW, WAR and WAW) are resolved in the latter stage. Also, in this stage the VC requests from the Scheduler access to the instruction bus in order to broadcast the vector instruction to the vector lanes. It is the Scheduler's responsibility to arbitrate between requests coming from both VCs and to acknowledge the one that will get access to the instruction bus. After decoding and hazard detection, the VC broadcasts the vector instruction to its assigned lanes by pushing it with the appropriate vector element ranges into small instruction FIFOs located

in the respective lanes. The Scheduler handles the control instructions coming from the scalar processors. Based on requests from the cores, the Scheduler properly configures the vector lanes. Also, as mentioned previously the Scheduler is responsible for arbitrating on concurrent requests coming from both VCs; control signals for the instruction bus are then asserted based on the arbitration decision.

As Figure 2.4 shows, the lane has a LDST unit (left side) and an FPU (right side). Similar to VIRAM, the LDST unit works with the MC memory crossbar. As mentioned, it can operate with or without a vector stride, and can also carry out indexed memory accesses using the crossbar going to the memory. Additional features are added in this implementation: vector element load/store instructions, where just one element from the vector is loaded or stored, and shuffle instructions to transfer elements between different lanes using a communication pattern stored in any vector register. For shuffle instructions, the LDST unit computes the target lane and destination element, and the data is transferred via the MC crossbar using the data path for standard memory accesses. The MC Arbiter is designed to distinguish between memory and shuffle transfers in order to forward properly the data to the appropriate destination. Table 2.1 presents the LDST instructions supported in the current implementation.

The initiation latency to fill-up the pipeline is 8, 13 and 8 clock cycles for a LDST, add-subtract/multiply and any other ALU instruction, respectively. The ALU in the current design contains 6-stage multiply and add single-precision FPUs. The latency parameters are provided by the Xilinx IP Core Generator and meet the requirements for a 125 MHz design frequency. The rest of the cycles, up to 13, are distributed as follows: the VC pipeline has two stages, one for hazard detection, and one for register renaming,

scheduling and issue to lanes; the lane instruction FIFOs consume one cycle; lane decoding, operand fetching and issue to execution units takes two clock cycles mainly caused by the latency of BRAMs; the result buffers involve one cycle; finally, one clock cycle is taken for the lane to inform the hazard detection mechanism in the VC about instruction completion. Without loss of generality, the FPU can execute single-precision floating-point addition, subtraction and multiplication, and can also evaluate the absolute and negate operations.

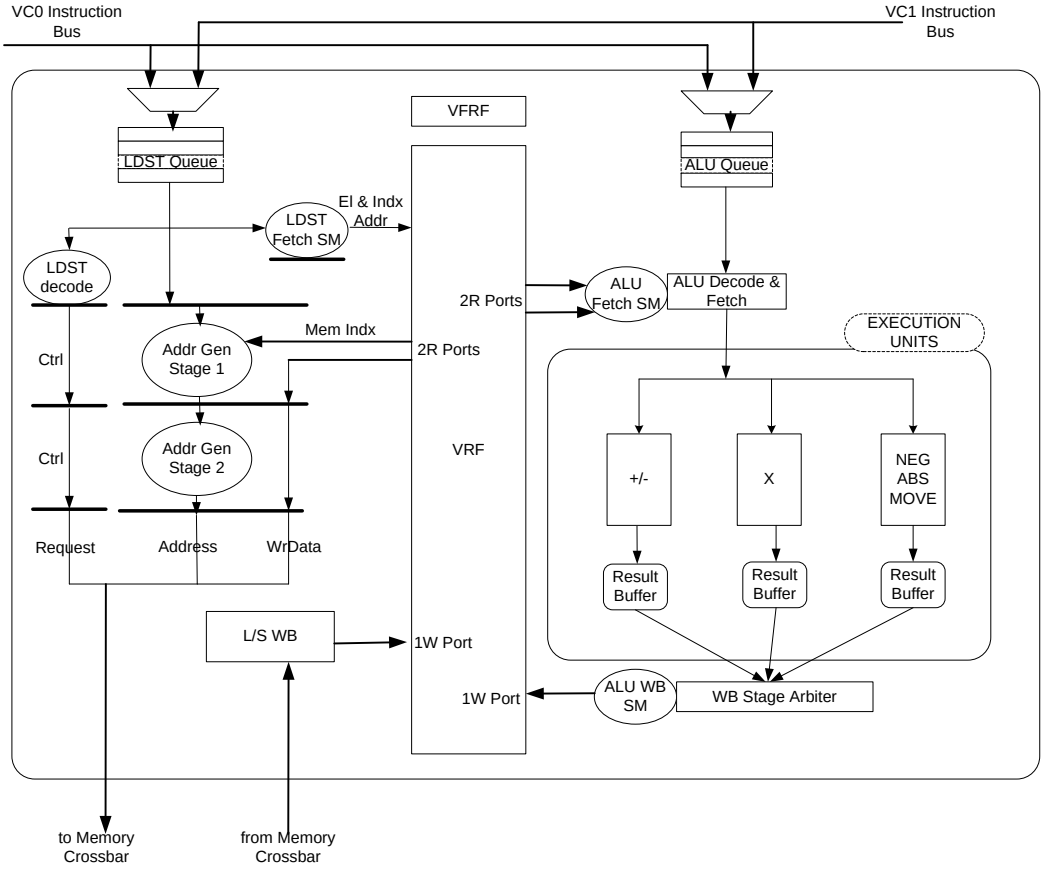


Figure 2.4 Vector lane architecture.

As shown in Figure 2.4, the LDST and ALU instructions involve separate paths. Therefore, it is possible to have concurrent execution of LDST and ALU instructions as long as there is no data dependence between them. LDST instructions are always

executed and committed in order. ALU instructions are issued in order but might commit out of order due to different pipeline depths in the execution units. However, this does not violate data dependencies since instructions that execute at the same time in a lane have no data dependence.

Table 2.1. Load/Store (LDST) Instructions Summary

	Details	**Initiation Latency (cycles)**
VLD **VST**	Unit stride Vector Load and Store instructions	8
VLDS **VSTS**	Stride Vector Load and Store instructions. Stride could take values up to 1024.	8
VLDX **VSTX**	Indexed Vector Load and Store instructions.	8
VELLD **VELST**	Element Load and Store instructions.	8
VSHFL	Vector Shuffle instructions. The instruction takes 3 parameters: destination vector register, source vector register and vector register containing permutation information.	8

Table 2.2 ALU Instructions Summary

Instruction	**Details**	**Initiation Latency (cycles)**
VMUL **VADD** **VSUB**	Vector-Vector Multiplication. Vector-Vector Addition. Vector-Vector Subtraction.	13
VMULS **VADDS** **VSUBS**	Vector-Scalar / Scalar-Vector Multiplication. Vector-Scalar / Scalar-Vector Addition. Vector-Scalar / Scalar-Vector Subtraction.	13
VMOV **VNEG** **VABS**	Vector move instruction. Vector negate instruction. Vector absolute instruction.	8
VFLD **VFMOV** **VFNEG**	Load Vector Flag from Scalar instruction. Vector Flag move instruction (from scalar to Vector). Vector Flag complement instruction.	8

Since it is possible to have two different functional units writing back the results to VRF in the same clock cycle, the ALU write port contains a write back arbiter which arbitrates between multiple requests from different functional units result buffer. Table 2.2 summarizes the ALU instructions supported in current implementation. Each lane assigned to a VC informs it upon instruction completion and the entire SIMD instruction

is considered completed by the VC when all the lanes have completed this step. Each SIMD instruction is labeled by the VC with a unique tag, and dedicated signaling from the lane to its assigned VC informs the latter about the completion of the instruction.

The elements corresponding to one vector register are distributed across multiple lanes in low-order interleaved fashion (also called folding), and the number of elements from a vector register corresponding to one lane is configurable. Each instruction consumes a start-up latency plus a number of cycles equal to the number of elements stored in the lane's vector register minus one. An instruction without dependencies consumes in the LDST or ALU unit a number of pipeline cycles equal to the size of the vector register used in the lane. Each lane contains a multi-ported Vector Register File (VRF) with 512 32-bit locations efficiently implemented with Xilinx FPGA 36Kbit BRAMs (Block RAMs). Each of the LDST and ALU units requires two reads and one write per clock cycle. Therefore, the memory has two write and four read ports (2W/4R), and is implemented by doing replication (2×) and multi-pumping with a double frequency [LaForest and Steffan, 2010]. In order to simultaneously support all three sharing contexts in the same architecture, each lane contains four configuration registers which are updated at runtime by the Scheduler. These are:

(i) The first register contains the VC ID to which the lane is assigned. This is used by the lane to inform the appropriate VC on instruction completion.

(ii) The second register contains the number of lanes assigned to the particular VC to which this lane is assigned. This register is updated when switching between the CTS/FTS and VLS operating contexts, and is necessary in order to compute the correct address for memory transfers and shuffle operations.

(iii) The third register contains the fixed lane ID (or lane index).

(iv) The fourth register contains the number of elements from a vector register which are located in the same lane.

Since the VRF memory within each lane has fixed size, as per Equation 2.1, increasing the number of elements in a vector register will automatically decrease the number of available registers. Table 2.3 presents some valid combinations of the vector length and the number of available vector registers. It is a software decision to tune the vector length and the available number of registers in order to optimize the execution time and/or power consumption for a specific task. Besides the VRF memory in each lane, there is a Flag Vector Register File (FVRF) memory which contains 512 1-bit elements. Each bit is used as a mask for conditional execution of vector instructions on the corresponding element in the VRF.

Table 2.3 Examples of Vector Length and Number of Registers

Configuration	Elements per register (1 lane)	Vector Length	Number of available registers
8 lanes	4	32	32
8 lanes	8	64	32
8 lanes	16	128	16
8 lanes	32	256	8
4 lanes	4	16	32
4 lanes	8	32	32
4 lanes	16	64	16
4 lanes	32	128	8

The Vector Memory (VM) contains eight low-order interleaved Xilinx BRAM banks for a total capacity of 64 Kbytes (8 banks x 8 Kbytes per bank). Without crossbar conflicts in accessing the VM banks, eight 32-bit data transfers can be performed on each clock cycle using the eight LDST units, giving a peak bandwidth of 32Gbs with a design frequency of 125 MHz. Of course, this bandwidth will double with an expanded design for double-precision floating-point operations and respective data transfers. Each BRAM

is a true dual-port memory; one port is used for data transfers between the VM and the VP's register file, and the second port is used for data transfers between I/O controllers and the VM through the PLB interface. Therefore, this architecture supports concurrency and yields high bandwidth for data transfers involving the VM.

2.2.1 VP Scheduler

The Scheduler controls the working context for the entire VP. Based on the chosen working state, the Scheduler provides configuration signals to all lanes and VCs. The signals for a particular lane provide information about: a) which VC the vector lane is assigned to, being VC 0, VC 1 or both; b) the total number of lanes assigned to the VC, including this particular lane; c) the offset/index of the lane in the lane array assigned to that VC; and d) the number of elements from a vector register which are located in this lane. The information from the first configuration signal (i.e., configuration a) is used by the lane to notify the appropriate VC of instruction completion, and the information derived from configurations b), c) and d) is used by the lane's LDST unit to properly translate addresses for memory accesses and shuffle operations. The configuration signals provided to the VC by the Scheduler configure the former to work either in the exclusive context (i.e., one thread arriving from one scalar processor) or in the lane-sharing context (i.e., two distinct threads arriving from the two scalar processors).

Figure 2.5 shows some of the possible states for the Scheduler; each cell in the figure contains the state of the corresponding lane: which VC it is assigned to, the total number of lanes assigned to that VC, the lane index, and the number of elements from a vector register in that lane. STATE1 is similar to CTS in which all eight lanes are assigned only to MB0 through VC0 and the Vector Length is 4×8=32. In STATE2, both

scalar processors have access to all eight lanes using FTS sharing; the value of VL is 8×8=64 with the application running on both MicroBlazes. In STATE3, each VC has *M*=4 lanes assigned to it; the VL value for the application running on MB0 and MB1 is 16 and 32, respectively. Finally, STATE4 has four idle lanes and four lanes assigned to VC1, and the VL value is 64.

Each MicroBlaze can use a set of four indivisible instructions to communicate with the Scheduler. These are VP_REQ, VP_REL, VP_GETSTAT1 and VP_GETSTAT2; in response, the Scheduler always replies with a message. For a core to get access to the entire VP or to a subset of its lanes, the VP_REQ instruction is used. This instruction contains two parameters: i) *vl_size*, which indicates the required vector length (i.e., the number of vector elements in a vector register). *vl_size* can take the values: 4, 8, 16, 32, 64, 128 and 256; and ii) *perf_req*, a three-bit field which indicates the performance requirements, thus distinguishing among eight priority levels. However, without loss of generality, in current implementation this field assumes two active values: *perf_req*=3'b000 corresponds to a low priority/performance application; and *perf_req*=3'b111 represents high priority. Based on the current VP state, any other pending VP requests, and the details of the current request, the Scheduler decides to grant a scalar processor request or not, and informs the requesting processor accordingly. In the extreme case where VP_REQ instructions arrive from both scalar processors in the same clock cycle, the Scheduler will reply to both of them but will positively acknowledge only one. For example, Figure 2.6 shows the reply word in response to a VP_REQ instruction. For a successful request, the Scheduler will reply with the acquired VL value

and the acquired performance fields. In the case of an unsuccessful request, the Scheduler will transmit the available VL value and the currently available highest priority.

Lane	L0	L1	L2	L3	L4	L5	L6	L7
STATE1	VC0 8 0 4	VC0 8 1 4	VC0 8 2 4	VC0 8 3 4	VC0 8 4 4	VC0 8 5 4	VC0 8 6 4	VC0 8 7 4
STATE2	VC0/1 8 0 8	VC0/1 8 1 8	VC0/1 8 2 8	VC0/1 8 3 8	VC0/1 8 4 8	VC0/1 8 5 8	VC0/1 8 6 8	VC0/1 8 7 8
STATE3	VC0 4 0 4	VC0 4 1 4	VC0 4 2 4	VC0 4 3 4	VC1 4 0 8	VC1 4 1 8	VC1 4 2 8	VC1 4 3 8
STATE4	Idle	Idle	Idle	Idle	VC1 4 0 16	VC1 4 1 16	VC1 4 2 16	VC1 4 3 16

Figure 2.5 State Examples for the Scheduler (each cell in the figure contains the state of the corresponding lane: which VC it is assigned to, the total number of lanes assigned to that VC, and the lane index).

31 28	27 17	16 14	13 1	0 0
OP_CODE	Acquired VL/ Maximum Avail VL	Acq PERF / Avail Perf	RSVD	SUCC

Figure 2.6 Scheduler to MicroBlaze reply word in response to a VP_REQ.

In response to the VP_GETSTAT1 instruction, the Scheduler will reply with the following information: status of VC1 and VC0 (idle or busy), number of lanes assigned to VC1 and VC0, and performance status of VC1 and VC0. In response to the VP_GETSTAT2 instruction, the Scheduler will reply with: status of VC1 and VC0 (idle or busy) and Vector Length assigned to VC1 and VC0. The VP_REL instruction is used to free all the VP resources previously acquired by a scalar processor. Table IV summarizes the control instructions.

Table 2.4 VP Control Instructions Summary

Ctrl. Instruction	Details
VP_REQ	Request for resource allocation.
VP_REQ	Request for release of allocated resources.
VP_GETSTAT1	Request for VL status.
VP_GETSTAT2	Request for Lanes, Performance and Power status reply.

In current VP prototype, three types of software-based adaptation are facilitated to take advantage at runtime of any available VP resources: (a) at the core-run software level, where the core changes at runtime the routine that implements a needed vector kernel based on the available VP resources (the routines may be parameterized by vector length or performance level); (b) closer to the VP level, the Scheduler is able to appropriately configure the working context of the VP based on its current state and the current set of requests coming from the scalar cores; and (c) at the lane level, where the Scheduler can configure some of the vector lane parameters (e.g., the number of elements per vector register contained in a vector lane).

Based on its current state and the request parameters, the Scheduler decides if any resources are available and replies with a successful or unsuccessful acknowledge message. Based on this information and the application routines that it has to run, the scalar processor makes the final decision on the number of lanes to acquire. To avoid the duplication of stored code, generic parameterized routines may be developed (e.g., routines with such parameters as the vector length, number of registers to be used, etc.).

Figure 2.7 shows the current algorithm run by the Scheduler, and Table 2.5 presents some examples of Scheduler state transition based on a request coming from one of the scalar processors. Under CTS each vector kernel in a thread runs to completion before releasing all the VP resources. In VLS context the scheduler gives equal priority to

competing threads by assigning the same number of exclusive VP lanes to each thread. In FTS the scheduler can accommodate simultaneously multiple threads in any given cycle as long as they need different VP resources; when competing for the same ALU or LDST unit, the scheduler applies round-robin arbitration per unit. Since the main objective here is to demonstrate the viability of VP sharing among cores and threads, for the sake of brevity the development of very sophisticated scheduling schemes will become a future research objective. Also, as experimental results dictate in Chapters 4 and 5, the specific configuration to be chosen could be driven by power/energy and performance tradeoffs.

```
if 8 lanes IDLE {
      if req_perf=low {
            assign 4 lanes to VC;
            VL=requested_VL;
            REPLY=SUCC;
      }
      if req_perf=high {
            assign 8 lanes to VC;
            VL=requested_VL;
            REPLY=SUCC;
      }
}
if 4 lanes IDLE {
            assign 4 lanes to VC;
            VL=requested_VL;
            REPLY=SUCC;
}
if all 8 lanes BUSY {
      if requested_VL = current_VL {
            assign 8 lanes to VC;
           VL=requested_VL;
            REPLY=SUCC;
      } else {
            REPLY=UNSUCC;
      }
}
```

Figure 2.7 Scheduler algorithm.

In current implementation, under CTS only one VC can issue an instruction to vector lanes at any time. In the FTS context, both VCs can issue simultaneously instructions to the lanes. The lane execution pipeline is capable of processing

simultaneously instructions issued by both scalar processors since multiple vector instructions can simultaneously reside in the pipeline. Under these circumstances, FTS requires vector register renaming because the scalar processors must be assigned distinct vector registers. Usually small- and medium-scale SIMD machines are currently used as stream processors. Data can be streamed into the VM of VP-based structure using the DMA capability; the program then operates on this data using the VM as a data workspace, and the results are streamed back to the main memory using again DMA control. This data streaming can occur simultaneously with arithmetic computations.

Table 2.5 Examples of Transition for Scheduler States

Scheduler state	**Request parameters**	**Reply**	**Scheduler next state**
all 8 lanes IDLE	MB0 req req_vl=64 req_perf=high	SUCC VL=64 perf=high	8 lanes assigned to VC0 VL=64 els per lane=8
all 8 lanes IDLE	MB0 req req_VL=128 req_perf=low	SUCC VL=128 perf=low	8 lanes assigned to VC0 VL=128 els per lane=32
all 8 lanes assigned to VC0 VL=64	MB1 req req_VL=64 req_perf=high	SUCC VL=64 perf=high	8 lanes assigned to VC0/1 VL=64
	MB1 req req_VL=128 req_perf=high	UNSUCC	8 lanes assigned to VC0 VL=64
4 lanes assigned to VC0 VL=64	MB1 req req_VL=128 req_perf=high	SUCC VL=128 perf=low	4 lanes assigned to VC0 4 lanes assigned to VC1

Figure 2.8 shows how the main routine of a MicroBlaze is developed for CTS/FTS and VLS sharing, and Figure 2.9 shows steps 2.1 to 2.3 for CTS sharing. Just before a thread becomes active, the software may clear all vector registers using a VP clear instruction. Another possibility is to implement additional hardware to support a local reset controlled by the Scheduler and triggered when the VP space is exclusively

acquired by one of the scalar processors. When the scalar processor finishes the vector routine, it releases the coprocessor by issuing a VP_REL instruction.

```
STEP 1.1 Lock DMA resource
STEP 1.2 Transfer data from DDR to Vector Memory (VM)
STEP 1.3 Unlock DMA resource

STEP 2.1 Acquire VP resources
STEP 2.2 Call VP routine to process data from VM
STEP 2.3 Release VP resources

STEP 3.1 Lock DMA resource
STEP 3.2 Transfer processed data from VM to DDR
STEP 3.3 Unlock DMA resource
```

Figure 2.8 Main MicroBlaze routine for CTS, FTS and VLS sharing.

Prior to this instruction the MicroBlaze code makes sure that no vector register is dirty; also, the state of the vector processor for the respective MicroBlaze program is saved back into the memory. Therefore, the state of the VP must be saved before the VP is released in a shared environment.

Under FTS, vector instructions received from both scalar processors share the VP resources. This context resembles fine-grain multithreading in superscalar processors, and increased throughput is expected because there are no data dependencies between instructions coming from different processors.

Under VLS, if the *req_perf* value is low (*req_perf=3'b000*), the Scheduler splits the VP into two distinct lower-sized VPs with each one having its own vector length. For example, if MB0 requests a VL=32 with *req_perf=low* and MB1 requests a VL=64 with *req_perf=low*, the final state of the Scheduler will be: four lanes assigned to VC0 with 8 elements per lane from the same vector register and four lanes assigned to VC1 with 16

elements per lane from the same vector register. Then, the VP will serve simultaneously two threads of different vector lengths.

```
STEP 2.1
while (ack != IDLE) {        //wait until the VP is idle
   VP_REQ ack;               // Scheduler returns a positive or negative reply;
}
STEP 2.2
  VLD VR0, A;                // Processor starts using the VP; loads
                             //the vector register (A is address in Vector Memory)
  …
  VST VR4, B;                // Processor finishes the routine;
                             // saves the vector result
                             // (B is address in Vector Memory)
STEP 2.3
VP_REL ack;                  // Unlock the VP resources and receives
                             // a reply if successful or not;
```

Figure 2.9 CTS vector sharing MicroBlaze routine.

2.2.2 Additional Architectural Features

During the architecture development new architectural features were added. This section summarizes the updates added to the already presented baseline architecture.

Different vector lengths per CPU. In FTS context sharing each CPU can request for any vector length that is a power of two. This requires duplicating the configuration register that keeps the number of elements per lane and adding to each instruction a field (bit) indicating the VC number.

AnyVectorLength support allows any scalar core to require a vector length that is between 0 and VL-1 (called, from now on, *a*VL). A new control instruction is added, that is VP_ANY_REQ. This feature has several advantages:

(i) Avoids strip-mining of loops with known number of iterations (`for` loops) since the *a*VL value could match exactly the number of loop iterations.

(ii) It could fit a natural vector length that is not a power of two. For example, in the Gaussian elimination algorithm the number of nonzero elements in the rows that needs to be processed decreases gradually from the width of the matrix down to one.

The Scheduler is responsible for reconfiguring appropriately in each lane the register that keeps the number of elements per lane. Also, for each lane configuration space, a mask bit per VC is required to disable the last operation of any instruction in each lane for vector lengths which are not multiple of the total number of lanes. For example, Figure 2.10 shows the state of each lane after a VP_REQ instruction and after a VP_ANY_REQ instruction in a VP having eight lanes. Any instruction prior to VP_ANY_REQ will be executed with the old vector length and the instruction following it will have the new vector length. *AnyVectorLength* does not change the hazard detection mechanism; still, the detection of all hazards is done on vector registers with VL vector length.

Quality of Service (QoS) support. The goal of scheduling is to provide the desired utilization to each thread that issues VP instructions. Managing VP instruction streams (VP threads) with different priorities is a daunting challenge. The main reason is that scheduling instructions coming from different threads may require different vector lengths and different throughputs. Scheduling at the instruction level may result in an unbalanced utilization. In the baseline FTS sharing context, the round robin policy at the instruction level implemented in the Scheduler is used to control the lanes. This scheme works quite well and provides fair utilizations to the threads when the vector lengths of instructions issued by both VCs are equal; but not so well in other cases.

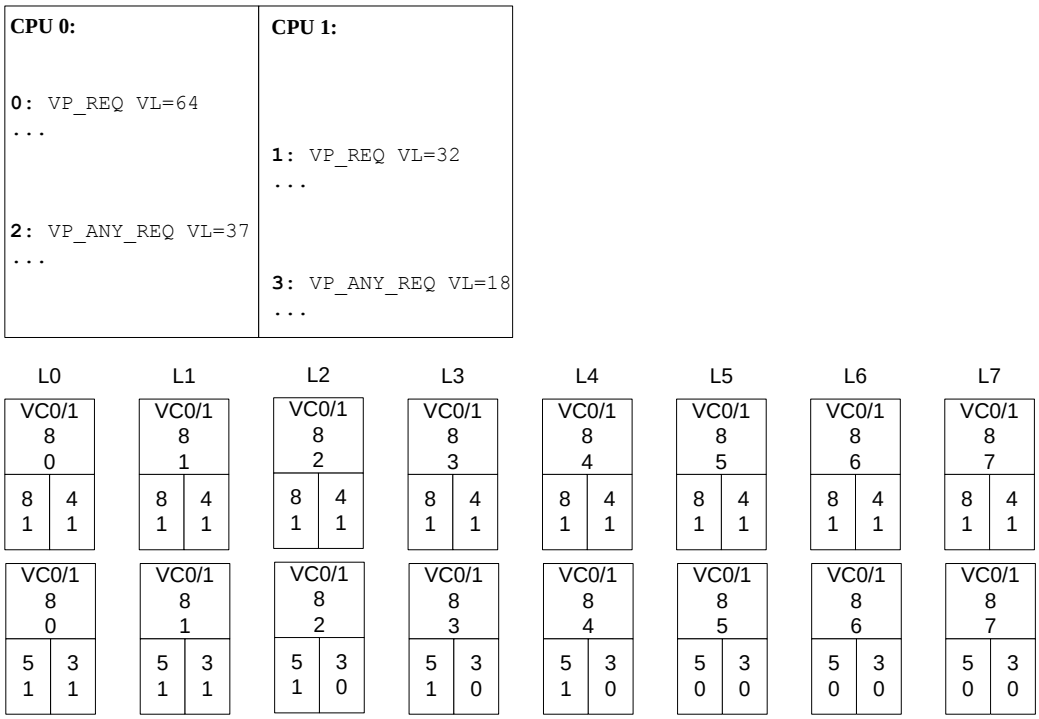


Figure 2.10 The configuration state of each lane after instructions 0 and 1 are executed (top row) and after instructions 2 and 3 are executed (bottom row). Each lane configuration state contains (in each cell from top to bottom): VC ID(s) indicating from which VC the lane receives instructions; number of total lanes forming the VP; the lane index; per VC (VC0 or VC1) number of elements from each vector register in the lane; per VC mask bit required to mask the last operation of any instruction in each lane for vector lengths which are not multiple of number of lanes.

Also, since the ALU and LDST instructions coming from both VCs are stored in the same lane's circular FIFO, one instruction issued by one CPU may slow down or block the execution of an instruction coming from the other CPU independent of the arbitration policy. Therefore, the following modifications are done in order to support cycle-based arbitration logic at the lane level:

(i) Per VC ALU and LDST instruction FIFOs in each lane. In order to reduce the impact of duplicating the hardware resources allocated to lane instruction FIFOs the number of FIFO locations is reduced by half. In the baseline implementation, each ALU or LDST FIFO stores a maximum of 8 instructions;

in the updated architecture, each VC ALU or LDST FIFO can store up to 4 instructions. In all simulations this modification has no impact on performance since the total maximum number of ALU or LDST instructions that can reside in a lane is still 4+4=8. This modification resembles the hardware support for virtual channels introduced in [Dally, 1992].

(ii) A round robin or strict priority arbitration logic at the lane level. In the ALU and LDST units a simple arbitration logic is added to arbitrate in each cycle which instruction element gets executed. This solution introduces flexibility to control individual thread performance (i.e., it satisfies thread quality of service at the expense of adding more complex arbitration logic).

Figure 2.11 shows additions to the lane. The Round Robin (RR) arbiter can be configured by Scheduler (as per software request) to work in the strict priority mode (SP); that is, always the arbiter will always choose the high priority instruction element to be executed; else, if no high priority instruction exists in the instruction FIFO, the low priority instruction element will get access to the lane execution pipeline. Additional flexibility could be supported by adding a weighted round robin logic. However, this will add additional logic delay to the instruction path for an FPGA implementation, but might be a good design choice in an ASIC design. Therefore, each instruction stream will have a separate path to the execution stages in LDST and ALU units. In the FTS context and the SP mode, a high priority thread could potentially have the same performance as if it running in the single thread configuration (CTS mode) assuming there is no contention on the Memory Crossbar on IO instructions. This case appears if in each lane a LDST instruction occupies a number of pipeline slots equal to the number of elements in the vector register corresponding to that lane. Contention may occur in programs with strided and indexed load/stores and shuffle operations.

2.3 Resource Consumption and Resource Scalability

The VHDL design is also synthesized using Xilinx ISE 12.3 synthesis flow for the Xilinx Virtex-6 XC6VLX130T FPGA device. The Virtex-6 FPGA is built using a 40 nm state-of-the-art copper process technology, and contains a column-based architecture comprising logic slices, 36-Kbit block RAMs (BRAMs - RAMB36_EXP), DSP slices (DSP48E) and many I/O hardwired IPs [Xilinx, 2011]. Each logic slice can implement functions using four 6-input look up tables (LUTs) and four flip-flops; the LUTs can also be configured to realize dual-output 5-input LUTs. A LUT is a 64-bit memory capable of realizing any of 32 or 64 functions. The DSP48E slice is based on a 25x18 bit multiplier and a 48-bit adder/subtractor/accumulator. As a note, the VLX130T FPGA fabric is equivalent with approximately one million ASIC gates.

Table 2.6 shows resource consumption figures for the VP and VM in the Virtex XC6VLX130T FPGA device. Note that a vector lane contains a LDST unit, an ALU unit, a VRF and a FVRF; the VP contains eight lanes, two VCs and one Scheduler. Except for the last row in the table, the percentage values are shown relative to the total design resource consumption. As expected, most of the design is occupied by ALU units. Each lane consumes 1066 LUTs and 3642 registers (i.e., 12.4% and 11.3%, respectively, of the entire design), and the device consumption collectively by the VC and Scheduler is less than 4%. The overall device consumption by the VP and VM is 8833 LUTs and 32106 registers, which represent 11.1% and 20%, respectively, of the VLX130T resources. The rest of the FPGA resources can be used for the realization of scalar processors, buses, DMAs, I/O controllers and other IPs.

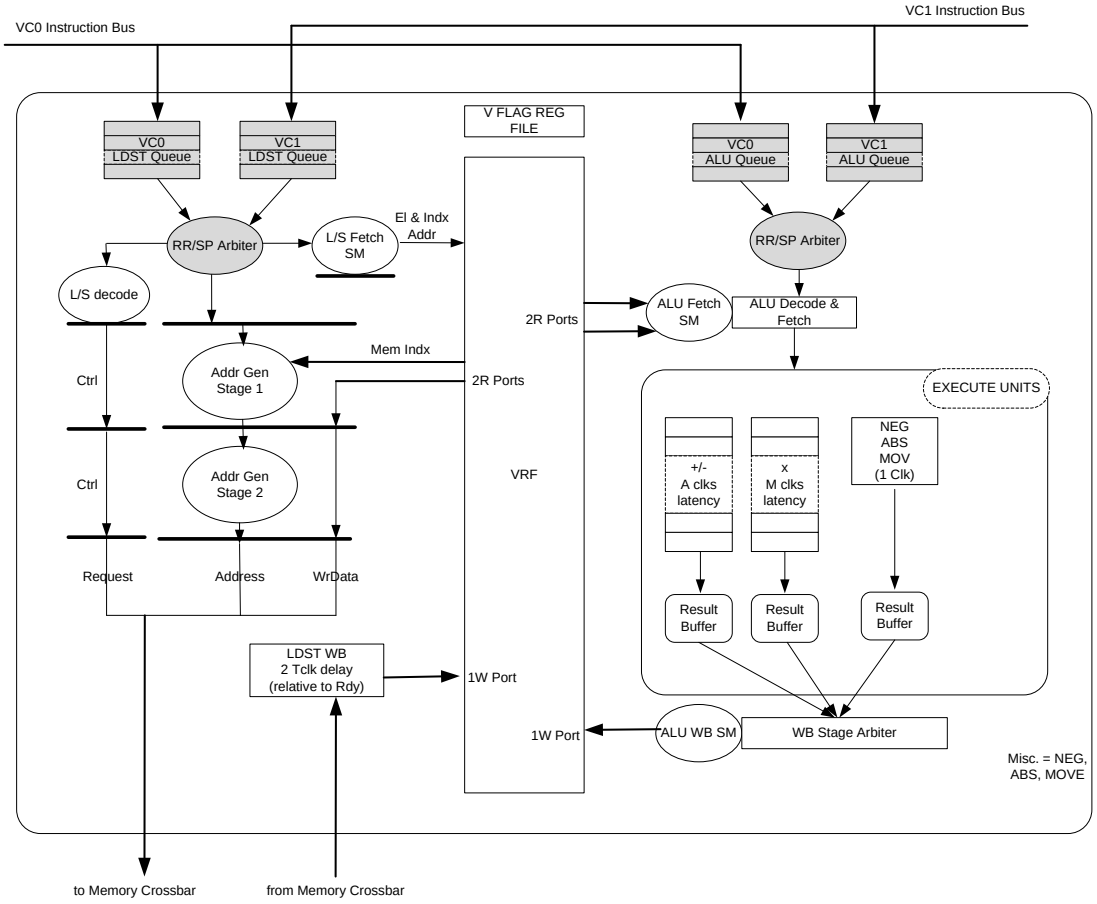
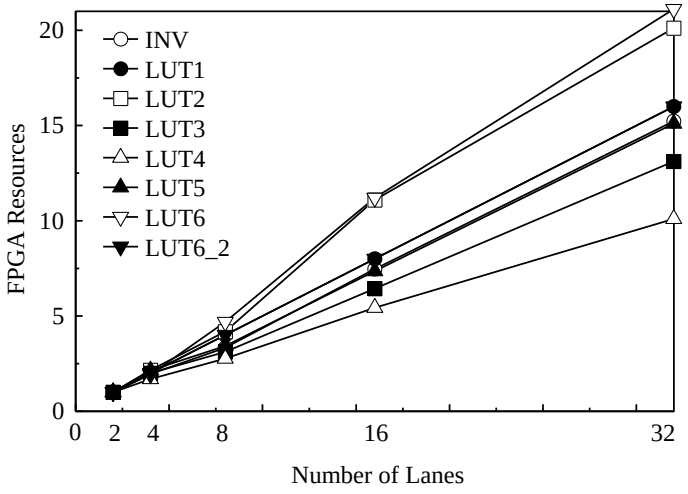


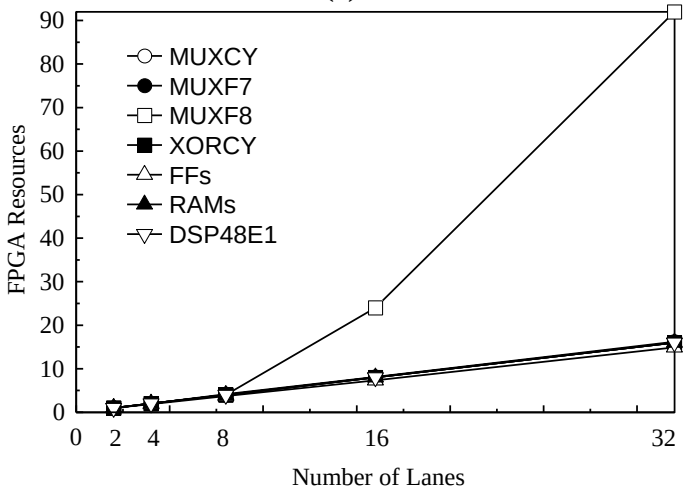
Figure 2.11 Vector Lane architecture to support QoS and two VP instructions per cycle. The modifications from the baseline architecture are colored in gray.

Table 2.6 Resource Consumption in the Virtex-6 XC6VLX130T FPGA Device for a Configuration of Eight Lanes and Eight Memory Banks

			FFs	LUTs	BRAMs	DSP48E1
VP (8LANES)			30310 (94%)	8518 (96%)	8	24
	LANE		3642 (11.3%)	1066 (12.4%)	1	3
		LDST unit	1156 (3.6%)	114 (1.3%)	-	1
		ALU unit	2343 (7.3%)	873 (10.1%)	-	2
		VRF	107 (<1%)	3 (<1%)	1	-
		FVRF	27 (<1%)	16 (<1%)	-	-
		CFG	12 (0.04%)	2 (~0%)	-	-
	VC		489 (1.5%)	71 (<1%)	-	-
	Scheduler		277 (0.8%)	80 (1%)	-	-
VM (8 MEM BANKS)			1796 (7.7%)	315 (3.8%)	16	-
VP+VM (% out of XC6VLX)			32106 (20%)	8833 (11.1%)	24(9%)	24(5%)
Total XC6VLX130T			160,000	80,000	264	488



(a)



(b)

Figure 2.12 Resource scaling for a vector processor with a number M of lanes equal with 2, 4, 8, 16 and 32 on XC6VLX130T FPGA device. Number of memory bank equals the number of lanes and the crossbar has the size M×M. All the numbers are normalized to the 2 lanes configuration numbers.

Figures 2.12 (a) and (b) show the usage of FPGA primitives for a vector processor with a number of lanes and memory banks between 2 and 32. As observed, resources scale linearly with the number of lanes except for the MUXF8 primitive from Figure 2.12 (b). This component is inferred by the Memory Crossbar and, as expected, scales quadratically with the number of lanes (especially for the 16×16 or 32×32 crossbar). Therefore, it is natural to assume that all resources scale linearly with the number of lanes

except for those inferred by the crossbar. The same conclusion is expected to hold for an ASIC implementation. Also, even if they have low contribution to the total budget, some components of the design have fixed resources for any number of lanes: VCs, the Scheduler, and the interface between VM and the main bus (PLB).

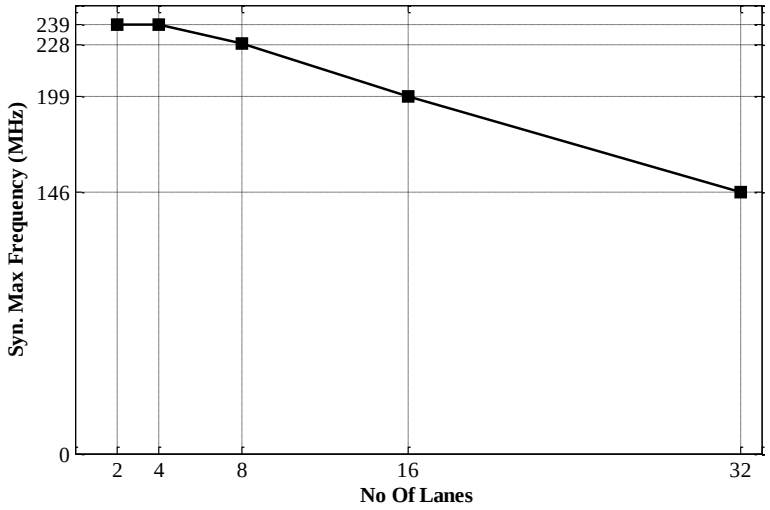


Figure 2.13 Maximum Frequency after synthesis for a Vector Processor with 2, 4, 8, 16 and 32 number of lanes on XC6VLX130T FPGA device. Number of memory bank equals the number of lanes and the fully connected crossbar has size M×M.

Figure 2.13 displays the maximum frequency after synthesis for a VP design configured to have 2, 4, 8, 16 or 32 lanes. For the 2 and 4 lane configurations, the critical path lies in the vector lane logic; more explicitly, it involves the vector register file because this component runs at double the speed, i.e., 250 MHz. Starting with the 8-lane configuration, the crossbar becomes the timing bottleneck. For more than 16 lanes, other solutions for access to memory banks can be implemented in order to keep the working frequency high. The lane access to the memory banks follows the Uniform Memory Access (UMA) memory model and there are two solutions to scale the design:

(i) The first design option is to implement a non-blocking multistage switch; this will increase the working frequency but will affect the latency of accesses. However, since the VP is a throughput oriented machine, the performance impact is expected to be minimal.

(ii) The second design option is to change the memory model to a Non-uniform Memory Access (NUMA); i.e., different access latencies for different memory banks. This option stems from the fact that the memory accesses in most of the current applications are frequently unit-strided. If the vector memory addresses for simultaneous memory accesses are distinct for the M lanes in a M×M configuration of the VP, the lane with index *n* will access in this unit-stride mode only the memory bank with index *n*. In these cases, lane accesses in the unit-stride mode will have minimal latency; however, non-strided and indexed accesses will potentially have increased latencies. Application Software should be aware of these particular architectural features and should favor memory accesses aligned to M element boundaries.

CHAPTER 3

APPLICATIONS

In order to evaluate VP design and prove the usefulness of the sharing schemes, a set of applications must be developed. Ideally, the processor should be evaluated using full-size, end-user applications running within the environment of a complete product. However, such an evaluation is rarely possible since it requires a full software-hardware co-design. Instead, processor designers evaluate and compare processors using a benchmark suite; i.e., a short collection of applications of interest. This chapter describes the software development process in Section 3.1 and some of the key benchmarks in Section 3.2.

3.1 Software Implementation

Software implementation requires handwritten or inline assembly code, translating vector instructions with a modifed GNU assembler (gasm). Researchers have investigated the auto-vectorization capability of gcc, but have not yet used it successfully [Yiannacouras, 2009]. Instead of an auto-vectorization compiler, the SW implementation uses C macros exclusively to emit Microblaze custom instructions on demand without modifying gcc. The custom instructions are Microblaze instructions that are communicated using Fast Simplex Link (FSL) channels.

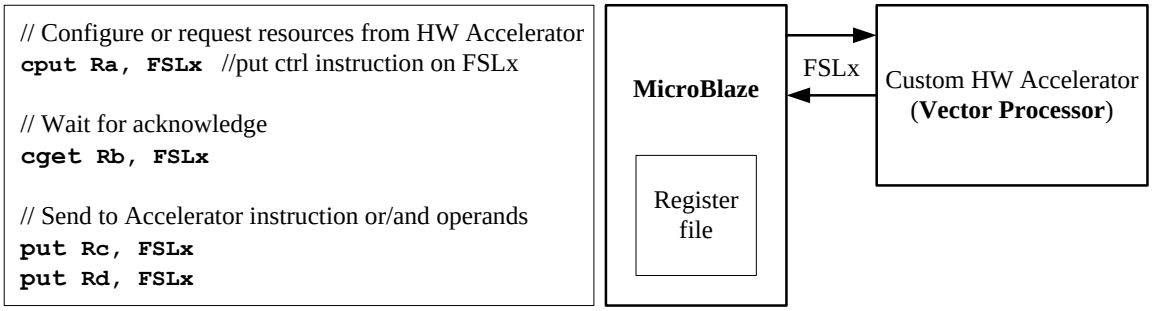


Figure 3.1 FSL used with the Vector Processor.

The FSL channels are dedicated uni-directional point-to-point 32-bit data streaming interfaces. In Figure 3.1 the `put` instruction from the MicroBlaze ISA is used to transfer information from a general-purpose register to an FSL port. The `get` instruction is used to transfer data in the opposite direction. Both instructions come in four flavors: blocking data, non-blocking data, blocking control, and non-blocking control. The FSL control instructions `cput` and `cget` are used by MicroBlaze to communicate with the control part of the VP, i.e., the Scheduler. This method is similar to extending the ISA with custom instructions, but has the benefit of not making the overall speed of the processor pipeline dependent on the custom function. Also, there are no additional requirements on the software tool chain associated with this type of functional extension. The macros are more readable, and the system is much simpler to program because the user does not need to track the scalar values as register numbers. Instead, the user tracks only the memory addresses and vector register numbers needed to form the VP instructions.

Some convenience routines are implemented to simplify the programming of the VP. These routines are implementing the kernels needed to benchmark the VP and also useful DMA data transfers. The VP architecture comprises a memory model (Vector Memory) that is not cacheable. The same memory model paradigm is used in the Cell

processor [Chen et al., 2007] with a Local Memory in each processing element and in most of the embedded systems that make use of such called SPM [Marongiu et al., 2011]. This simple design has several practical advantages and is particularly profitable in the embedded domain. SPM requires up to 40% less energy and 34% less area than cache [Banakar et al., 2002], and provides better performance than cached systems for applications with regular memory accesses. Unlike caches, it is the programmer's responsibility in VP system (possibly with the help of the compiler) to explicitly manage data transfers between the main memory and the Vector Memory using DMA transfers.

Figure 3.2 (a) shows the declaration of two functions used to transfer data between VM and the main memory. `DMA_Transfer_Blocking()` stalls the execution of the CPU until the entire transfer is completed, and `DMA_Transfer_NonBlocking()` initiates the DMA transfer and resumes execution in parallel with the data transfer. Except in cases where synchronization between data transfers and the VP instruction streams is required, non-blocking version is used in order to overlap DMA transfers with VP execution. Figure 3.2 (b) presents the implementation of a Finite Impulse Response (FIR) function where the FIR size and the vector length VL are input parameters. The development of libraries where the vector length is passed as a parameter introduces flexibility and portability to the programmer.

3.2 Benchmarks

The software routines were implemented using the Xilinx Platform Studio (XPS) and Xilinx Software Development Kit (SDK) [Xilinx SDK, 2011], and compiled with MicroBlaze gcc (mb-gcc).

```
void DMA_Transfer_NonBlocking(void *Src_Addr, void *Dst_Addr, u32 ByteCount);

void DMA_Transfer_Blocking   (void *Src_Addr, void *Dst_Addr, u32 ByteCount);
```

(a)

```
void fir_outprod_v01(Xfloat32 *CoefPtr, Xfloat32 *Addr_Src_in, \
                     Xfloat32 *Addr_Dest_in, \
                     u32 FirSize, u32 ElemCount, u32 VectorLength) {
...
   for (chunk_indx=0; chunk_indx < ElemCount/VectorLength; chunk_indx++){
       ...
       for (n=0; n < FirSize/4; n++) {
       ..
              _VLD(VREG_01, VF_0, Addr_Src+4*n);          // load VREG_01
              _VLD(VREG_02, VF_0, Addr_Src+4*n+1);        // load VREG_02
              ...
              _VADD(VREG_MAC, VREG_MAC, VREG_03, VF_0);
              ...
       }
       _VST(VREG_MAC, VF_0, Addr_Dest);
     ...
     Addr_Src = Addr_Src + VectorLength;
     Addr_Dest = Addr_Dest + VectorLength;

  }
...
}
```

(b)

Figure 3.2 (a) DMA transfer utilities and (b) implementation of a FIR kernel.

Five vector intensive programs, namely 32-tap FIR filtering, 32-point decimation-in-time radix-2 butterfly FFT, 1024x1024 dense matrix multiplication (MM), LU decomposition, and Sparse Matrix Vector Multiplication (SpMVM) were tested on VP architecture. The routines for the VP were hand-coded, trying to improve the instruction throughput by using data prefetch via load instructions. Figure 2.8 from Chapter 2 shows how the main routine of each MicroBlaze processor is built for CTS sharing. With FTS and VLS sharing, there is no exclusive access to the VP, so STEPs 2.1 and 2.3 are removed; that is, a request for VP resources can be granted without waiting for the VP to be idle. Except for LU decomposition, each MicroBlaze uses its own partition in the VM,

and there are no data dependencies between threads running on the two processors. In order to have exclusive access to the single DMA module, the Mutex IP core provided by Xilinx is used. The lock and unlock procedures for the DMA module require locking and unlocking the Mutex, respectively. For an in-depth evaluation of the architecture, for each benchmark several performance-power scenarios are created; this involves loop unrolling, different vector lengths and instruction rearrangement optimizations.

32-tap FIR filtering (**FIR32**) is implemented using the outer product [Sung and Mitra, 1987] that avoids the reduction operation. Using a loop of 32 iterations and a given vector length for the VL, VL results are computed at the end of the loop. 45 FIR scenarios were produced for various combinations of: (i) CTS, FTS and VLS VP sharing contexts; (ii) vector lengths of 32, 64, 128 and 256; (iii) no loop unrolling, or unrolling once or three times; and (iv) instruction rearrangement optimization. All vector memory accesses are unit-strided.

FFT on 32 elements is implemented using a five-stage butterfly; each stage involves complex multiply and add vector operations, and a shuffle operation. 12 scenarios were produced for various combinations of: (i) CTS, FTS and VLS contexts; (ii) vector lengths of 32 and 64; (iii) no loop unrolling or unrolling once; and (iv) instruction rearrangement optimization. Since the number of vector registers for FFT is more than 16, in a 8-lane configuration of the VP, the maximum vector length cannot be greater than 64 (see Table 2.3). The VP routines contain indexed loads with deterministic index and shuffle operations with deterministic non-unit stride patterns (butterfly).

MM is based on the same procedure as FIR filtering using Single-precision real Alpha X Plus Y (SAXPY) in a loop to obtain one row result at the end of the loop; 21

scenarios were produced for combinations of: (i) CTS, FTS and VLS contexts; (ii) vector lengths of 32, 64, 128 and 256; (iii) no loop unrolling or unrolling once; and (iv) instruction rearrangement optimization.

LU decomposition consists of generating the L and U matrices from a dense 128×128 matrix using the Doolittle algorithm [Golub and Van Loan, 1996]. As the number of nonzero elements decreases, the value of VL is successively decremented using AnyVL support during Gaussian elimination, starting with 128 and then becoming 64, 32 and 16. Therefore, the time for LU decomposition depends on the execution times for VL between 128 and 1. Three scenarios were produced corresponding to the CTS, FTS and VLS contexts. Under FTS and VLS, the workload is split evenly between the two MicroBlaze processors.

Sparse Matrix Vector Multiplication (**SpMV**) is implemented using the data in the Compressed Row Storage (CSR) format and consists of two stages. In the first stage (named *SpMV_k1*) the array values are multiplied with the corresponding elements from the vector and in the second stage (named *SpMV_k2*) addition along each row is performed. In order to speed-up the addition stage, the rows of the sparse matrix were stored in increasing order of their number of non-zero elements. The Load Index instruction is intensively used in both stages (the index vector has random values corresponding to the column position in the sparse matrix). Therefore, the non-uniform access of the LDST units to VM banks produces contention in the crossbar such that the crossbar throughput never reaches 100%. This case is similar with Head of Line (HOL) blocking in input buffered switches. Thus, as the number of lanes M increases, the LDST throughput of each lane is expected to decrease. As M goes to infinity, the throughput

goes to 58.6% for uniform random I/O patterns [McKeown, 1999]. However, usually, better throughput is obtained because besides the load index instructions there are unit-stride load/store instructions in the LDST instruction stream.

CHAPTER 4

ANALYSIS OF PERFORMANCE AND POWER RESULTS

Sharing a Vector Processor in a multicore system as an accelerator for computation-intensive tasks could greatly increase the overall throughput through DLP and TLP at low area and power costs. The evaluation procedure and results to support this argument are shown in this chapter.

Section 4.1 presents the evaluation procedure; Section 4.2 presents relevant performance, power and energy results for popular vector-dominant floating-point applications and it is followed by a comparative analysis. Section 4.3 analyzes the performance scalability. Section 4.4 presents the quality of service results as per Section 2.2.2 and the Chapter ends with conclusions summarized in Section 4.5.

4.1 Evaluation Procedure

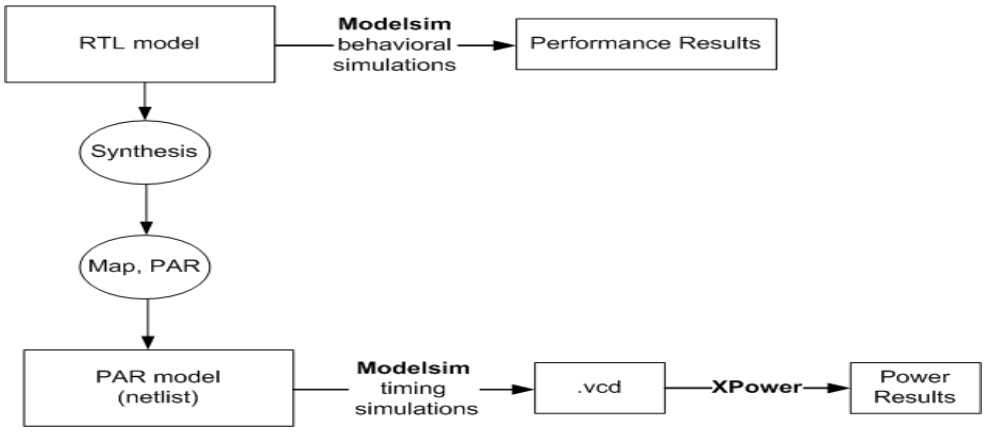


Figure 4.1 Evaluation Procedure.

Figure 4.1 displays the evaluation methodology used to evaluate the VP sharing contexts. Execution times and the utilization of lane units were obtained with ModelSim

simulations using the RTL system model. The Xilinx XPower tool [Xilinx Inc., 2010a] is then used to calculate the dynamic power dissipation based on data stored in the simulation record files (.vcd files recording the switching activities of all the logic and wires in the FPGA, which are generated by ModelSim during the timing simulations with the place-and-route netlist). Static power is computed based on total static (also called quiescent) power of the entire FPGA device and the percentage of resource occupied by the implemented design:

$$P_{ST}^{VP} = P_{ST}^{FPGA} \frac{Design\_Resources\_Count}{Total\_FPGA\_Resources\_Count} \quad (4.1)$$

To obtain realistic power figures, the timing simulations employed real floating-point input data. In all power calculations, all the design nets were matched; i.e., toggle information is extracted from all the nets in the netlist. Besides the execution times under various scenarios, figures for the average utilization of the ALU and LDST units (per vector lane) are also produced. The ALU average utilization is defined as the average number of results produced by a lane's arithmetic and logic execution unit in 100 clock cycles, and the LDST utilization is the average number of data words sent or received to or from the MC crossbar in 100 clock cycles. The peak performance of a unit has a utilization of 100.

4.2 Performance and Power Results

All the performance and power results from this section were obtained for the Virtex-5 FPGA device. Tables 4.1-4.5 show the ALU and LDST utilization and performance results in reference to the execution time for various configurations of the

system: a) one scalar processor working without the VP and the DMA unit, and all data and instructions are pre-stored in the on-chip local memory; b) two scalar processors working without the VP and the DMA unit, and all data and instructions are pre-stored in the on-chip local memory; c) a scalar processor using exclusively the VP and the DMA unit (this represents CTS); d) two scalar processors working with the VP in the FTS context and the shared DMA unit; e) two scalar processors working with the VP in the VLS context and the shared DMA unit (each MicroBlaze acquires four lanes); and, for fair comparisons across platforms, f) 3.2 GHz Intel Xeon SL7DX (Nocona) processor in a commercial PC running the same algorithmic implementation as the scenarios except that the vectorized code is replaced with sequential C code (standard); the compilation is done using the O3 option with no vectorization (no SSE extensions); and g) the same Xeon processor running optimized routines with the FFTW library for FFT [Frigo and Johnson, 2005], Intel Integrated Performance Primitives (IPP) [Intel IPP, 2010] for FIR and LU factorization and Math Kernel (MKL) [Intel MKL, 2011] libraries for matrix multiplication; the compilation is done using the O3 option with SSE3 vector extensions. For each one of the c), d) and e) configurations, the results for three distinct scenarios that combine different vector lengths with loop unrolling are presented. For FIR filtering, the results are shown in ns per dot product. For FFT, the results are in μs per 32-point complex FFT operation, and for MM the results are in μs for the calculation of a single element in the product matrix. Besides the total execution time for the LU decomposition of a 128×128 dense matrix, Table 4.4 shows the time to process one single row for various vector lengths. Since recording a .vcd file for an entire LU decomposition task is impractical due to its size, Table XIV shows the power and energy dissipation for one

processed row in Gaussian elimination. The SpMVM kernel uses *bccsstk13* matrix from the Matrix Market [Mtx Market, 2007] as input data, and the performance and energy results are presented per resulting vector (averaged over 2003 SPFP elements).

Table 4.1 Performance Comparison for 32-tap FIR

		Average utilization (%)		Execution Time (ns)	Speedup
		ALU	LDST		
One MB w/o VP		N/A	N/A	4060	1
Two MB w/o VP		N/A	N/A	2030	2
CTS	VL=32; no loop unrolled	17.51	8.86	371.25	10.93
	VL=128; no loop unrolled	39.24	19.94	165.56	24.52
	VL=128; unrolled three times	83.31	42.51	78.31	51.85
FTS	VL=32; no loop unrolled	34.97	17.70	186.01	21.83
	VL=128; no loop unrolled	75.66	38.24	85.98	47.22
	VL=128; unrolled three times	99.71	50.67	65.19	62.27
VLS	VL=32; no loop unrolled	27.68	14.09	234.12	17.34
	VL=128; no loop unrolled	49.51	25.29	131.28	30.92
	VL=128; unrolled three times	89.89	45.71	72.21	56.22
FTS	VL=4; unrolled three times	9.47	4.74	685.11	5.92
VLS		10.94	5.83	593.24	6.84
GPP Xeon - standard		N/A	N/A	340.08	11.94
GPP Xeon - IPP library		N/A	N/A	9.23	439.87

Table 4.2 Performance Comparison for 32-point Complex FFT

		Average utilization (%)		Execution Time (μs)	Speedup
		ALU	LDST		
One MB w/o VP		N/A	N/A	160.01	1
Two MB w/o VP		N/A	N/A	80.01	2
CTS	VL=32; no loop unrolled	43.29	23.38	3.264	49.02
	VL=32; unrolled once	65.10	34.78	2.172	73.66
	VL=64; unrolled once	78.92	43.09	1.782	89.78
FTS	VL=32; no loop unrolled	76.28	42.39	1.844	86.76
	VL=32; unrolled once	87.20	46.44	1.618	98.89
	VL=64; unrolled once	89.45	48.60	1.573	101.72
VLS	VL=32; no loop unrolled	62.74	35.11	2.192	72.99
	VL=32; unrolled once	74.23	41.60	1.848	86.58
	VL=64; unrolled once	79.18	44.56	1.701	94.06
GPP Xeon - standard		N/A	N/A	100.01	1.60
GPP Xeon - FFTW		N/A	N/A	0.312	512.85

For FIR, FFT, MM and SpMV in the VLS and FTS contexts, both scalar processors run the same routine. For all benchmarking scenarios under CTS that keep the VP active throughout execution, the performance is independent of the number of

involved cores and threads. Compared to the classic implementation where a VP is always tied to the same scalar processor, the advantage of CTS in a multicore environment is that VP ownership can change dynamically for more robust application realization.

Table 4.3 Performance Comparison for Matrix Multiplication

		Average utilization (%)		Execution Time (µs)	Speedup
		ALU	LDST		
One MB w/o VP		N/A	N/A	130.90	1
Two MB w/o VP		N/A	N/A	65.45	2
CTS	VL=32; no loop unrolled	20.37	20.70	10.09	12.97
	VL=32; unrolled once	33.94	34.50	6.03	21.71
	VL=128; unrolled once	68.30	69.51	3.01	43.49
FTS	VL=32; no loop unrolled	40.59	41.29	5.055	25.89
	VL=32; unrolled once	67.09	68.20	3.048	42.95
	VL=128; unrolled once	97.32	98.91	2.114	61.92
VLS	VL=32; no loop unrolled	33.83	34.34	6.086	21.51
	VL=32; unrolled once	53.51	54.45	3.791	34.53
	VL=128; unrolled once	81.88	83.40	2.494	52.48
GPP Xeon - standard		N/A	N/A	20.56	6.36
GPP Xeon - MKL library		N/A	N/A	0.651	201.38

Table 4.4 Performance Comparison for LU Decomposition

		Average utilization (%)		Execution Time (µs) per row of size VL	Execution Time (µs) for entire LU dec.	Speedup
		ALU	LDST			
One MB w/o VP		N/A	N/A	N/A	1,034,340	1
Two MB w/o VP		N/A	N/A	N/A	517,170	2
CTS	VL=16	4.73	5.34	0.632	5,137	201.35
	VL=32	9.88	10.36	0.632		
	VL=64	20.11	20.42	0.632		
	VL=128	40.44	40.54	0.632		
FTS	VL=16	8.32	8.54	0.312	2,568	402.78
	VL=32	18.74	21.08	0.316		
	VL=64	39.93	41.36	0.316		
	VL=128	81.05	82.30	0.316		
VLS	VL=16	8.70	11.11	0.316	3,522	293.68
	VL=32	19.05	21.03	0.316		
	VL=64	39.62	41.05	0.316		
	VL=128	53.86	54.95	0.472		
GPP Xeon - std		N/A	N/A	N/A	89,060	11.62
GPP Xeon (IPP)		N/A	N/A	N/A	587	1762.08

Table 4.5 Performance Comparison for Sparse Matrix Vector Multiplication (Eight Lanes and Eight Memory Banks Configuration); Sparse Matrix is *bcsstk13*; Utilization and Time is Averaged Over one Dense Row (2003 Elements)

		Average utilization (%)		Execution Time (µs)	Speedup
		ALU	LDST		
One MB w/o VP		-	-	59,018	1
Two MB w/o VP		-	-	29,509	2
CTS	SpMV_k1 VL=32 nu SpMV_k2 VL=32 nu	**9.35**	**20.90**	3,378	17.48
FTS	SpMV_k1 VL=32 nu SpMV_k2 VL=32 nu	**18.22**	**39.39**	1,711	34.49
VLS	SpMV_k1 VL=32 nu SpMV_k2 VL=32 nu	**14.79**	**33.11**	2,020	29.22
GPP Xeon - standard		-	-	8,401	7.025

Table 4.6 Average Execution Time (µs) for the 32-tap FIR Routine with Various Statistical Average Stall Ratios (VL=128; Unrolled Three Times)

		Average stall ratio (%)				
		0	25	50	75	100
One CPU with VP	VL=128; unrolled three times.	78.31	98.25	117.78	137.55	157.42
CTS		78.31	78.54	79.19	86.95	92.07
FTS		65.19	69.84	76.11	83.15	91.61
VLS		72.21	73.86	78.44	85.01	92.81

From these performance results the following conclusions can be made:

i) The best performance is provided by FTS followed by VLS and CTS;

ii) A higher VL value increases the data-level parallelism, and therefore the performance.

iii) Loop unrolling increases the utilization of the units and also the overall performance.

iv) With a low utilization of the units the speedup doubles from CTS to FTS (see VL=32 without loop unrolling for FIR, FFT, MM and LU); moreover, if the utilization from each thread is less than 50%, the speedup of FTS almost doubles as compared to CTS.

v) For kernels with a high utilization of the lane units in the CTS mode, FTS can provide a speedup of 1.2 to 1.5 as compared to CTS. This is caused by the fact that FTS achieves close to 100% utilization (peak performance) and the VP can no longer accommodate more instructions in its pipeline.

vi) Thread-level parallelism can provide higher speedup than data-level parallelism and loop unrolling (for FFT, FTS with VL=32 and without loop unrolling yields almost the same performance as CTS with VL=64 and the loop unrolled once). Therefore, the lack of data-level parallelism and inadequate compiler optimization (loop unrolling) for an application can be alleviated by simultaneously processing an additional thread.

vii) LU decomposition exhibits low utilization for low vector lengths. This is caused by the scalar code run by MicroBlaze that involves one floating-point division and two memory accesses per processed row; it can fully overlap VP code runs. As a consequence, two scalar processors in the FTS context provide a speedup of two as compared to the CTS context. This is a good example of applications where the fraction of sequential code is substantial and the utilization of the VP accelerator is low. Thus, adding threads from two or more processors will increase the speedup almost linearly for the same VP resources.

There are cases where VLS can provide better results than FTS. Table 4.1 presents a scenario where each core issues instructions for FIR kernels requesting vector length smaller than the number of VP lanes. Since in VLS four exclusive lanes are assigned to each core, all eight lanes will be used. In FTS, four lanes will be idle in each execution cycle since all eight lanes simultaneously receive the same vector instruction. Therefore, for small vector sizes FTS forces several lanes to be idle, thus yielding performance inferior to VLS. CTS will perform worse than both since only one thread that utilizes half of the lanes is active in each cycle. Contrary to FTS, however, a thread that enters the VP under CTS completes execution without any interruption as long as all dependencies can be resolved internally. As compared to Xeon standard routines, FTS provides a speed-up between 5 (for FIR) and 63 (for FFT) despite the much lower operating frequency of the FPGA-based prototype. On the other hand, highly optimized routines running on Xeon outperformed all VP sharing schemes. However, if the

execution times are translated into clock cycles for fairness since FPGA implementations run at much lower clock frequencies, then the best FTS-based scenario for FIR32 consumes just 8.15 clock cycles as compared to 29.54 cycles for Xeon; these numbers are averages for a single FIR32 run obtained after running a large number of consecutive FIR32 routines. In this case, the FTS-based cycle speedup is 3.62. The best FTS-based scenario for a complex FFT32 routine consumes 196 clock cycles while Xeon takes 998 cycles, for a speedup of 5.09. The 1024×1024 matrix-multiplication FTS scenario takes 262.75 cycles as compared to 2080 cycles for the respective optimized MKL matrix function running on Xeon, for a resulting 7.92 speedup. Finally, 321,078 clock cycles are taken by FTS to compute LU decomposition as compared to 1,878,411 cycles on Xeon, for a 5.85 speedup. Therefore, with the performance is expressed in clock cycles, the VP sharing techniques demonstrate 3.62-7.92 speedups compared to optimized Xeon runs.

In many cases, a thread may stall at various times. Stalls may occur during the execution of a single or multiple threads running on a single core with a dedicated VP, or during the execution of threads running on multiple cores sharing a VP. Table 4.6 shows the average execution time for scenarios where each core runs FIR routines of random size interleaved with stalls of random duration. The stall ratio is defined as the ratio between the average duration of a stall and the average time that the routine utilizes the VP. Without stalls (i.e., the ratio is zero), CTS provides the same performance as a single core attached to a VP with the same total number of lanes (eight in the prototype); FTS gives the best performance. As the stall ratio increases, the performance between CTS and a single core with a VP increases. Also, the performance numbers for CTS and VLS approach that of FTS and become almost identical for a stall ratio of 100%.

Table 4.7 Power Comparison for 32-tap FIR

		Dynamic Power (mW)		Energy (nJ)		nJ/FLOP
		VP	**VP, Crossbar and Memory**	**Dynamic**	**Total**	
One MB w/o VP		N/A		225.37	380.78	5.951
CTS	VL=32; no loop unrolled	92.02	114.19	42.39	190.89	2.982
	VL=128; no loop unrolled	185.43	225.66	37.36	120.14	1.877
	VL=128; unrolled three times	398.40	479.28	37.53	68.85	1.075
FTS	VL=32; no loop unrolled	182.37	220.98	41.10	115.50	1.804
	VL=128; no loop unrolled	359.56	432.74	37.21	71.61	1.118
	VL=128; unrolled three times	474.41	567.82	37.01	63.09	0.985
VLS	VL=32; no loop unrolled	140.84	187.76	43.96	137.61	2.150
	VL=128; no loop unrolled	238.09	319.13	41.89	94.41	1.475
	VL=128; unrolled three times	429.01	554.97	40.07	68.96	1.077
CTS VL=32; no loop unrolled 4 lanes used; other 4 lanes are power gated.		69.50	93.51	43.76	148.59	2.325

The dominant cause of dynamic power consumption is the charging and discharging of parasitic capacitance within the device as it manipulates or moves data during computation. Static power, dominated by the gate and sub-threshold leakage currents, increases as transistor shrinks, and is a major concern at 40 and 45nm. Smaller channel lengths and thinner oxide gates make it easier for current to "leak," either across the channel region or through the gate oxide of the transistor. As can be seen in Figure 4.2, starting with 90nm technology node, the reduction in leakage power is less than the reduction in dynamic power [Xilinx wpp, 2009]. Static power is becoming an important component on the total energy budget and the power results confirm that the contribution of static power to total power budget is between 32% and 80%.

Table 4.8 Power Comparison for 32-point Complex FFT

		Dynamic Power (mW)		Energy (nJ)		nJ/FLOP
		VP	VP, Crossbar and Memory	Dynamic	Total	
One MB w/o VP		N/A		8562.13	14687.38	22.949
CTS	VL=32; no loop unrolled	195.66	233.59	762.40	2068.01	3.231
	VL=32; unrolled once	279.46	330.07	716.91	1585.71	2.477
	VL=64; unrolled once	337.21	398.79	710.64	1423.44	2.224
FTS	VL=32; no loop unrolled	344.96	405.14	747.07	1484.46	2.319
	VL=32; unrolled once	390.97	456.32	738.32	1385.52	2.164
	VL=64; unrolled once	395.12	460.97	725.11	1352.72	2.113
VLS	VL=32; no loop unrolled	302.43	356.43	781.29	1658.09	2.590
	VL=32; unrolled once	347.54	406.09	750.45	1489.65	2.327
	VL=64; unrolled once	352.23	429.24	730.14	1410.53	2.203
CTS VL=32; no loop unrolled 4 lanes used; other 4 lanes are power gated.		147.87	178.30	781.66	1763.68	2.755

Table 4.9 Power Comparison for MM

		Dynamic Power (mW)		Energy (nJ)		nJ/FLOP
		VP	VP, Crossbar and Memory	Dynamic	Total	
One MB w/o VP		N/A		7806.88	12817.73	6.258
CTS	VL=32; no loop unrolled	131.68	166.22	1677.16	5713.16	2.789
	VL=32; unrolled once	234.75	296.69	1787.85	4198.25	2.049
	VL=128; unrolled once	433.28	555.02	1671.16	2875.68	1.404
FTS	VL=32; no loop unrolled	263.99	332.86	1682.61	3704.60	1.808
	VL=32; unrolled once	482.95	610.20	1859.89	3079.08	1.503
	VL=128; unrolled once	621.84	793.65	1668.25	2509.05	1.225
VLS	VL=32; no loop unrolled	222.48	311.86	1897.98	4332.38	2.115
	VL=32; unrolled once	386.59	513.59	1947.02	3463.42	1.691
	VL=128; unrolled once	508.44	668.76	1667.89	2665.49	1.301

Table 4.10 Power Comparison for LU Decomposition

		Dynamic Power (mW)		Energy (nJ) per row processed		nJ/FLOP
		VP	VP, Crossbar and Memory	Dynamic	Total	
One MB w/o VP (row length 128)		N/A		2559.51	4473.54	17.473
CTS	VL=16	37.10	46.03	29.09	281.89	8.809
	VL=32	68.54	85.46	52.01	306.81	4.794
	VL=64	130.33	164.71	104.09	356.89	2.788
	VL=128	250.37	317.69	200.78	453.58	1.771
FTS	VL=16	68.59	87.24	27.21	152.01	4.750
	VL=32	105.12	132.95	48.01	168.41	2.631
	VL=64	198.56	252.94	85.92	206.32	1.611
	VL=128	371.26	471.15	192.88	275.28	1.075
VLS	VL=16	64.24	89.82	28.74	156.74	4.898
	VL=32	114.05	157.59	49.79	176.19	2.763
	VL=64	214.53	290.38	91.76	218.16	1.704
	VL=128	311.84	422.28	203.31	388.11	1.515

Table 4.11 Power Comparison for Sparse Matrix Vector Multiplication (Eight Lanes and Eight Memory Banks Configuration); Sparse Matrix is *bcsstk13*; Utilization and Time is Averaged over One Dense Row (2003 Elements)

		Dynamic Power (mW)		Energy (nJ)/Vector Result		nJ/FLOP
		VP	VP, Crossbar and Memory	Dynamic	Total	
CTS	SpMV_k1 VL=32 nu SpMV_k2 VL=32 nu	35.13	51.04	172,278	1,537,900	9.167
FTS	SpMV_k1 VL=32 nu SpMV_k2 VL=32 nu	67.34	104.11	177,944	862,344	5.141
VLS	SpMV_k1 VL=32 nu SpMV_k2 VL=32 nu	58.45	89.46	179,780	987,780	5.888

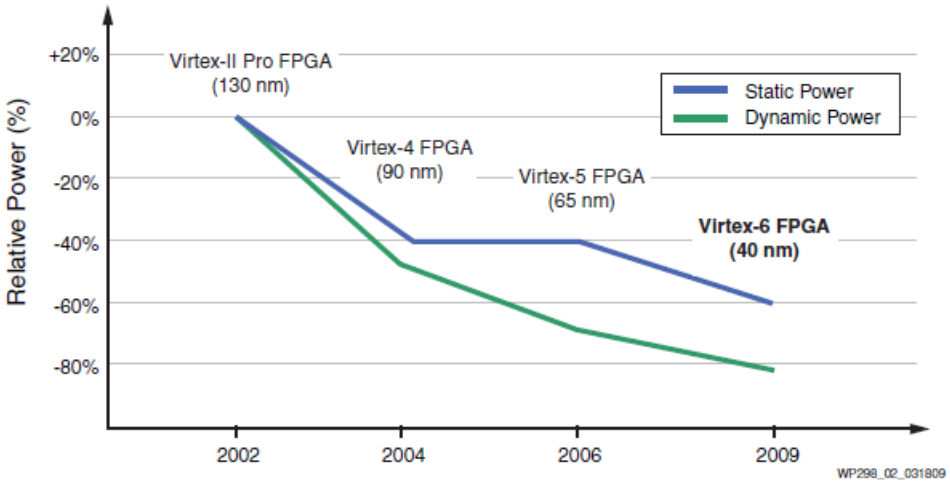


Figure 4.2 Relative power reduction of different Xilinx Virtex FPGA families.

Source: Xilinx white paper wp298 [Xilinx wpp, 2009].

The power Tables 4.7-4.11 show that:

(i) The lowest dynamic energy is provided by FTS followed by CTS and VLS, with the values having a small dispersion.

(ii) However, if static power is included, the advantage of FTS and VLS is substantial, especially for low average utilization (see the FIR benchmark for CTS, FTS, and VLS with VL=32 and no unrolling).

(iii) Adding a new core that runs a thread has almost the same performance gain and total energy consumption as doubling the data-level parallelism and unrolling the loop once (see FFT under CTS with VL=64 and loop unrolled once as compared to FFT under FTS with VL=32 and without loop unrolling).

(iv) Under similar LDST utilization, the MC crossbar and VM dynamic power consumption is higher in VLS than in any other VP sharing context. This is because of high contention in the crossbar due to the presence of two LDST threads corresponding to two distinct VPs, with no synchronization for accessing the VM. Similar behavior has been observed for SpMV scenarios; under the same LDST utilization, the dynamic power consumption of the crossbar increases for sparse scenarios as compare with FIR and MM.

Tables 4.7-4.11 also contain energy figures for a MicroBlaze without the VP. The conclusion is that the best VP sharing scheme consumes 5 to 16 times less energy per operation than MicroBlaze. The power analysis the Xeon general-purpose processor is not included since it has very high power consumption (103 Watts) and is not suitable for high-performance embedded applications.

VP sharing in FTS with an increased number of cores requires either more vector controllers, one per core, or the capability of a controller to handle multiple threads coming from many attached cores. Simulations for the latter approach where each core in the prototype emulates a dual-threaded microprocessor are carried out. This approach suffices for current systems that normally contain less a dozen cores. The FTS results show high throughput for threads with low VL and no loop unrolling because in this case FTS can accommodate the simultaneous execution of multiple threads, thus increasing the VP throughput. However, if individual threads have high utilization of VP resources, it will be difficult to accommodate simultaneously more threads under FTS. For example, the overall throughput of FIR with VL=128 and no loop unrolling is increased by about 20% with four threads compared to two threads. On the other hand, with loop unrolling FTS cannot easily facilitate additional threads for FIR since the utilization per thread is already 83%. VLS can facilitate better scalability if the lanes are assigned to the threads

in a manner similar to the allocation of pages in virtual memory implementations. However, a study must be made of lane fragmentation and interconnection problems. To further improve scalability for increased numbers of cores, the design of suitable networks to interconnect cores to vector controllers is needed.

Table 4.12 Advantages and Disadvantages of the VP Sharing Schemes

	CTS	VLS	FTS
Advantages	Simple to implement. No per instruction scheduling. Can take advantage of stalls in VP routines to increase the average utilization.	No per instruction scheduling. Increases utilization (due to increased number of elements per lane corresponding to one vector register).	Increases the overall throughput. Increases instantaneous utilization by mixing VP instructions from two or more cores in any lane. Low energy per operation.
Disadvantages	Low throughput since the instantaneous utilization does not increase (still one thread runs at any time). High energy per operation, especially for kernels with low VP utilization.	A single thread uses a lane. Crossbar dynamic power higher due to potential contentions. Complex task to assign lanes, especially if more than two cores share the VP. It can result to lane fragmentation problems for VPs with large numbers of lanes.	Needs arbitration (the complexity increases if more than two cores share the VP). Requires register renaming. May give worse results than VLS when the vector length is less than the number of lanes.

Table 4.7 (last row) also shows the power and energy figures when a scalar processor issues VP instructions to four lanes. If the static power for the other four lanes is ignored, the total energy consumption is lower as compared to using all eight lanes (CTS with VL=32 and without an unrolled loop). Thus, under low utilization the energy consumption due to the static power is substantial; it then becomes imperative to decrease the number of active lanes and power gate the idle ones. Even if the actual FPGA technologies do not facilitate power gating, next chapter discuses the finding of optimum

number of lanes for given LDST and ALU utilizations that minimizes the total energy consumption. Finally, Table 4.12 summarizes the advantages and disadvantages of the VP sharing schemes.

4.3 Performance Scalability

In order to analyze the scalability of the proposed VP-sharing schemes, four of the applications are benchmarked for VP configurations with 2, 4, 8, 16 and 32 lanes [Beldianu et al., 2011c]. Also, just for performance evaluation purposes, the design supports also a parameterized implementation where the execution unit can instantiate a fused floating point multiply-add (MADD) or floating point divide unit. Since the FPU has only two read ports to VRF, the third operand in the multiply-add instruction is always a scalar supplied by one of the scalar processors thorough FSL channel.

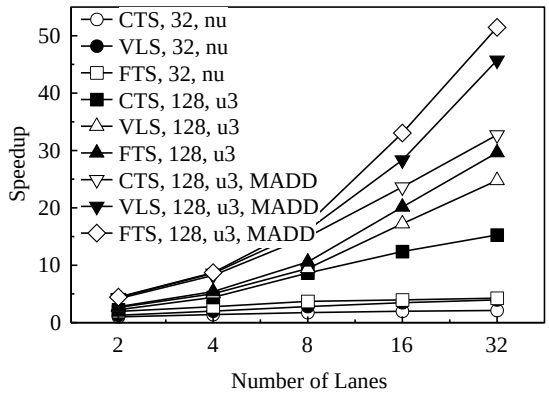


Figure 4.3 FIR routine for 2, 4, 8, 16 and 32 lanes configuration. Each application consists in sharing context, Vector Length, unroll type (nu=no unroll; u3=unrolled three times), and with or without VMADD instruction extension.

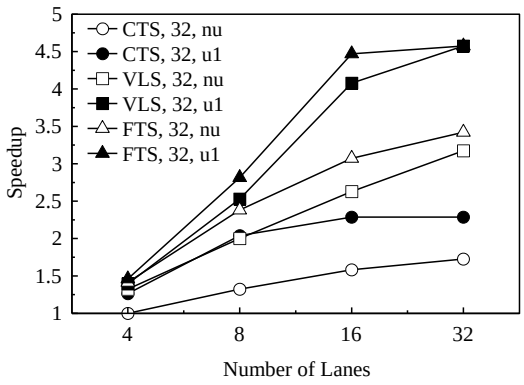


Figure 4.4 FFT routine for 4, 8, 16 and 32 lanes configuration. Each application consists in sharing context, Vector Length, and unroll type (nu=no unroll; u1=unrolled once).

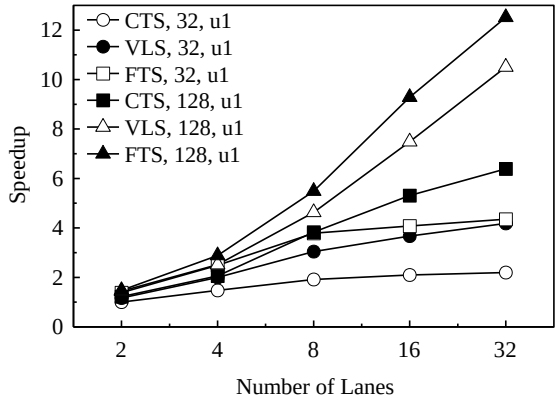


Figure 4.5 MM routine for 2, 4, 8, 16 and 32 lanes configuration. Each application consists in sharing context, Vector Length, and unroll type (u1=unrolled once).

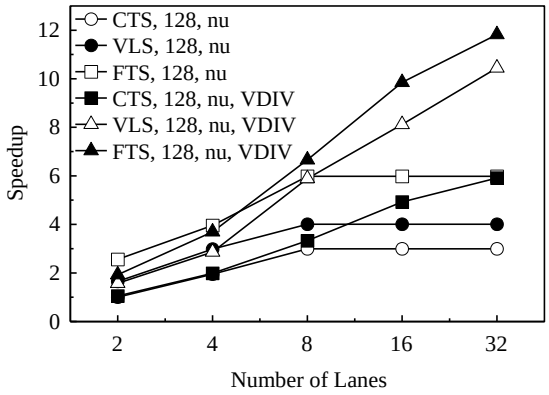


Figure 4.6 LU decomposition routine for 2, 4, 8, 16 and 32 lanes configuration. Each application consists in sharing context, Vector Length, unroll type (nu=no unroll), and with or without VDIV instruction extension.

Figures 4.3 to 4.6 show that the FTS scheme scales better than CTS and VLS. Also, the application scales better with increasing data parallelism caused by high vector length and loop unrolling. Additionally, conclusions can be summarized:

(i) For the FIR application, the fused multiply-add MADD instruction extension increases the speedup with almost 60% compared to the corresponding schemes without MADD.

(ii) In LU decomposition, all schemes without the VDIV extension provide the same performance with 8, 16 and 32 lanes in the configuration. However, the inclusion of division in the FPUs allows the offloading of the scalar processors, thus improving the performance as the number of lanes increases. It can be observed that FTS with VDIV provides almost 100% improvement in the 32-lane configuration as compared to FTS without the VDIV extension.

(iii) It should be also emphasized that for applications with low parallelism increasing the number of lanes does not improve the performance.

4.4 Guaranteed Quality of Service

In the FTS context, the application layer may require guaranteed QoS for a high priority (HP) critical thread. In Section 2.2.2 HW support for guaranteed quality of service is presented. Figure 4.7 shows the obtained relative performance of a high priority thread when it shares the VP resources in the FTS mode for different numbers of VP lanes (M lanes and M memory banks). The normalized speedup is defined as the ratio between the execution time of the thread when it runs in the CTS mode (by itself - that is, one thread running at full speed) and the execution time when it runs in the FTS sharing context. A maximum value of one shows that the HP thread runs unaffected by the low priority thread (guaranteed quality of service). Three scenarios are presented:

(i) Two FIR threads: the quality of service is guaranteed for the HP thread. With no contention on the crossbar, the low priority (LP) thread will access the remaining pipeline slots and, at 16 lanes, will have the same performance as the HP thread.

(ii) FIR for the HP thread and SpMV_k1 for the LP thread; Due to I/O non-uniform access patterns exhibited by the sparse kernel, some of the LDST pipeline slots are wasted due to crossbar contentions. Thus, the LP thread will "slow down" the HP thread up to 5% for some lane configurations.

(iii) Putting together two sparse threads will affect the speed-up of the HP thread by 10-13% for 8 to 32 lanes. Also, as the number of lanes increases, the throughput of the HP thread is more affected (as per Section's 2.2.2 conclusion). One solution to alleviate contention on the crossbar is to use a number L of banks greater than the number of lanes. Statistically, the probability of contention will decrease as the M/L the ratio decreases.

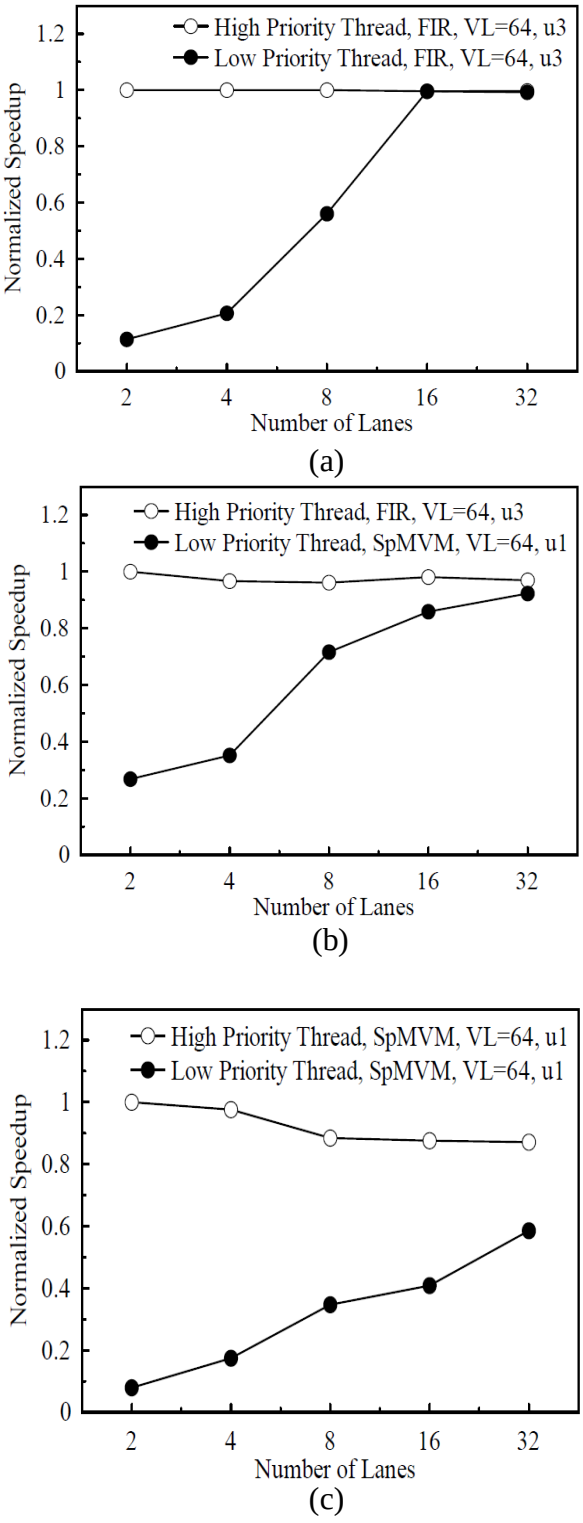


Figure 4.7 Relative performance of high priority and low priority threads on a VP with a number *M* of lanes between 2 and 32, and *M* memory banks: (a) two FIR VL=64, u3; (b) FIR VL=64 u3 & SpMV_k1 VL=64 u1; (c) two SpMV_k1 VL=64 u1 (u1 – loop unrolled once; u3-loop unrolled three times).

4.5 Conclusions

Finally, it is pertinent to summarize the main conclusions of the results presented in this chapter:

(i) The utilization of the lane units and, as a consequence the total consumed energy, can be improved by: increasing the vector length; unrolling the loop and, thus, increasing the instruction parallelism; or accommodating more than one instruction stream in the lane's functional pipelines.

(ii) The FTS context provides the best performance and energy gains followed by VLS and CTS.

(iii) Extending the VP ISA with multiply-add and division instructions increases substantially the performance of the applications that can make use of them.

(iv) Under low utilization the energy consumption due to the static power is substantial; it then becomes imperative to decrease the number of active lanes and power gate the idle ones in order to reduce the impact of leakage.

(v) In the FTS context, a high priority thread may run unaffected by its counterpart as long as the memory accesses to memory banks are uniform and unit-strided.

CHAPTER 5

PERFORMANCE AND POWER CHARACTERIZATION

The ultimate objective is to develop a robust runtime framework that can make highly accurate predictions at runtime about performance and energy figures for various VP assignments to applications. Vector lanes could then be assigned effectively to resource-competing threads in ways that could minimize thread execution times, maximize thread throughput, minimize energy consumption for guaranteed performance or independent of performance (e.g., for battery-operated devices), etc. To this extent, there is a need for highly accurate models for performance and power prediction.

Section 5.1 presents the performance model, and Sections 5.2 and 5.3 present the dynamic and static power models respectively. Finally, Section 5.4 shows the opportunity of trading the energy for performance.

5.1 Performance Model

As stated in Section 2.2, each ALU or LDST instruction finishes in $SU_{ALU/LDST} + VL/M$ clock cycles after it leaves the hazard detection stage in the VC. *VL* is the vector length, $SU_{ALU/LDST}$ is the start-up latency of ALU/LDST units and M is the number of lanes that receive this instruction. The instruction start-up time directly depends on the pipeline depth of the control stages and the functional unit implementing that instruction. In current implementation, for a LDST instruction with no contention in the crossbar the start-up time is eight clock cycles. For floating-point operations the start-up time is 13 clock cycles for multiply and add, and eight clock cycles for the rest of the instructions.

Figure 5.1 shows how SIMD instructions are executed in each lane in two distinct cases: a) consecutive instructions with data dependence such that in $SU_{ALU/LDST}+VL/M$ clock cycles only VL/M results are produced; and b) all instructions issued to lanes have no data dependence such that results are produced in each clock cycle. The average utilization of the ALU or LDST unit can be conveniently defined as the average number of ALU results produced or the average number of data transfers via the memory crossbar, respectively, in $SU_{ALU/LDST}+VL/M$ clock cycles. The number of results is the product of the average number of instructions $IP_{ALU/LDST}$ ready for execution (i.e., the average number of ALU or LDST instructions issued to VP lanes in $SU_{ALU/LDST}+VL/M$ cycles) and VL/M (i.e., the number of elements in each lane to be processed with an SIMD instruction). Equation 5.1 computes the ALU and LDST utilization.

$$U_{ALU/LDST}=\frac{IP_{ALU/LDST}\dfrac{VL}{M}}{SU_{ALU/LDST}+\dfrac{VL}{M}}=\frac{IP_{ALU/LDST}}{SU_{ALU/LDST}\dfrac{M}{VL}+1} \tag{5.1}$$

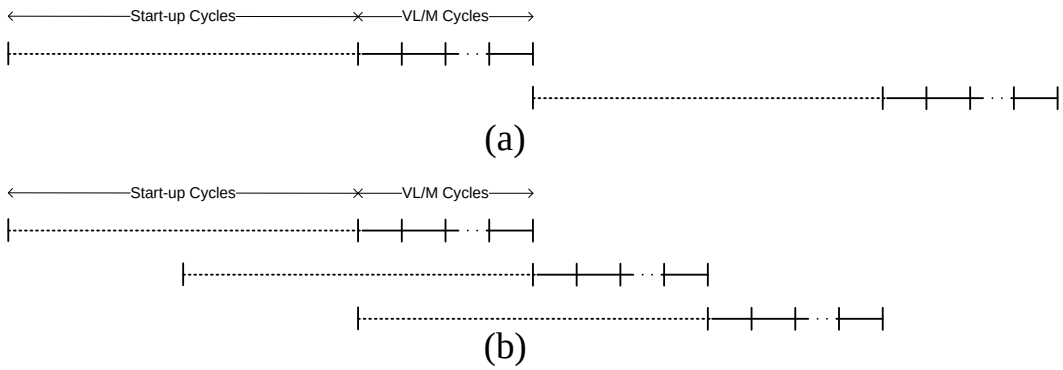


Figure 5.1 Execution of a) two data dependent instructions; b) three instructions without data dependencies.

Ideally, peak performance is achieved when there is no contention on the memory crossbar and $IP_{ALU/LDST} = SU_{ALU/LDST} \cdot M / VL + 1$, which represents the maximum instruction parallelism needed to fully utilize (saturate) one of the units. The utilization of the ALU and LDST units can be increased by:

(i) Increasing the vector length VL.

(ii) Reducing the number of lanes assigned to a VC.

(iii) Increasing the average instruction-level parallelism $IP_{ALU/LDST}$; or

(iv) Reducing the start-up time.

The first option could be used whenever possible. However, there are applications with low or difficult to identify data parallelism. The second option increases the utilization of the units but degrades the overall performance since each VP instruction takes more time to execute. Instruction-level parallelism can be increased via loop unrolling and multithreading that involves two or more scalar processors. Improving the start-up time may not be an option, especially for FPGAs, since it involves reducing the pipeline depth of the VP, and therefore the design frequency.

The utilization of a unit in a lane can be estimated at runtime as a function of the average instruction throughput $IT_{ALU/LDST}$ (i.e., the average number of vector instructions issued in 100 clock cycles) and the number of vector elements used per lane (i.e., VL/M), as per Equation 5.2. This could be implemented easily by embedding appropriate hardware counters (profilers) in the design. Actually, utilization figures presented in this work were obtained by using Equation 5.2 for observation periods representing 1000 runs of the same kernel.

$$U_{ALU/LDST} = IT_{ALU/LDST} \cdot VL / M \tag{5.2}$$

Finally, the execution time of a specific kernel is proportional to the inverse product of the ALU utilization in each lane and the number of lanes, as per Equation 5.3. $K_{\ker nel}$ is a constant dependent on the workload required for that kernel (for example, the number of FIR points computed, the number of FFTs, etc) and $M \cdot U_{ALU}$ is the overall sustained performance of the VP.

$$t_{exec} = \frac{K_{\ker nel}}{M \cdot U_{ALU}} \tag{5.3}$$

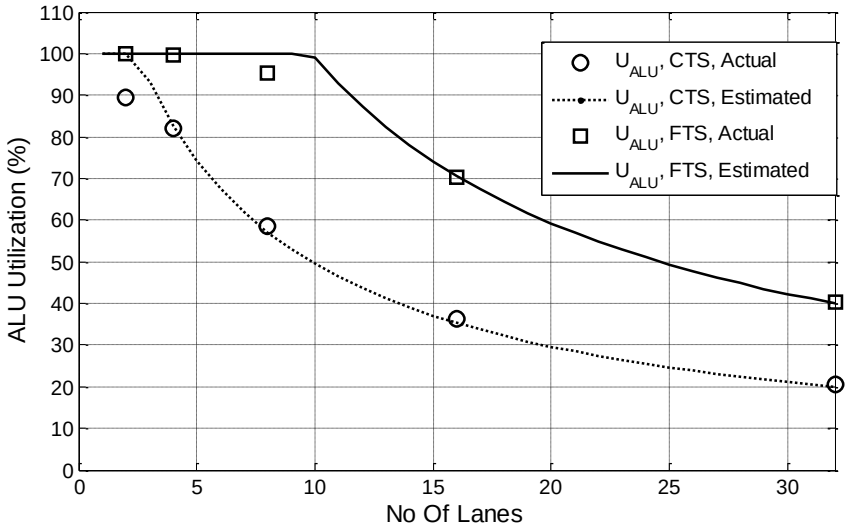


Figure 5.2 Estimated and actual ALU utilization for FIR 32 with VL=64 and loop unrolled three times ($SU_{ALU} = 13,\ IP_{ALU}^{CTS} = 1.5,\ IP_{ALU}^{FTS} = 3.0$).

Figures 5.2 and 5.3 display the estimated and actual utilization for a FIR kernel with VL=64 which is loop unrolled 3 times and for SpMV_k1 with VL=64 and loop unrolled once. For the FIR kernel, the ALU utilization is displayed; since SpMV_k1 exhibits higher utilization for the LDST unit, the LDST utilization is plotted for this kernel. The model applies well for the FIR kernel. Even, if not shown here, the same behavior is observed for I/O uniform patterns: MM and LU kernels. However, for

SpMV_k1 the model matches the actual data only for low percentages of utilization; for high utilization the behavior of random accesses to the crossbar is not straightforward to model. Also, the maximum value of the LDST utilization that can be obtained is around 92%.

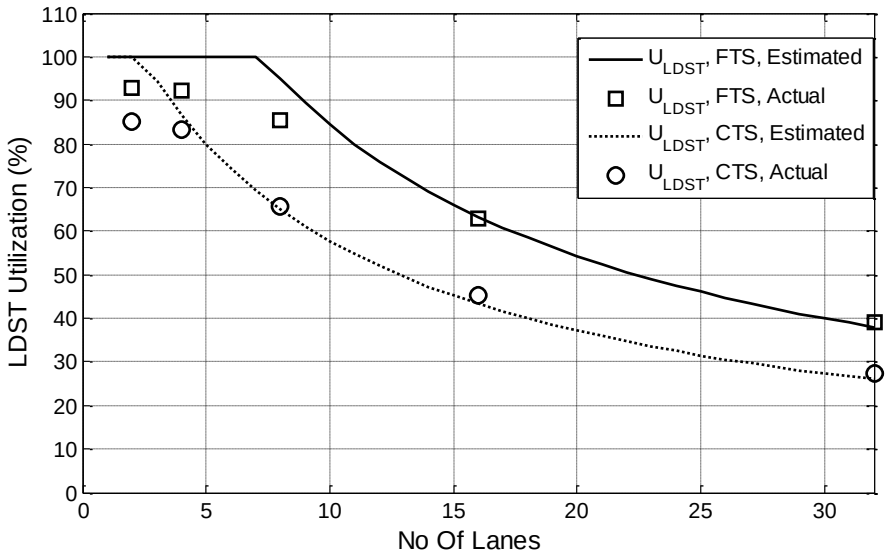


Figure 5.3 Estimated and actual LDST utilization for SpMV (kernel 1) VL=64 and loop unrolled one time ($SU_{LDST} = 8,\ IP_{LDST}^{CTS} = 1.3,\ IP_{LDST}^{FTS} = 1.9$).

5.2 Dynamic Power Model

The dynamic Power model presented in this section is based on the activity rate of the design. It resembles the activity-based strategy for estimating the average power dissipation of hard DSP and multiplier blocks embedded in FPGAs [Choy et al., 2006]. In the VP architecture, the activity rate comprises the utilization of the ALU and LDST units which further translates into instruction and data throughput. It is obtained by implementing timing simulations for many scenarios with each kernel. The model assumes a fixed combination of Voltage, Frequency and Technology, and is easy to extend since only constants change. These constants in the model are functions of the

Frequency, Voltage, Technology, Temperature, etc. Also, it should be mentioned that all the power figures are extracted from timing simulations with the Virtex-6 FPGA placed-and-routed netlist.

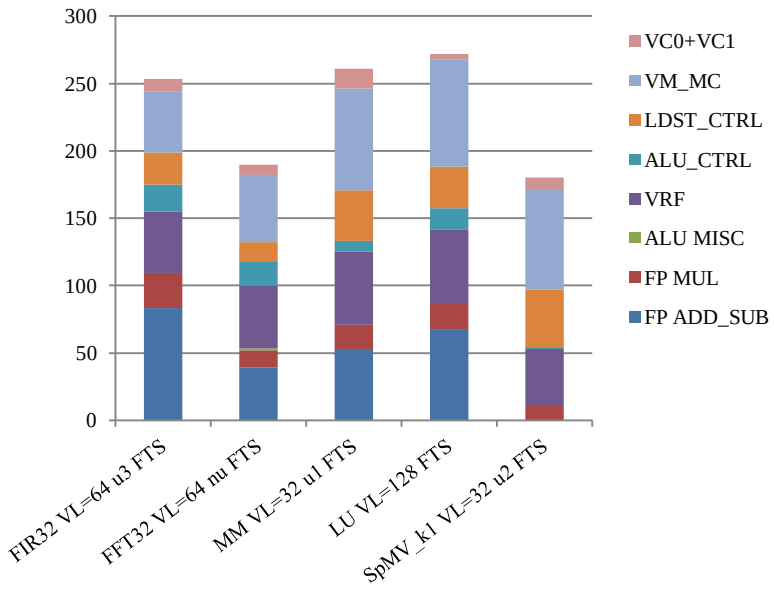


Figure 5.4 Dynamic power breakdown (in mW) for a Vector Processor with eight lanes and eight memory banks running different application kernels.

Figure 5.4 shows the power breakdown gathered from simulations on an 8×8 VP running different applications. As can be depicted, the FIR and FFT kernels exhibit a high dynamic power consumption for the arithmetic units and the register file. The MM, LU and Sparse kernels have high utilization of the LDST units and, thus, high power for the LDST controller and memory banks. Also, as can be observed in the LU case, kernels with high vector length and, as a consequence low instruction throughput, have small power consumption in the VCs (see LU) as compared to the total power consumption.

Figure 5.5 (a) shows a linear dependence between the ALU dynamic power consumption and the ALU utilization (that actually represents the ALU activity rate). Also, as shown in Figure 5.5 (b), the number of accesses to VRF is proportional to

$U_{LDST} + 2 \cdot U_{ALU}$; in most of the cases, a LDST instruction has one access to VRF, either Read or Write; an ALU instruction has one or two Reads and one Write (a fine grain model could be further developed). Therefore, the VRF dynamic power consumption is modeled as having a linear dependence on the average VRF utilization expressed as $(2U_{ALU} + U_{LDST})$.

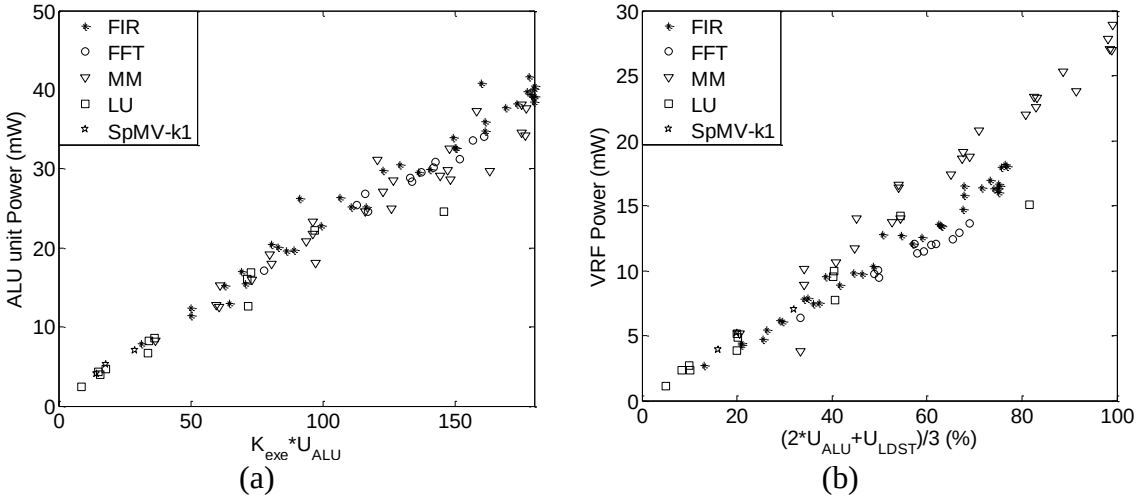


Figure 5.5 a) ALU power consumption vs. ALU utilization ($K_{exe} = \sum_i K_{exe(i)} w_i$); b) VRF power consumption vs. ALU and LDST utilization.

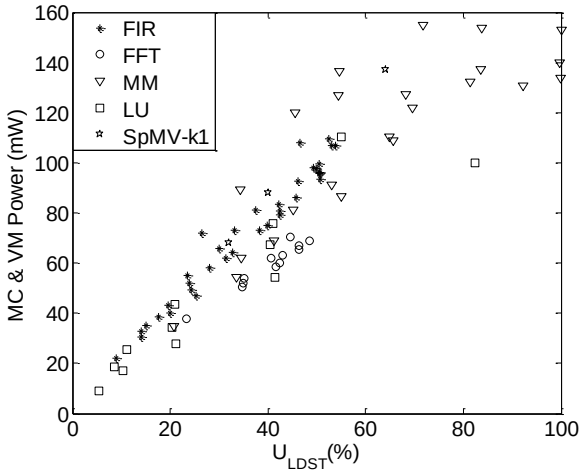


Figure 5.6 Memory Crossbar (MC) and Vector Memory (VM) power consumption vs. LDST utilization.

The LDST unit requires a complex power model; however, the LDST power can be freely expressed as a linear function of the LDST utilization. Moreover, as Figure 5.6 shows, the MC and VM dynamic power consumption also shows an almost linear dependence on the LDST utilization. The small errors are caused by fine grain effects like different memory access patterns, especially in the VLS context and SpMV, and different toggling rates in netlist signals due to the randomness of the data used in simulations.

Table 5.1 Dynamic Power Model Equations

Component	Model	Details
Instruction Queues and ALU controller	$P_{ALU\_CTRL} \approx K^{INTSR}_{ALU\_CTRL}\frac{M}{VL}U_{ALU} + K^{DATA}_{ALU\_CTRL}U_{ALU}$	Dynamic Power depends on the instruction throughput and data throughput.
ALU Execution units	$P_{ALU\_EXE} \approx \sum_i P_{exec(i)} = \sum_i K_{exec(i)}U_{exec(i)} = = U_{ALU}\sum_i K_{exe(i)}w_i$	$\sum_i w_i = 1;$ w_i is the fraction of the ALU utilization that targets the floating-point execution unit *i.*
Instruction Queues and LDST controller	$P_{LDST} \approx \left(K^{INTSR}_{LDST}\frac{M}{VL} + K^{DATA}_{LDST}\right)U_{LDST}$	Dynamic Power depends on the instruction throughput and data throughput.
Vector Register File	$P_{VRF} \approx K_{VRF}\left(2U_{ALU} + U_{LDST}\right)$	The power is a linear function of data throughput
Vector Controller	$P_{VC} \approx K_{VC}I_{TH} = K_{VC}\frac{M}{VL}\left(U_{ALU} + U_{LDST}\right)$	Exhibits a linear dependence on vector instruction throughput I_{TH}.
Memory Banks and Crossbar	$P_{MEM\_BANKs} \approx K_{MEM\_BANK}\frac{M}{L}U_{LDST}$	Extended to VPs with M lanes and L memory banks. Sparse matrix or VLS scenarios consume more power than in CTS and FTS in the arbiters due to contentions and non-uniform accesses to the crossbar.

Table 5.1 summarizes the power model equations for all VP design components. All *K*s are constant coefficients measured in mW per percent of utilization (mW/%). These equations apply if the utilization is the same for all the lanes. Otherwise, the power

consumption for each lane has to be computed separately according to its own ALU and LDST utilization figures. Using a linear approximation method the values for the K coefficients are found. They are shown in Table 5.2 along with the mean absolute estimation error for the VP, and collectively for the VP, MC and VM. The utilization of the lane units can be used to estimate the dynamic power consumption within a 10% confidence interval.

The total dynamic power for M lanes and L memory banks can be expressed as:

$$P_{TOTAL}^{D} = 2P_{VC} + MP_{LANE} + LP_{MEM\_BANKs} = 2P_{VC} + M\left(P_{ALU\_CTRL} + P_{ALU\_EXE} + P_{LDST} + P_{VRF}\right) + LP_{MEM\_BANKs} \quad (5.4)$$

Table 5.2 Mean Absolute Error for Dynamic Power Estimation

	$w_{ADD\_SUB} / w_{MUL} / w_{MISC}$	**Mean Absolute Error (%)**	
		VP	**VP, MC and VM**
FIR	0.48/0.48/0.04	6.83	7.89
FFT	0.36/0.36/0.27	8.98	10.43
MM	0.5/0.5/0	6.29	7.74
LU	0.5/0.5/0	8.72	9.76
SpMVM_k1	0/0.99/0.01	7.98	10.11
SpMVM_k2	0.96/0/0.4	10.20	13.72
OVERALL		8.16	9.95
By linear approximation ($\mu W / \%$) $K_{ALU\_CTRL}^{INTSR} \approx 28$ $K_{ALU\_CTRL}^{DATA} \approx 18$ $K_{ADD\_SUB} \approx 215$ $K_{MUL} \approx 71$ (uses DSP48E1) $K_{MISC} \approx 18$		$K_{LDST}^{INTSR} \approx 34$ $K_{LDST}^{DATA} \approx 55$ $K_{VRF} \approx 34$ $K_{VC} \approx 240$ $K_{MEM\_BANK} \approx 147$	

For a given kernel application, the ratio $U_{LDST} / U_{ALU} = \alpha$ is constant and the total dynamic power can be estimated by Equation 5.5.

$$P_{TOTAL}^{D} \approx MU_{ALU}\left[2K_{VC}\frac{1}{VL}(1+\alpha)+K_{ALU\_CTRL}^{DATA}+\sum_{i}K_{exe(i)}w_i+\right.$$
$$\left.+\alpha K_{LDST}^{DATA}+K_{VRF}(1+2\alpha)+\alpha K_{MEM\_BANKs}\right]+ \qquad (5.5)$$
$$+MU_{ALU}\frac{M}{VL}\left(K_{ALU\_CTRL}^{INSTR}+\alpha K_{LDST}^{INSTR}\right)$$

Consequently, the total dynamic energy consumed for a given workload can be obtained by Equations 5.3 and 5.5.

$$E^{D}=P_{TOTAL}^{D}t_{EXEC}\approx$$
$$\approx K_{EXEC}\left[2K_{VC}\frac{1}{VL}(1+\alpha)+K_{ALU\_CTRL}^{DATA}+\sum_{i}K_{exe(i)}w_i+\right. \qquad (5.6)$$
$$\left.+\alpha K_{LDST}^{DATA}+K_{VRF}(1+2\alpha)+\alpha K_{MEM\_BANKs}\right]$$
$$+K_{EXEC}\frac{M}{VL}\left(K_{ALU\_CTRL}^{INSTR}+\alpha K_{LDST}^{INSTR}\right)$$

Assuming a given kernel application with fixed *VL* and *M=L*, the following conclusions can be made:

(i) The first part in the right hand term is constant.

(ii) The second part increases linearly with M. However, M has a small impact on dynamic energy because the scaling factor is small; especially if VL is high (for example, for FIR, VL=64, loop unrolled three times: $E^{D}\approx K_{EXEC}(324.5+M\cdot 1.03)$). This conclusion is intuitive: the number of instructions for a given workload is the same but the number of lane controllers that process this stream increases with M.

(iii) For the same VP architecture and for the same kernel application, it is pertinent to assume that the dynamic energy will be almost constant for any number of lanes.

Some deviations from the model could be discussed also:

(iv) The dynamic power of the crossbar and instruction buses may not scale linearly with crossbar size. As the number of masters and slaves for an all-to-all switch increases, the wires are longer and dissipate more energy per atomic transfer. This will produce a model for energy that is not constant but it is rather a function of M^{β}, $\beta > 0$.

(v) Kernel applications with conditional execution may exhibit different power profiles for different lanes. However, if the condition flags are randomly distributed across lanes (and vectors), the energy consumption for each lane will be the same on average. Conditional execution is exercised by the FFT kernels. As a consequence, as also observed in Figure 5.5 (b), the actual dynamic power of VRF is under the linear curve (circled scenarios).

(vi) The same conclusion as in (ii) but for cases where any VL (*a*VL) is not a multiple of number of lanes.

5.3 Static Power Estimation

Static power measurements on the FPGA require adjustments to account for the fact that different configurations of the VP design do not fully utilize the FPGA device. Accordingly, the static power consumption (also called quiescent power) reported by the Xilinx XPower is scaled by the fraction of the core FPGA resources used by the design. Table 5.3 shows the static power breakdown for a 8×8 VP design implemented on the XC6VLX130t device. On top of leakage power, there is a dynamic power component produced by FPGA's clock tree that cannot be clock gated by the Xilinx synthesis tools. This component is constant and is consumed independent of the VP activity (idle dynamic power).

Table 5.3 Static Power Breakdown for a 8×8 VP Design on XC6VLX130t Device (Internal Supply Voltage Relative to Ground is 1V; Junction Temperature is 85° C)

Component	Static Power (mW)
Total XC6VLX130t	1544
Entire VP, VM, MC and VC	270
VP Lane	25
VM, MC and VC	70

Similarly with leakage power case, VP clock tree power is scaled by the fraction of the core FPGA resources used by the design. Quantitatively, it counts for less than 10% of the leakage power consumption, and is incorporated in Table 5.3 numbers for static power.

5.4 Energy Performance Trade-off Preliminaries

In order to analyze, model and implement an efficient lane-based flexible VP, the spaces that may represent potential opportunities for Energy-Performance gains have to be identified as accurately as possible.

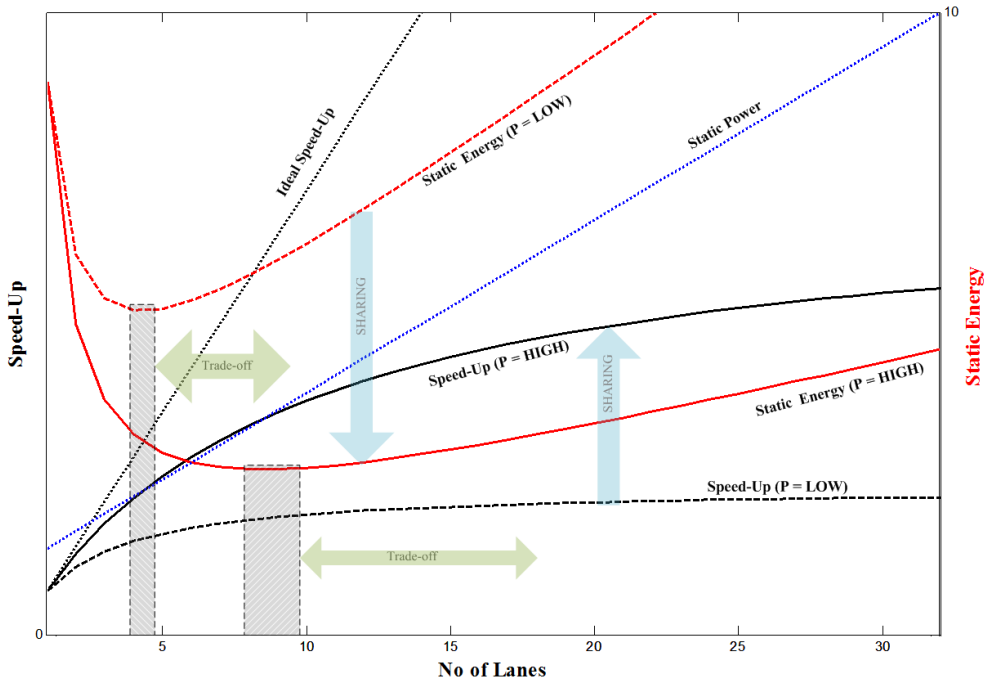


Figure 5.7 Graphing performance-energy scalability opportunities for a lane-based VP system. The speed-up is represented by black lines and the static energy by red lines. The static power is shown in a dotted blue line and its non-zero offset for zero lanes is due to VP hardware components that do not scale (VC, MC, VM, buses, etc.). The vertical axes for the speed-up and static energy are shown in the linear scale.

Figure 5.7 plots in a single graph the performance and static energy scalability for a lane-based VP assuming two distinct cases of data parallelism; i.e., low and high data-level parallelism (DLP). Additionally, the static power is shown in a dotted blue line. Speed-up curves are drawn according to Amdhal's Law that shows the upper bound on the performance for a VP and a given level of DLP. Besides the leakage power, the static power may also reflect dynamic power oriented components that are independent of the workload (i.e., they are present independent of the VP activity). Usually, these dynamic power components are mainly resulting from the clock distribution tree that cannot be gated and the clock gating components. Static power is consumed when the VP is idle; that is, when no operation is performed. In order to compute the total power budget, the static power is added to the dynamic power that is consumed when some workload exists in the VP. As discussed in Section 5.2, for a given kernel, the dynamic energy consumed to perform a given task (fixed number of operations) is constant or almost constant for any number of VP lanes, and it is not shown in Figure 5.7.

As depicted in Figure 5.7, three major opportunities can exist in optimization studies:

(i) The static energy impact could be minimized by increasing the DLP (as per the blue arrows). This approach was taken in Chapter 4 and the results show that the overall performance is increased while the energy can be reduced by following any of these steps, or their combinations, that can effectively increase the overall parallelism: (a) increasing ILP - loop unrolling; (b) increasing DLP - vector length; or (c) increasing TLP – sharing (governed by Gustafson's law).

(ii) The static energy, and thus the total energy, could be minimized by adjusting the number of VP lanes (as per the search spaces represented by the gray boxes). It can be seen that the optimal number of lanes varies with the DLP, and this number increases as the parallelism increases. For cases that have a low

DLP, it becomes imperative to *tune* the VP system in order to reach the optimal number of lanes; otherwise, a non-optimal point will increase substantially the static energy. Additionally, the offset of the static power caused by VP hardware components that do not scale (VC, MC, VM, buses, etc.) influences the optimal number of lanes. As a consequence, when more VP components that do not scale are added (the same components are present for any VP size), the optimal point will move towards the right; i.e., the minimum energy will be achieved for a higher number of lanes.

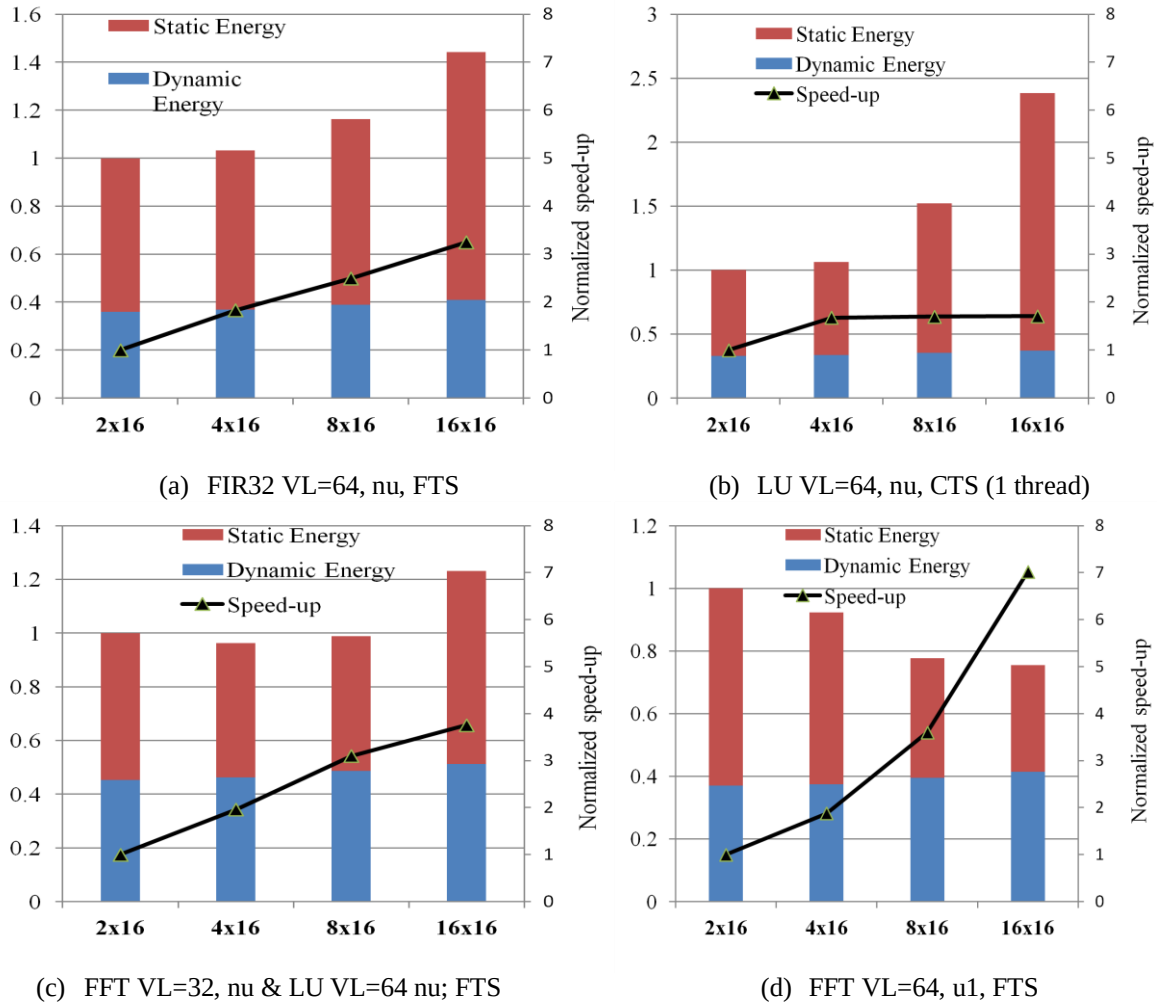
(iii) A trade-off mechanism can be developed to adjust the VP size on the right side of an optimal point (search spaces represented by the green arrows). This will give priority to higher performance at the expense of additional static energy consumption. It must be noted also that, the static energy penalty is lower for applications with high DLP due to shorter execution times (solid red line above the respective green box) as compared with applications having low DLP (dashed red line above the respective green box). This is a very good reason to enforce optimization opportunity (i), whenever possible. Additionally, VP or SIMD systems should be forbidden to enter in the regions located to the left of the green boxes where the energy and performance penalties are both very substantial.

The last two opportunities are discussed in more detail and also tackled in Chapter 6. However, in this section some performance-energy figures are presented for the FPGA implementation in order to justify the analytical discussion in this section.

As deduced from Equation 5.6, for a given kernel application the dynamic energy model shows almost constant behavior for any VP model with M lanes and L memory banks. For this section, the original 16×16 configuration (16 lanes and 16 memory banks) is configured in any of the following combinations: 2 lanes × 16 memory banks, 4×16, 8×16, and 16×16. The Hardware update is done by disabling the write enable signal for the lane instruction queues of inactive lanes. Also, the configuration fields are

appropriately configured in each lane as per Section 2.1. Also, the static power corresponding to the inactive lanes can be removed from the total power budget in Equation 5.7. The Vector Controllers, Crossbar and memory banks are always active.

$$P_{ST}^{M} = P_{ST}^{16} - (16 - M)P_{ST}^{LANE} \quad M \in \{2,4,8,16\} \tag{5.7}$$



(a) FIR32 VL=64, nu, FTS

(b) LU VL=64, nu, CTS (1 thread)

(c) FFT VL=32, nu & LU VL=64 nu; FTS

(d) FFT VL=64, u1, FTS

Figure 5.8 Normalized energy consumption for a workload of 10K FP operations for various kernels (normalization is with respect to the 2x16 configuration; nu - no loop unrolling, u1- loop unrolled once).

Current commercial FPGA technologies do not support power gating or driving a part of the FPGA fabric to a low consumption standby power state. However, over the last several years, various power gating techniques for FPGAs have been proposed to

mitigate the impact of subthreshold and gate leakage currents. In [Rahman et al., 2006] a design methodology to determine the granularity of power gating for FPGAs is presented. However, the sleep transistors are controlled in the FPGA configuration space, allowing power gating only during the bit-stream generation (statically controlled) or by using partial reconfiguration. More flexible solutions have been proposed in [Ishihara et al., 2011] where the logic clusters can be selectively powered-down at run-time either autonomously or dynamically from the FPGA logic itself [Bsoul et al., 2010].

Figure 5.8 shows the normalized total energy consumption for various application kernels running a fixed workload of 10,000 SPFP operations. As can be seen, the minimal energy is obtained at different configurations. At one extreme, as Figure 5.8 (b) displays, the minimum total energy for the LU decomposition kernel is provided by the 2×16 configuration (close to the 4×16) configuration and the 16×16 configuration consumes 2.4 times more energy. The reason is that, as per Section 4.3 and Figure 4.6, the LU performance remains the same starting with the 8×16 configuration regardless the number of lanes added to the system. More static power is consumed due to additional lanes but the execution time remains the same. As a reminder, LU exhibits this performance behavior due to stalls caused by the scalar division and memory accesses. Therefore, for these types of applications that provide low utilization and no (or low – see Figure 5.8(a)) performance scalability, the optimal number of lanes that minimizes the total energy will be small. On the other hand, if the kernels scale well with number of lanes, as is the case of FFT with VL=64 in Figure 5.8 (c), the total energy drops as the number of active lanes is increased. Therefore, it becomes imperative to provide a

methodology to change at runtime the size of the Vector Processor as the workload changes dynamically.

CHAPTER 6

PERFORMANCE-ENERGY OPTIMIZATIONS FOR SHARED VECTOR ACCELERATOR IN MULTICORES

For the majority of applications that use a dedicated vector coprocessor per processor core, its resources are not highly utilized due to the lack of sustained sequences of vector instructions, and/or the presence of limited data-level parallelism. Also, under low coprocessor utilization static power dominates the total energy consumption. Based on these observations, this chapter targets high resource utilization for vector coprocessors associated with multicores in order to enhance the performance, while also reducing the impact of static energy consumption. Chapter 2 proposes a robust design framework for vector coprocessor sharing in multicore environments that targets these objectives. This chapter further enhance the vector coprocessor sharing framework by proposing two power gating (PG) techniques that can dynamically control the width of the shared coprocessor based on the utilization of vector lanes [Beldianu and Ziavras, 2012]. Results for several floating-point intensive benchmarks run on an FPGA-based prototype show that the proposed PG techniques reduce the energy needs by 30-35% with negligible performance reduction as compared to a multicore with the same amount of hardware resources where, however, each core is attached to its own dedicated vector coprocessor. Additionally, a performance-energy tradeoff mechanism is introduced, which gives priority to performance gains at the expense of higher energy consumption; the results show a performance gain of 18% with an increase in the energy consumption by 13-22 %.

Related work is discussed in Section 6.1. Energy minimization is discussed in Section 6.2. It is followed in Section 6.3 by a simulation model and a description of the experimental setup. Experimental results are presented in Section 6.4 and an energy-performance trade-off mechanism is presented in Section 6.5. Conclusions follow in Section 6.6.

6.1 Related Work

Static power will become a larger source of consumption in future technologies due to reduced feature sizes and increased transistor counts [Keating et al., 2007]. Starting with the 45nm technology, leakage power consumption catches up with, or surpasses, dynamic power consumption. However, the sustained performance does not normally follow this upward trend, primarily because of decreases in the average transistor utilization from load imbalances that become preeminent at finer resource levels. A new limit on scaling will eventually arise creating a transistor utility economics wall. A study employs device, core and CMP (chip multiprocessor) scaling models to show that, regardless of the multicore organization, a large area on future chips will have to be frequently powered down [Esmaeilzadeh et al., 2011]. At 22 nm, 21% of a chip on the average should be powered down; it grows to more than 50% at 8 nm. Moreover, according to this unified model an average speed-up of just eight will be possible in the next decade for common parallel workloads; it will result in a substantial gap (up to 24) between the expected (as per Moore's Law) and actual performance figures.

Leakage power has increasingly become a substantial component of the total energy consumption of silicon chips. Studies have shown that the leakage power is responsible for more than 40% of the overall power dissipation for the 90nm technology

node [Kao et al., 2002] and can exceed the 50% figure at 65nm and below [Kim et al., 2003; Scogland et al., 2010]. Besides CMOS process solutions, various techniques have been proposed to reduce the leakage power. These techniques either trade-off increased performance for reduced static energy consumption, or completely turn-off circuit components by gating the ground or the voltage supply (the latter approach is called power gating). Dynamic Voltage and Frequency Scaling (DVFS) is a power consumption limiting technique that reduces the clock frequency and/or the supply voltage [Hong et al., 1999]. Since the dynamic power dissipation is proportional to the operating frequency and the square of the supplied voltage, the reduction in dynamic power dissipation can then become very substantial. However, DVFS becomes less beneficial for leakage dominant components, such as SRAM caches or large register files [Wang and Mishra, 2011]; therefore, its effectiveness with future multicores is highly questionable. Multiple threshold CMOS circuits can be used to deal with the leakage problem in low voltage, low power and high performance applications. Several such CMOS circuit design techniques have been introduced, such as multi-threshold voltage CMOS [Anis et al., 2003] and variable threshold CMOS [Hiramoto and Takamiya, 2000]. This work does not use DVFS or techniques involving multiple thresholds to lower or trade energy consumption but they could still be complementary to proposed schemes for even higher gains in energy consumption.

Additional elaboration on power gating is pertinent to work presented in this thesis. Power gating was initially proposed to reduce the static power of static RAM (SRAM) cells in cache memories. A fine-grain technique for an embedded processor uses a sleep instruction to power gate individual functional units [Roy et al., 2009]. When an

instruction subsequently decoded needs to use a sleepy functional unit, the latter is waken up to become ready before the instruction reaches the execute stage. An ultrafine-grain power gating scheme for on-chip routers individually controls the power supply to each router component (e.g., virtual-channel buffer, virtual-channel multiplexer, crossbar multiplexer and output latch) based on the present workload [Matsutani et al., 2011]. However, as the granularity becomes too fine, the power gating technique becomes less effective due to the large overheads introduced by the control circuitry and the power supply network. A coarse-grain per-core power gating architecture for multicore processors allows software to turn on and off individual cores as the utilization varies for datacenter workloads [Leverich et al., 2009].

Some work has been done on finding the optimal number of active cores that minimizes the energy consumption for a given task. A theoretical study determines the optimal number of cores that minimizes the energy consumed by a parallel algorithm on a shared-memory architecture [Korthikanti and Agha, 2010]. The results suggest a divergence of power and performance scalability for parallel algorithms. Nevertheless, even if the optimal number of cores is derived for a few parallel applications, no runtime framework capable of adjusting the number of active cores for a dynamic workload is presented. An analytical model involving energy and performance for a chip multiprocessor finds the number of cores that maximize the power savings while meeting a given level of performance [Li and Martinez, 2005]. It also shows that the power savings increase with more processors, up to a point where any savings stagnate and eventually recede. Other theoretical works on energy minimization for many-core systems could be found in [Woo and Lee, 2008; Cho and Melhem, 2008]. An instruction-

level energy prediction mechanism [Wang and Ranganathan, 2011] estimates statically the number of active streaming multiprocessors (SMs) that minimize the dynamic energy for CUDA workloads on an Nvidia GPU [Nvidia CUDA, 2011]. The static power is completely ignored and the optimization framework is based on the number of active SMs rather than on the number of active CUDA cores within an SM. These rigid architectures cannot tolerate efficiently dynamic application environments with many cores that may require the runtime adjustment of assigned vector resources in order to operate at desired energy/performance levels that change frequently.

In contrast to all of these works, the efforts rely on information which is extracted at static time or gathered at runtime by embedded hardware counters. This information is used to dynamically change the number of active lanes in a shared vector coprocessor in order to minimize the overall impact of static power. The dynamic energy consumption depends basically on the application itself, therefore every effort is simultaneously made to maximize the utilization of the resources within the active lanes. Static information can be extracted for the application using standard program profilers embedded in software development environments for the scalar cores; e.g., GNU gprof, Intel VTune Amplifier XE, etc. On the other hand, the hardware profilers use special registers to monitor the utilization of instruction paths within the vector lanes; they have very low cost and need very little information to extract the utilization of vector lanes (e.g., vector length, number of active lanes and, optionally, operation type to monitor).

6.2 Total Energy Minimization

From Section 5.4 it can be concluded that the optimal number of lanes that minimizes the total energy is small for applications with low performance scalability. On the other hand,

if the performance of a kernel scales well with the number of lanes, as is the case for FFT with VL=64 in Figure 5.7c, then the total energy drops as the number of active lanes is increased. Therefore, it becomes imperative to provide a methodology to change at runtime the number of active lanes in the VP as the workload changes dynamically in order to minimize the overall energy consumption without inadvertently affecting the performance. This is the ultimate objective of this work.

By combining Equations 5.3, 5.6 and 5.7, the total energy consumption of a VP with *M* active lanes and *L* memory banks is given by Equation 7.

$$E_{TOTAL} = t_{exec}\left[P_D + P_{ST}^{L} - (L-M)\left(P_{ST}^{LANE} - P_{OFF}^{LANE}\right)\right] = E^{D} + K_{\text{ker}nel}\frac{P_{ST}^{L} - (L-M)\left(P_{ST}^{LANE} - P_{OFF}^{LANE}\right)}{MU_{ALU}^{M}} \tag{6.1}$$

where U_{ALU}^{M} is the ALU utilization for the *M*×*L* VP configuration.

It is safe to assume from Equation 5.6 that the dynamic energy is almost constant independent of *M*. This should be expected of a good coprocessor design since the dynamic energy consumption will then rely almost exclusively on the actual amount and type of work in the application itself. Therefore, minimizing the total energy implies the minimization of the static energy as a function of *M*. The optimal value of *M* is then given by:

$$M_{\min} = \arg\min_{M\in\Phi}\left\{\frac{P_{ST}^{L} - (L-M)\left(P_{ST}^{LANE} - P_{OFF}^{LANE}\right)}{MU_{ALU}^{M}}\right\} \tag{6.2}$$

where Φ is the set of permissible values for *M*.

6.2.1 Dynamic Power Gating with Static Information (DPGS)

Each time a VP event occurs (i.e., a scalar core requests or releases VP resources that will change the workload profile), apriori information is used to compute the optimal number of lanes that minimizes the energy consumption for the requested or active task(s). Since the static power variables from Equation 6.2 are fixed for a given VP architecture, the only information required is the utilization U_{ALU}^{M} for all permissible values of M. A simple and very efficient, but not necessarily highly accurate way, of acquiring this information at static time will be to employ offline simulations of single kernel executions and combinations involving any pair of kernels that the VP may have to simultaneously run in the future (since two vector threads arriving from the two cores may have to be run simultaneously). Static information can be extracted for the application using standard program profilers embedded in software development environments for the scalar cores; e.g., GNU gprof, Intel VTune Amplifier XE, etc. A more effective way is described later in the next section. To speed-up this process, a look-up table can be created to contain the optimum value of M for every possible pair of kernels (θ_i, θ_j), where $i, j \in \Theta$ and Θ is the set of all possible kernels that can be run simultaneously on the VP, including also the idle kernel.

Figure 6.1 presents hardware extensions to the VP architecture that can support software controlled DPGS power gating. The hardware support consists of a power gate sequencer to be configured by software and other specific power gate elements (sleep transistors and isolation cells). This software can be implemented in the form of operating system (OS) routines for Power Management (OSPM) running on one of the processors in the multicore environment or can be realized by a dedicated Power Control Unit (PCU;

e.g., Intel7 Nehalem). Figure 6.2 shows the details of these OS or PCU-driven interrupt routines, along with the relevant routine run by cores that acquire VP resources.

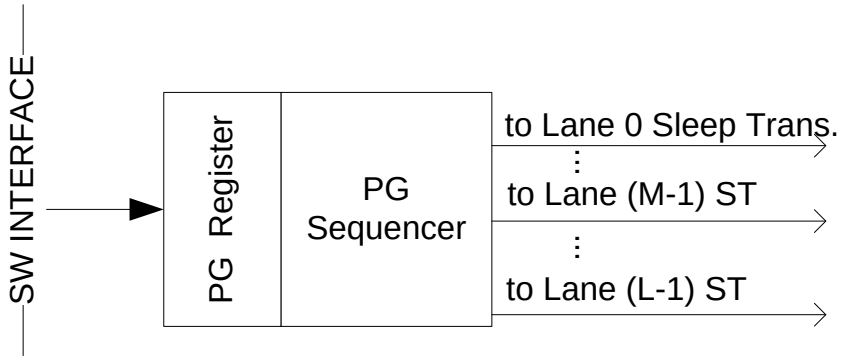


Figure 6.1 Hardware support for DPGS scheme. In DPGS, the Power Gate (PG) Register is configured by software. ST: Sleep Transistor (Header or Footer).

OS or PCU-driven interrupt routine run upon a VP event (i.e., any scalar core releases or acquires the VP)

1. *Based on the active kernels running on the VP, new kernel request, utilization table and Equation 8, compute the optimum number of lanes M for the VP.*
2. *If the state of the VP doesn't need any change, then EXIT; else, go to step 3.*
3. *Stop the Scheduler to receive any new VP acquiring requests.*
4. *Assert a software interrupt to the scalar CPUs that have VP resources acquired.*
5. *Wait for ACK signals from all the CPUs.*
6. *Reconfigure the PG register.*
7. *Enable the Scheduler to receive new requests and EXIT.*

Scalar CPU interrupt routine in response to an OS/PCU-initiated change

1. *Finish the inner loop of the kernel, and save the results or dirty vector registers in the memory.*
2. *Release VP resources (VP_REL).*
3. *Send an ACK signal to OS/PCU.*
4. *Attempt to acquire VP resources (VP_REQ) and wait until the Scheduler acknowledges the request.*
5. *Restore the saved vector registers and EXIT.*

Figure 6.2 Interrupt routines to handle DPGS.

The main disadvantage of this scheme is the fact that obtaining at static time the combined utilization of VP units for any possible pair of simultaneously running vector kernels is impractical and often inaccurate since the kernels may start executing with previously unknown phase delays. Also, it assumes that all possible vector kernels that

may be encountered at run time are known apriori. This assumption does not allow the power efficient implementation of previously unknown vector-oriented tasks. An option is to approximate the overall utilization of VP lanes with a function that involves the unit utilizations of individual kernels obtained when run by themselves without any interference from other kernels. However, due to the intrinsic behaviors of individual kernels and the aforementioned phase delays, finding such a generic function independent of the involved kernels becomes a Herculean task. The second scheme, namely APGP, eliminates the need of DPGS to estimate kernel utilization information at static time by incorporating hardware profilers that can extract accurate utilization information for vector lane units at run time. The extra hardware needed for the profilers and the associated control circuit is minimal.

6.2.2 Adaptive Power Gating with Profiled Information (APGP)

Using embedded hardware profilers at run time, the utilization of individual VP units can be measured precisely in a perpetual effort to minimize the energy consumption. A decision can then be made by specialized control hardware in order to determine if the current number of active lanes should be changed or not. The following theorem can be used to find the optimal number of lanes that minimizes the energy consumption at run time based on the instantaneous utilization of the VP units.

Theorem 1. If the total energy consumption for a given application kernel in the *M*-lane VP configuration is smaller than the total energy consumption in the *N*-lane configuration, then the following inequality holds:

$$\frac{U_{ALU}^{M}}{U_{ALU}^{N}} > RTh_{M/N} \tag{6.3}$$

where U_{ALU}^{M} and U_{ALU}^{N} are the ALU utilizations of the kernel for the *M*-lane and *N*-lane configurations, respectively; $RTh_{M/N}$ is a constant independent of the application running on the VP, and depends on *M* and *N*. Additionally, if *M*>*N* then $RTh_{M/N} \leq 1$.

***Proof*:** From $E_T^M < E_T^N$ and Equation 6.1, the following inequality follows:

$$E_D^M + K_{\text{ker}nel} \frac{P_{ST}^L - (L-M)\left(P_{ST}^{LANE} - P_{OFF}^{LANE}\right)}{MU_{ALU}^M} < E_D^N + K_{\text{ker}nel} \frac{P_{ST}^L - (L-N)\left(P_{ST}^{LANE} - P_{OFF}^{LANE}\right)}{NU_{ALU}^N}$$

According with Equation 5.6 and conclusions drawn in Section 5.2, i.e., $E_D^M \cong E_D^N$, the above inequality then becomes:

$$\frac{U_{ALU}^M}{U_{ALU}^N} > \frac{N}{M} \frac{P_{ST}^L - (L-M)\left(P_{ST}^{LANE} - P_{OFF}^{LANE}\right)}{P_{ST}^L - (L-N)\left(P_{ST}^{LANE} - P_{OFF}^{LANE}\right)} \tag{6.4}$$

where the right hand term is $RTh_{M/N}$. For *M*>*N*, $U_{ALU}^M \leq U_{ALU}^N$ since the lane ALU utilization will decrease or, in the best case stay constant, when the number of VP lanes increases. Thus, $RTh_{M/N} \leq 1$. Perfect performance scalability is reached when $U_{ALU}^M = U_{ALU}^N$.■

After a VP event, in order to evaluate the inequality in Equation 6.3 the profiled unit utilizations for at least two VP configurations are required. To accomplish this task, this work proposes a dynamic scheme in which the state of the VP is changed successively in the right direction (i.e., increasing or decreasing number of lanes) until the optimum VP state is reached. Since for most of the benchmark scenarios the minimum energy consumption results for $M \in \{4,8,16\}$, the runtime framework is

developed based on the four VP states shown in the set: {all lanes are gated so the VP is idle ($0L$), 4 lanes active ($4L$), 8 lanes active ($8L$), 16 lanes active ($16L$)}.

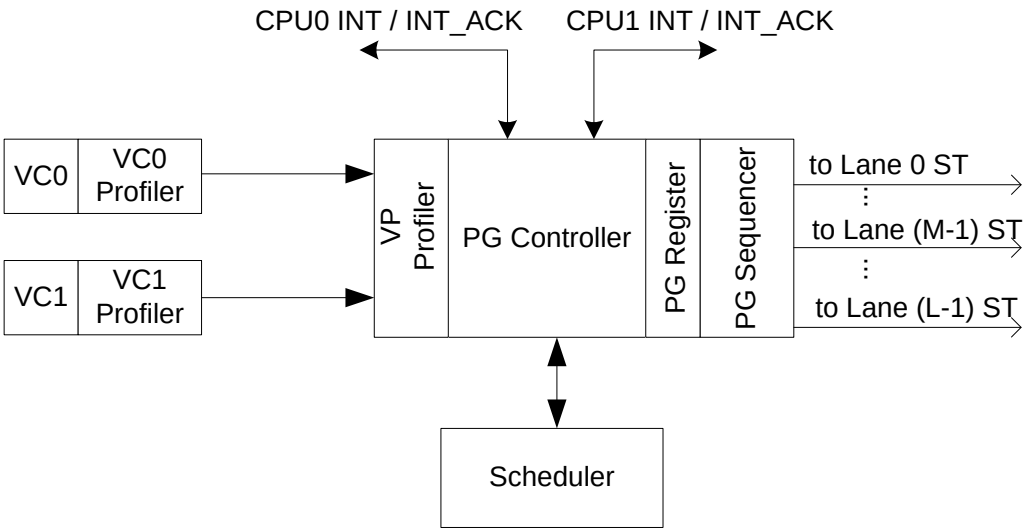


Figure 6.3 Hardware support for APGP scheme. In APGP, the PG Register is configured by the PG Controller. The VP Profiler aggregates the utilizations from both VCs. ST: Sleep Transistor (Header or Footer).

Figure 6.3 shows the hardware (HW) components that support APGP. Each VP profiler is attached to a VC, and monitors the ALU and LDST utilizations by the respective vector kernel. It captures the average ALU utilization based on the instruction stream that flows through the VC over a given time window, as per Equation 5.2. The implementation is simple, consisting of an IIR (infinite impulse response) filter with a sample rate of 256 cycles according to $U_{ALU}^{next} = U_{ALU}^{prev} + U_n^{256} - U_{n-4}^{256}$, where U_n^{256} and U_{n-4}^{256} are the cumulative numbers of operations in the lane's ALU in the last 256 cycles and in the [1024, 1279] cycle frame, respectively.

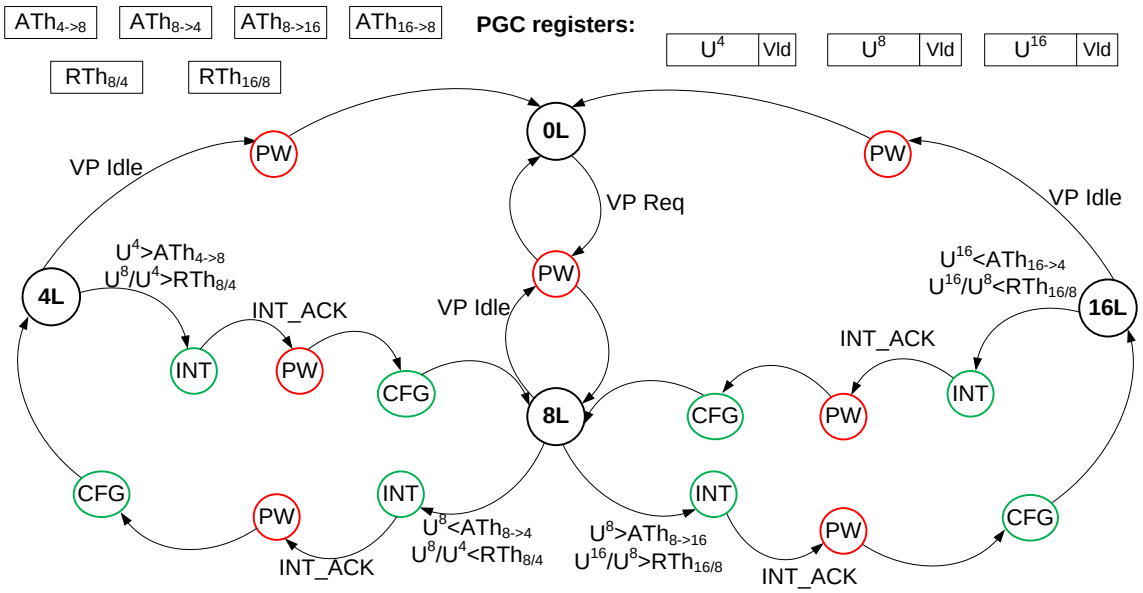


Figure 6.4 PG Controller (PGC) state machine and PGC registers for state transitions under APGP. INT, PW and CFG are transitional VP (i.e., non-operating) states. 4L, 8L and 16L are stable VP operating states that represent the 4-, 8- and 16-lane VP configurations. *M*L is a PGC state with *M* active lanes, $M \in \{0,4,8,16\}$; INT is a PGC state where the PGC asserts an interrupt and waits for an Interrupt Acknowledge (INT_ACK); PW is a PGC state where some of the VP lanes are powered-up/down; CFG is a PGC state where the Scheduler is reconfigured to a new VP state. Threshold registers are fixed during runs and utilization registers are updated for every profile window. The registers store 8-bit integers. The Vld bit is used to show that the utilization register U^M, with *M*= 4, 8 or 16, for the *M*-lane VP configuration does not contain an updated value.

Simulations show that a profile window of 1024 clock cycles with a sample rate of 256 cycles gives an accurate estimation of the average utilization for all kernels presented in Section 3.2. The HW PG Controller aggregates the utilizations produced by both threads (using the VP profilers) and implements the PG Controller state machine shown in Figure 6.4. The proposed scheme is based on two types of thresholds: (i) the absolute threshold $ATh_{M->N}$ which is used when the ratio U^M_{ALU} / U^N_{ALU} is not available for the current kernel combination and $M \rightarrow N$ represents the transition from the *M*-lane to the *N*-lane VP configuration, and (ii) the relative threshold $RTh_{M/N}$ computed in Equation 10. The relative threshold $RTh_{M/N}$ is used for comparison when the utilization for both configurations with *M* and *N* lanes is profiled and stored in appropriate registers.

Absolute thresholds are empirically chosen such that, for a given ALU utilization, the probability that the current VP configuration state will be kept is minimum if a VP configuration with lower energy consumption state exists. In other words, the absolute threshold will enable the PG Controller to initiate a state transition if there is a probability greater than zero that the current state does not yield the minimum consumption. For example, $ATh_{8->16}$ is chosen such that the following condition is true for the probability $P(U_{ALU}^{8} < ATh_{8->16} | 16L \min energy) \cong 0$. $RT_{M/N}$ is less than one since it is a ratio of ALU utilizations with *M* and *N* lanes, respectively, and *M*>*N*. Also, the upper bound on *ATh* is one since it represents a utilization figure. Besides the above mentioned thresholds, the PG Controller contains the utilization registers $U_{ALU}^{M}, M \in \{4,8,16\}$, (one for each VP configuration) which are updated at run time by the profilers.

The proposed scheme for APGP power gating works as follows. After a VP request or release event that may potentially change the utilization figures and, thus, the optimum configuration, the utilization registers are reinitialized. The Vld bit in Figure 8 is used to show that the utilization register U^{M}, with *M*= 4, 8 or 16, for the *M*-lane VP configuration does not contain an updated value. If the VP is initially idle (0L), the PG Controller (PGC) will power up eight lanes and will enter the 8L VP state. The reason to move the PGC from the 0L directly to the 8L configuration (that is, bypassing the 4-lane configuration) is that, statistically, 8L has the highest probability to be the optimum energy state for the set of scenarios used in the experiments. The VP will use data from at least a single profile window in order to update the utilization for this configuration. If one of the inequalities based on the absolute threshold is satisfied, the controller will initiate a transition to another state. A profile window is the time window in clock cycles

for which the utilization of lane's ALU is monitored. After each profile window, the utilization register corresponding to the current state is updated.

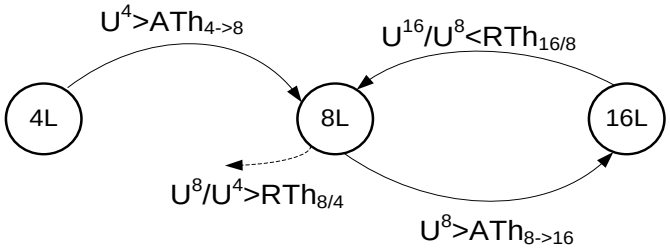


Figure 6.5 Example of state transitions upon a VP event.

A transition between two stable VP operating states involves the following steps and three transitional VP non-operating states:

1. ***INT state****: stop the Scheduler to receive any new VP acquire requests and send a hardware interrupt to the scalar CPUs that have VP resources acquired.*
2. ***PW state****: after ACKs from all CPUs are received, configure the PG Sequencer for a new VP power state.*
3. ***CFG state****: reconfigure the Scheduler with the new number of lanes and enable it to acknowledge new VP acquire requests.*

The CPUs run the interrupt routine in Figure 7. In the new state, the utilization register will be updated after a profile window; if one of the inequalities is met, it will initiate a new transition. As discussed earlier, the inequality may be based on the relative threshold if the ratio $U_{ALU}^{M} / U_{ALU}^{N}$ is available; otherwise, it will rely on the absolute threshold. Figure 6.5 shows an example of state transitions upon a VP event. Initially the VP is in the 4L state and will move to the 8L state because, after a full profile window, the utilization is greater than the absolute threshold $ATh_{4->8}$. Subsequently, the VP state will transit to 16L and will then return to the 8L state due to relative threshold inequality. According to the state machine in Figure 8 up to three transitions are necessary to reach the minimum energy consumption state. Therefore, in order to avoid multiple transitions

that will increase the time and energy overheads, after each VP event a maximum of three state transitions are allowed. The resources consumed by the HW profilers and the PGC account for less than 1% of the total resources occupied by the VP. Also, since the PGC events are scarce, simulations with different scenarios showed that the dynamic power consumption of the PGC is insignificant as compared to the VP's dynamic power.

6.3 Simulation Model and Experimental Setup

6.3.1 Simulation Model

In order to prove the benefits of the proposed energy-saving schemes, a simulator that models the execution of VP threads for different execution configurations is developed. The simulation model is based on performance and power figures gathered from RTL and netlist simulations, as described in Section V. The model contains the information necessary to compute the execution time and energy consumption for any combination of kernels (θ_i, θ_j) running in any possible VP state. Each kernel θ_i or combination of kernels (θ_i, θ_j) is represented by the utilization(s) $U^M(\theta_i)$, $U^M(\theta_j)$ and the total power $P^M(\theta_i, \theta_j)$ when the θ_i and θ_j kernels run on the VP, for all possible values of $M \in \{4,8,16\}$. These values are obtained after performance and power simulations as per Section V.B. The ALU utilization $U^M(\theta_i)$ is used to compute the execution time for each kernel and $P^M(\theta_i, \theta_j)$ is used to compute the energy consumption. Also, the model accounts for all the time and energy overheads incurred due the state transition processes.

Table 6.1 summarizes the time and energy overheads taken into account by the model. Since the lane implementation is almost eight times bigger than a floating-point

multiply unit in [Roy et al., 2009] which is power gated in one clock cycle, the model assumes that a VP lane will wake up in 8 clock cycles. Also, a conservative approach is considered, where one lane is powered up/down at a time by the PG Sequencer in order to avoid excessive currents in the power net. The VP components that are not powered off or power on during state transition consume static energy as usual.

Table 6.1 Time and Energy Overheads for PGC State Transition

	Time overhead (cycles)	**Energy overhead**
Call interrupt routine	20 (for MicroBlaze)	Based on actual runs
Save vector registers (M-lane configuration)	$No\_dirty\_vregs * \frac{VL}{M}$ *No_dirty_vregs* = number of vector registers that need to be saved/restored.	Time overhead × [(Dynamic power to store the vector registers) + (Static power)]
Power up (one lane at a time)	8×(No of lanes to be powered up) [Roy et al., 2009]	20×(Time overhead)×(Static power when the lane is ON) [Roy et al., 2009]
Power down (one lane at a time)	0	(8 cycles)× (Static power when the lane is ON) [Roy et al., 2009]
Acquire VP and restore the vector registers (N-lane configuration)	$10 + No\_dirty\_vregs * \frac{VL}{N}$ Startup of 10 cycles to acquire the VP. Also, cycles to restore the dirty registers.	Time overhead × [(Dynamic power to load the vector registers) + (Static power)]

6.3.2 Experimental Setup

In order to expose the VP to dynamic workloads, benchmarks composed from random threads running on the scalar cores are created. Each thread has VP busy periods (i.e., vector kernels targeting the VP) and VP idle periods, as shown in Figure 10. These are realistic scenarios since during idle periods the core is often busy either with memory transfers or executing a critical section of the program. A thread busy period is uniquely denoted by a kernel θ_i and a workload expressed in a random number of floating-point operations; a thread idle period is described in terms of a random number of VP clock cycles. Ten fundamental vector kernels were used to create these scenarios. More

specifically, two versions of each benchmark kernel in Section IV were first produced, one having relatively low ALU utilization while the other has higher ALU utilization. The workload of each kernel, which is expressed as a random number of operations, is uniformly distributed between chosen limits, in such a way that enough data exists in the Vector Memory for processing without the need for additional DMA transfers (this is valid for any present kernel). With the inclusion also of an idle kernel to this set of ten fundamental kernels, 55 unique pairs of kernels, plus 10 scenarios with a single kernel active on one core only, were produced. Table 6.2 shows the absolute and relative thresholds for APGP. Although not shown in Table 6.2, in the 8L configuration only two scenarios that do not have this state as the optimum energy state have an ALU utilization in the interval [50, 60]%.

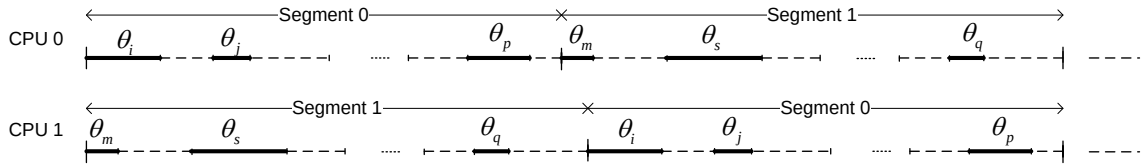


Figure 6.6 VP threads issued by each scalar core with embedded idle times. Each thread contains 1000 segment runs. Each segment contains 10,000 kernel runs. A solid line shows the time spent by the core to issue the entire code for the corresponding kernel workload.

Table 6.2 Absolute and Relative Thresholds for APGP Implementation

Threshold	Value
ATh 4->8	50%
ATh 8->16	60%
ATh 8->4	50%
ATh 16->8	72%
RTh8/4	0.6739
RTh16/8	0.7581

Simulation results were produced for the following combinations:

(i) 2 CPUs (i.e., cores), each one having exclusive access to a VP with the same fixed number of lanes (4 and 8), and all lanes of a VP are power gated during idle periods.

(ii) 2 CPUs sharing a VP with a fixed number of lanes (4, 8 or 16) under CTS; all VP lanes are power gated when both VP threads are idle.

(iii) 2 CPUs sharing a VP under CTS and DPGS (for selective per lane power gating).

(iv) 2 CPUs sharing a VP under CTS and APGP (for selective per lane gating).

(v) 2 CPUs sharing a VP with a fixed number of lanes (4, 8 or 16) under FTS; all VP lanes are power gated when both VP threads are idle.

(vi) 2 CPUs sharing a VP under FTS and DPGS (for selective per lane gating).

(vii) 2 CPUs sharing a VP under FTS and APGP (for selective per lane gating).

6.4 Experimental Results and Discussion

Figure 6.7 displays the breakdown of the normalized execution time (in reference to the first execution scenario with two scalar CPUs, each attached to its dedicated 4-lane VP) and the normalized energy consumption for the execution of the same benchmark, where the majority of vector kernels in the threads have low ALU utilization. The ratio between low and high utilization kernels in a thread is 4:1. This figure assumes idle periods between consecutive vector kernels in a thread which are expressed in VP clock cycles and are uniformly distributed in the ranges [1000, 4000], [5000, 10000] and [10000, 30000].

Some conclusions can be drawn:

(i) FTS sharing generally produces the lowest energy consumption. For a given VP sharing policy, this being CTS or FTS, the application of DPGS or APGP brings

the overall energy consumption to a minimum compared to scenarios that do not incorporate such an intelligent power gating approach.

(ii) Except for two cases, namely 2x(1cpu_8L) and 2cpu_16L_FTS, FTS sharing with DPGS or APGP minimizes the execution time as well. To their advantage, however, the latter pair of power-gating schemes also consume 30-35% and 18-25% less energy as compared to 2x(1cpu_8L) and 2cpu_16L_FTS, respectively.

(iii) Scenarios with two scalar CPUs, each with its own dedicated VP (i.e., the 2x scenarios), yield lower execution time than the CTS sharing schemes because CTS does not sustain a high utilization across all the functional units within a lane.

(iv) Usage of DPGS or APGP to CTS sharing reduces the energy consumption compared to the 2x scenarios. As the idle period between successive kernels decreases, the CTS technique becomes less effective as shown in Figures 6.8e and 6.8f; for example, just a 5% gain in energy consumption for DPGS-driven CTS with a slow down of 70% as compared to 2x(1cpu_4L).

(v) Finally, the time and energy overheads caused by state transitions are negligible, and therefore cannot be shown in Figures 6.7 to 6.9. The total time overheads have an upper bound of 0.3% of the total execution time for DPGS and 0.7% for APGP; the energy overheads are upper bounded by 0.23% of the total energy consumption for DPGS and 0.57% for APGP.

Figure 6.8 shows the normalized execution time and energy consumption for threads containing kernels with mixed utilization figures, such that the ratio between low and high utilization kernels in a thread is 1:1 (i.e., they appear with the same probability). FTS under DPGS or APGP yields the minimum energy while the performance is better than FTS with eight lanes. Figure 6.9 shows the normalized execution time and energy consumption for threads dominated by high utilization kernels, where the ratio between low and high utilization kernels is 1:4. As the number of kernels providing high

utilization increases, the portion of time spent in the 16L state increases for FTS under the DPGS and APGP schemes. As a consequence, the performance of the proposed power-gating schemes is better than the one provided by a fixed VP with eight lanes, and approaches the performance of the 16L FTS-driven configuration.

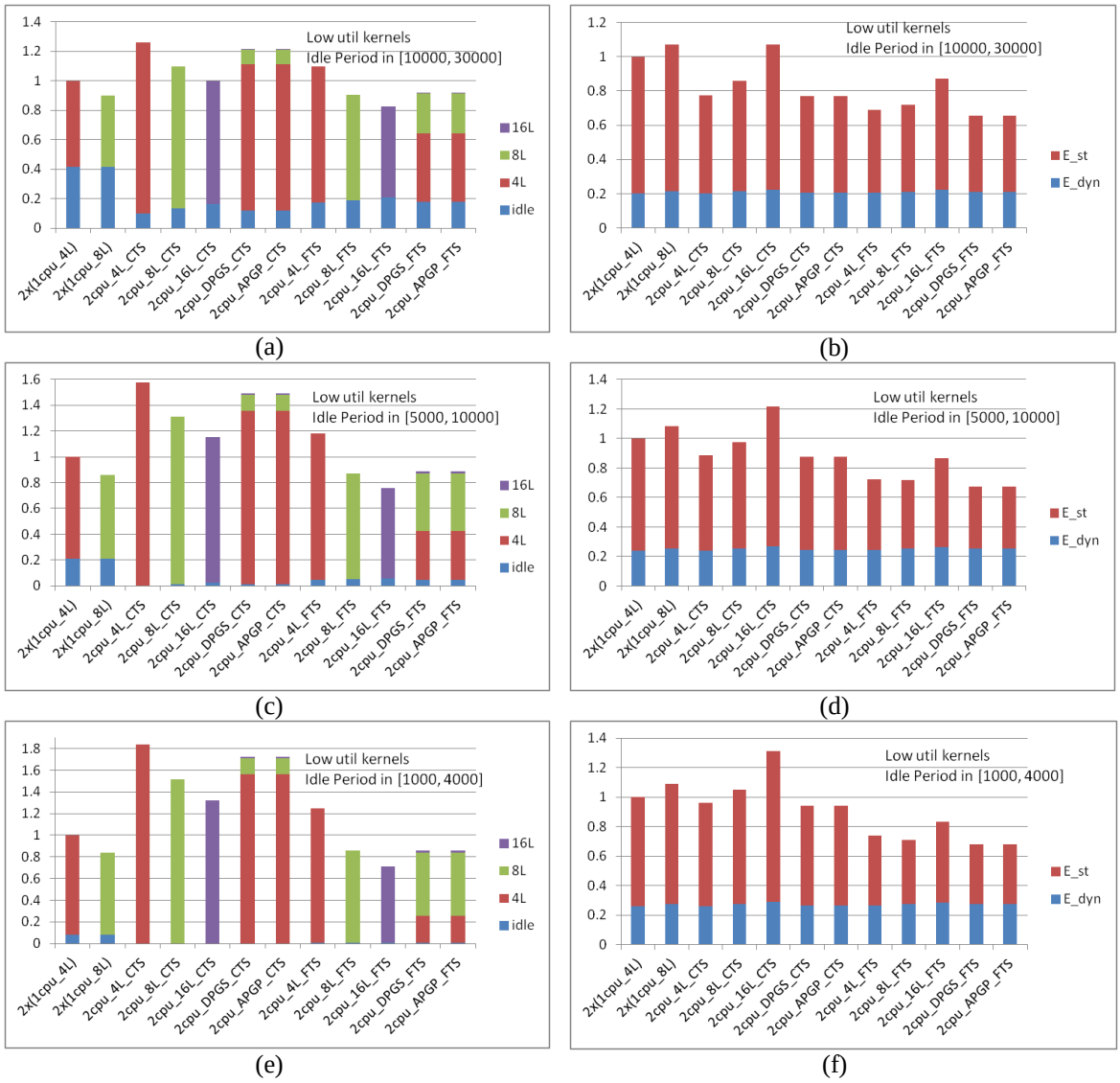


Figure 6.7 Normalized execution time (a, c, e) and normalized energy consumption (b, d, f) where the majority of kernels in a thread have low ALU utilization, for various idle periods. The ratio of low to high utilization kernels in a thread is 4:1. E_st and E_dyn are the energy consumptions due to static and dynamic activities, respectively. "2x" means two scalar CPUs of the type that follows in parentheses, such as "(1cpu_4L)" which means one CPU having a dedicated VP with four lanes. Whenever CTS or FTS shows, it implies two CPUs with VP sharing.

As expected, the energy consumption is reduced drastically with FTS and DPGS or APGP power gating as compared to all other scenarios in the figure.

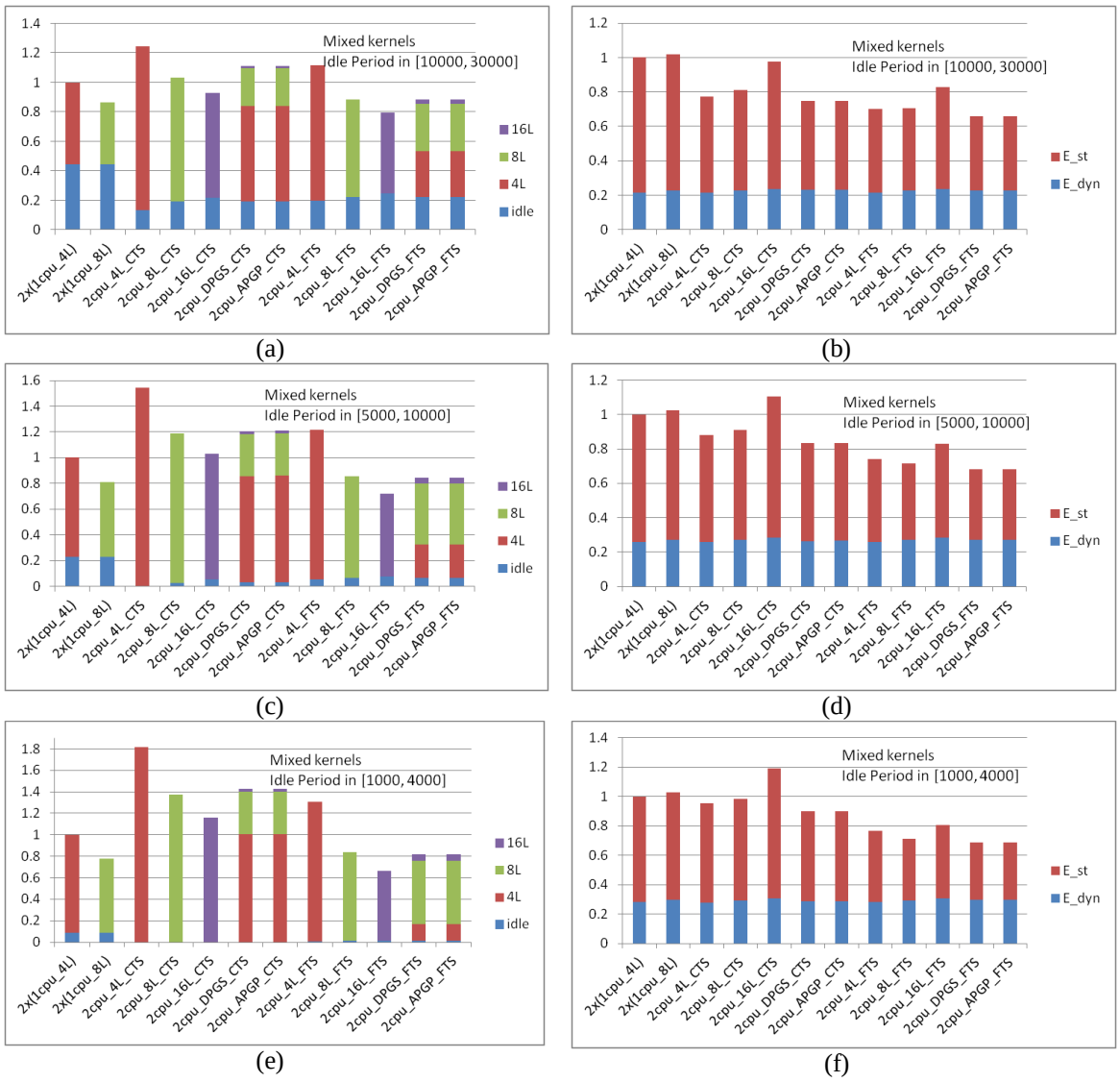


Figure 6.8 Normalized execution time (a, c, e) and normalized energy consumption (b, d, f) for threads with mixed utilization kernels, for various idle periods. The ratio of low to high utilization kernels in a thread is 1:1.

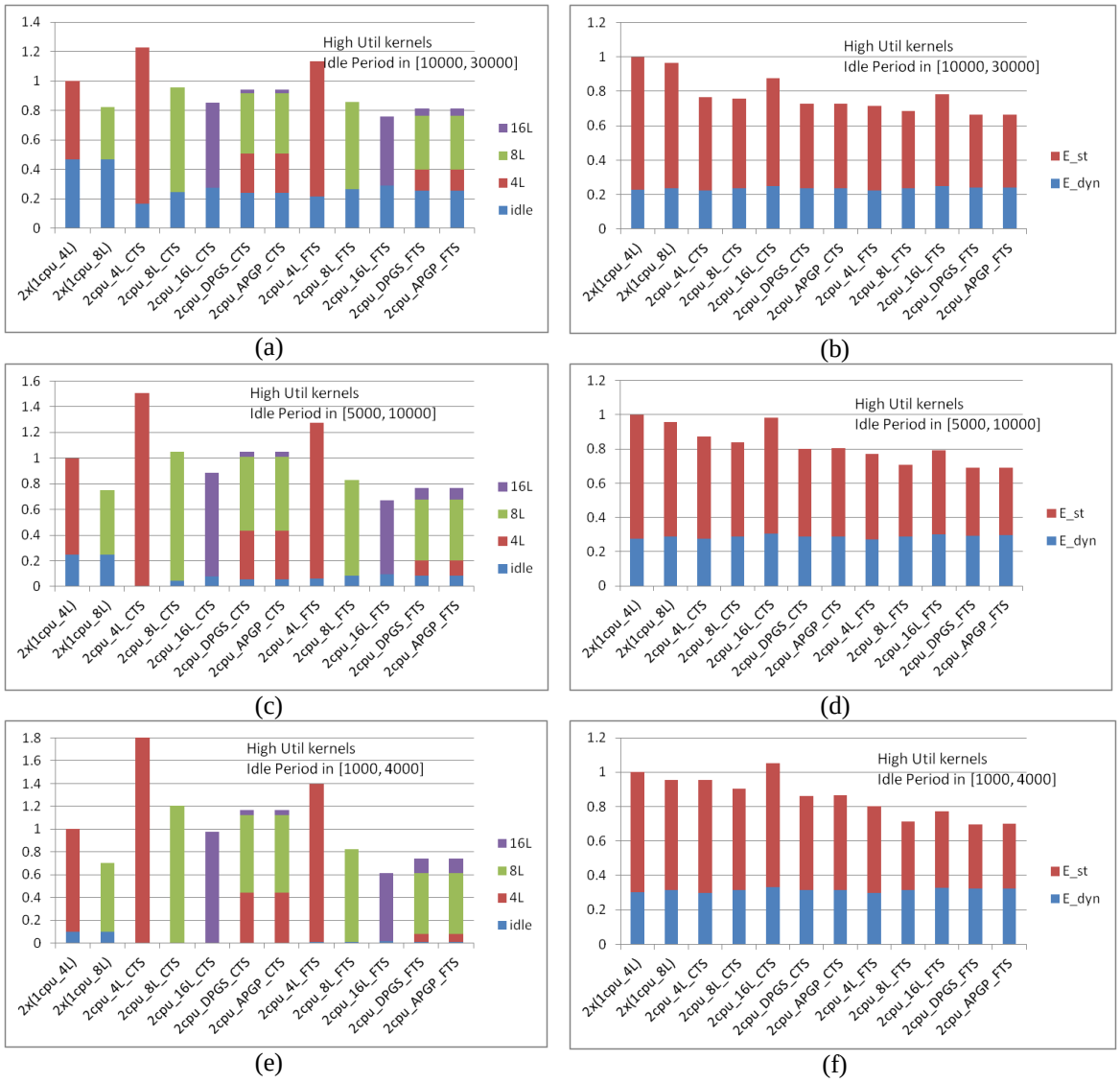


Figure 6.9 Normalized execution time (a, c, e) and normalized energy consumption (b, d, f) for threads dominated by high utilization kernels, for various idle periods. The ratio of low to high utilization kernels in a thread is 1:4.

6.5 Energy-Performance Trade-off Mechanism

The proposed power-gating approaches minimize the overall static energy consumption in all these cases that do not assume any performance constraints. Additionally, a trade-off mechanism could be used to utilize DPGS or APGP power gating schemes in order to increase the performance at the expense of an increased energy consumption. More specifically, in order to reduce the average execution time per thread, the absolute and

relative thresholds are changed in such a way that more kernels can run in a VP state that involves more active lanes.

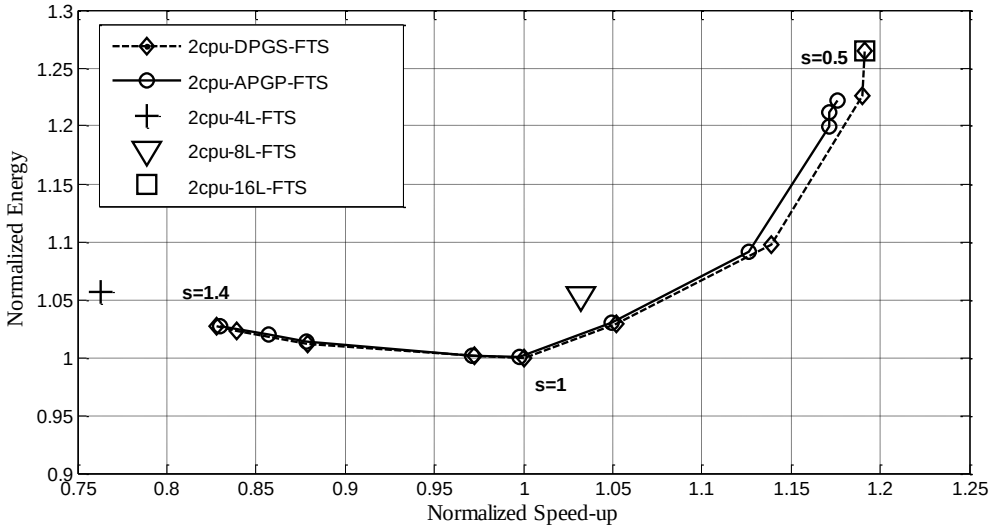


Figure 6.10 Normalized energy vs. normalized execution time for threads dominated by low utilization kernels. The idle period is in the range [5000, 10000] VP clock cycles.

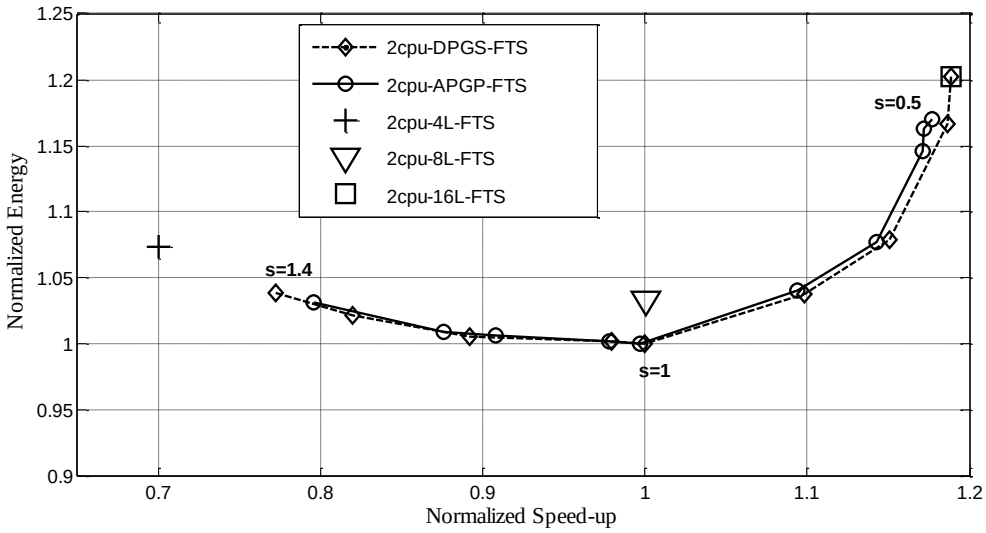


Figure 6.11 Normalized energy vs. normalized execution time for threads dominated by mixed utilization kernels.

To demonstrate the viability of such a performance-vs.-energy trade-off approach, Figures 6.10 to 6.12 plots the normalized energy versus the normalized speed-up for a set of thresholds obtained by multiplying the original thresholds with a scale factor ***s*** that lies

between 0.5 and 1.4. Based on the data from these figures, the lower limit of ***s*** (i.e., 0.5) is chosen such that DPGS and APGP achieve close to the maximum possible performance, which is given by FTS when all 16 lanes are used (2cpu-16L-FTS).

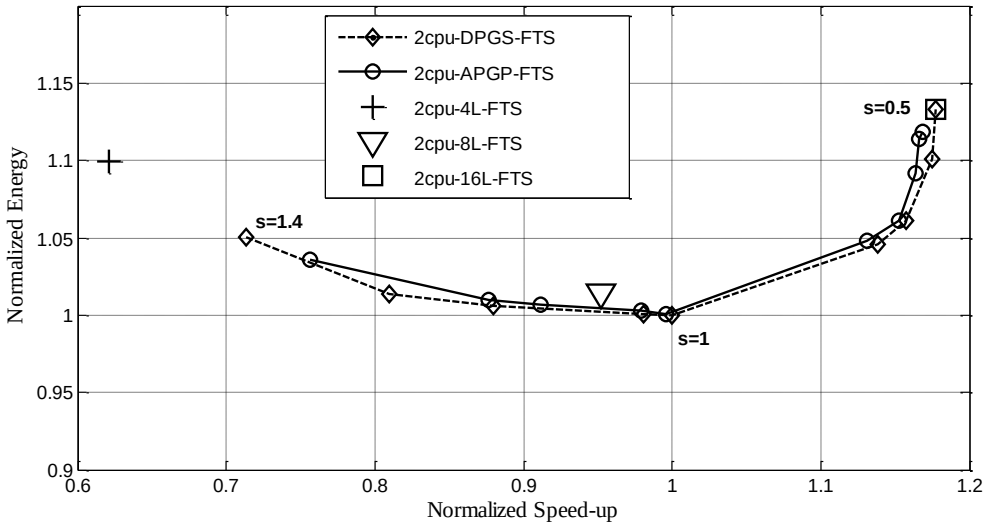


Figure 6.12 Normalized energy vs. normalized execution time for threads dominated by high utilization kernels.

To simplify the process without loss of generality, the same scaling factor is used for all the thresholds, and the simulations are conducted for threads with idle periods in the range [5000, 10000] cycles. For FTS sharing and DPGS-driven power gating, the maximum performance is reached for ***s***=0.5; the configuration is obviously for 16L. APGP follows closely the behavior of DPGS with a small deviation as *s* approaches 0.5. As depicted in Figure 6.10 for threads dominated by low utilization kernels, the performance can be increased by as much as 18% with an increase in the energy consumption by 22%. As shown in Figures 6.11 and 6.12, similar performance improvements can result for threads containing kernels with mixed or high utilization, when an energy increase of 17% and 13%, respectively can be tolerated. On the other hand, both the performance and energy degrade for values of *s* greater than one.

Therefore, it is not desirable to slow down the VP below a certain level (similar to what is called *critical speed* for the DVFS technique [Jejurikar et al., 2004]) in order to avoid simultaneous deterioration in the energy consumption and performance.

Accordingly, a mechanism can be developed to change the absolute and relative thresholds at runtime. Figure 6.13 sketches such an algorithm that minimizes the energy consumption for a given kernel or pair of kernels requiring minimum performance (i.e., minimum value for $U_{ALU}^{MIN} = M * U_{ALU}^{M}$).

Update *s* to minimize the total energy for a given level ($P_{ALU}^{\min}$). of the overall performance.

1. $s = 1$;
2. wait for a profile window;
3. if $M * U_{ALU}^{M} < U_{ALU}^{MIN}$ {
$s = s - 0.05$; // decrement s in steps of 0.05;
update all thresholds *ATh* and *RTh* according to the new *s*;
$ATh = ATh_{s=1} * s$;
$RTh = RTh_{s=1} * s$;
}
4. go back to step **2**.

Figure 6.13 Routine to minimize the energy consumption for a given kernel or pair of kernels requiring minimum performance. This routine runs continuously after a VP event.

6.6 Conclusions

This chapter proposes two energy reduction techniques that employ power gating to dynamically control the width of a shared vector coprocessor (VP) based on lane utilization. The motivation is that a rigid VP architecture shared by multiple cores in a dynamic environment cannot adjust its resources at runtime in order to achieve desired energy-performance levels. The power estimation model introduced in Chapter 5 suggests that, for a given vector kernel or combination of kernels, the dynamic energy does not vary substantially due to fixed workloads. Consequently, two power-gating techniques are proposed to control the number of active VP lanes in order to minimize

the static energy. The first technique, *DPGS*, uses apriori information of lane utilizations to choose the optimal number of lanes that minimizes the energy consumption for known kernels. *APGP* uses embedded hardware utilization profilers in order to make runtime decisions about VP resizing. To find each time the optimal number of lanes that minimize the energy consumption, the current state of the VP is changed sequentially until an optimum VP state is reached for the current workload. Floating-point intensive benchmarking on an FPGA prototype show that these techniques reduce the total energy by 30-35% while maintaining performance comparable to a multicore with the same amount of VP resources, where each core has exclusive access to its own dedicated VP.

Additionally a trade-off mechanism is developed to increase the performance at the expense of increased energy. This allows an increase in performance of about 18% while increasing the energy consumption by 22% for scenarios with low utilization kernels and by 13% for scenarios with high utilization kernels. Also, the work can be extended to find the optimal scaling factor ***s*** that minimizes the energy under a given performance constraint.

Finally, this work could be a starting point in developing a framework that minimizes the energy consumption within a single streaming multiprocessor in a GPU Fermi architecture [Nvidia CUDA, 2011] by adjusting at runtime the number of active CUDA cores based on the present workload (i.e., warp throughput, etc). Memory bounded applications will benefit mostly since the cores will be underutilized. Of course, this will require a few changes in the PTX ISA and GPU architecture; e.g., support for dynamic power gating of individual CUDA cores and dynamic adjustment of the number of threads in a warp.

CHAPTER 7

ASIC IMPLEMENTATION OF THE VECTOR PROCESSOR

As presented in Chapter 4, pre-silicon prototyping was initially carried out on Virtex 5 and 6 FPGAs. The current chapter presents the ASIC (Application-Specific Integrated Circuit) implementation of the Vector Processor (VP). Section 7.1 shows the conversion details for the FPGA to ASIC transition. Section 7.2 presents the Synopsys design flow used to implement and analyze the design, and Section 7.3 details the decision process of choosing a specific process corner. Section 7.4 shows the obtained results and Section 7.5 draws the conclusions of this Chapter.

7.1 FPGA to ASIC Design Transition

In order to implement the Vector Processor in ASIC, some of the Xilinx IP (Intellectual Property) cores, which are particular to the FPGA implementation, have to be replaced. Table 6.1 shows the VP components replaced for the FPGA to ASIC transition, along with some details. The Add/Subtract and Multiply components are replaced with designs taken from Open Cores [Open Cores, 2012] and customized/optimized to have a latency of six clock cycles. Also, the Synopsys DesignWare library offers floating point support. However, like in the case of the Xilinx IPs, these IPs are encrypted and their customization for different cycle latencies is done directly by the synthesis tool. As compared with non-encrypted IPs these modules do not perform too well for deep pipelines (due to high clock latencies), especially when the clock gating feature is

enforced. Therefore, the ASIC implementation uses the in-house built custom floating point modules.

Table 7.1 VP Components Replaced for the FPGA to ASIC Transition

Component	FPGA	ASIC
SPFP ADD/SUBTRACT	Xilinx IP core 6 cycles latency	OpenCores [Open Cores, 2012] IP in-house customized and optimized, or Synopsys DesignWare IP. 6 cycles latency
SPFP MULTIPLY	Xilinx IP core 6 cycles latency	OpenCores IP in-house customized and optimized, or Synopsys DesignWare IP. 6 cycles latency
Vector Register File Bank	Xilinx Block RAM 512 32-bit elements 4 Read and 2 Write ports	Latch-based Register File - for simulations. CACTI model [Muralimanohar et al., 2012] - for area/delay/power analysis. 128 32-bit elements per bank; the VRF within each lane has 4 banks. 4 Read and 2 Write ports
Vector Memory Bank	Xilinx Block RAM 8 KBytes 2 Read/Write ports, 32-bit width	CACTI model - for power analysis
LDST Address Computation	Xilinx DSP block for multiplication	Synopsys DesignWare Basic Block multiplier inferred during synthesis

Xilinx Block RAMs from the Vector Register File are replaced with latches inferred by the synthesis tool from a behavioral description. However, big register files like those used in Vector Processors require Static Random Access Memory (SRAM) to retain the register state. The Synopsys and TSMC (Taiwan Semiconductor Manufacturing Company) libraries do not provide the SRAM models for feature sizes lower than 90 nm and the solution is to use CACTI 6.0 [Muralimanohar et al., 2012] to characterize/extract the area, delay and power figures for VRF. Similarly, CACTI is used to extract the area, time and power parameters for the Vector Memory banks. Table 6.2 shows all the parameters for the VRF and Vector Memory SRAM block given by CACTI for a design frequency of 1 GHz and a feature size of 40nm.

Additionally, in order to test the ASIC version of the VP and also run netlist simulations, the rest of the VP environment (i.e. the Microblaze scalar cores, the PLB bus, and the DMA and Memory Controller) are replaced by a Verilog testbench that is capable of issuing VP instructions and moving data to/from the Vector Memory.

Table 7.2 VRF and Vector Memory Area and Power Consumption Figures for a Frequency of 1.0 GHz (CACTI 6.0 for a Feature Size of 40nm)

Component	Details
Vector Register File bank	4 Read ports/ 2 Write ports (6 ports totally) 128 32-bit elements (2 KBytes) VDD: 1.061 V Access time (ns): 0.9096 Total read dynamic energy per read port (nJ): 0.00192 Total read dynamic power per read port at max freq (mW): 4.177 Total standby leakage power per bank (mW): 2.098 Total area (μm^2): 28017.44
Vector Memory bank	2 Read/Write ports 2048 32-bit elements (8 KBytes) VDD: 0.661V Access time (ns): 0.8069 Total read dynamic energy per read port (nJ): 0.00316 Total read dynamic power per read port at max freq (mW): 7.763 Total standby leakage power per bank (mW): 3.102 Total area (μm^2): 82835.40

7.2 ASIC Design Flow

The hierarchical design flow is followed with the application of standard EDA tools. Synopsys VCS-MX [Synopsys VCS-MX, 2011] is used for simulation and verification of the RTL design and the netlist produced by synthesis. The Synopsys Design Compiler [Synopsys DC, 2011; Synopsys DC Optim., 2011] is used for synthesis and Synopsys Prime Time [Synopsys PX. 2011] is used for timing and power analysis.

Figure 7.1 shows the Synopsys front-end design and power flow. It comprises the following steps [Beldianu, 2012]:

(i) Simulation of the RTL description logic using Synopsys VCS-MX for

performance purposes.

(ii) Synthesis using the Synopsys Design Compiler.

(iii) Simulation of the netlist produced by synthesis using Synopsys VCS-MX.

(iv) Analysis of the power consumption for the implemented design using Synopsys Primetime-PX that involves stimuli provided by the testbench.

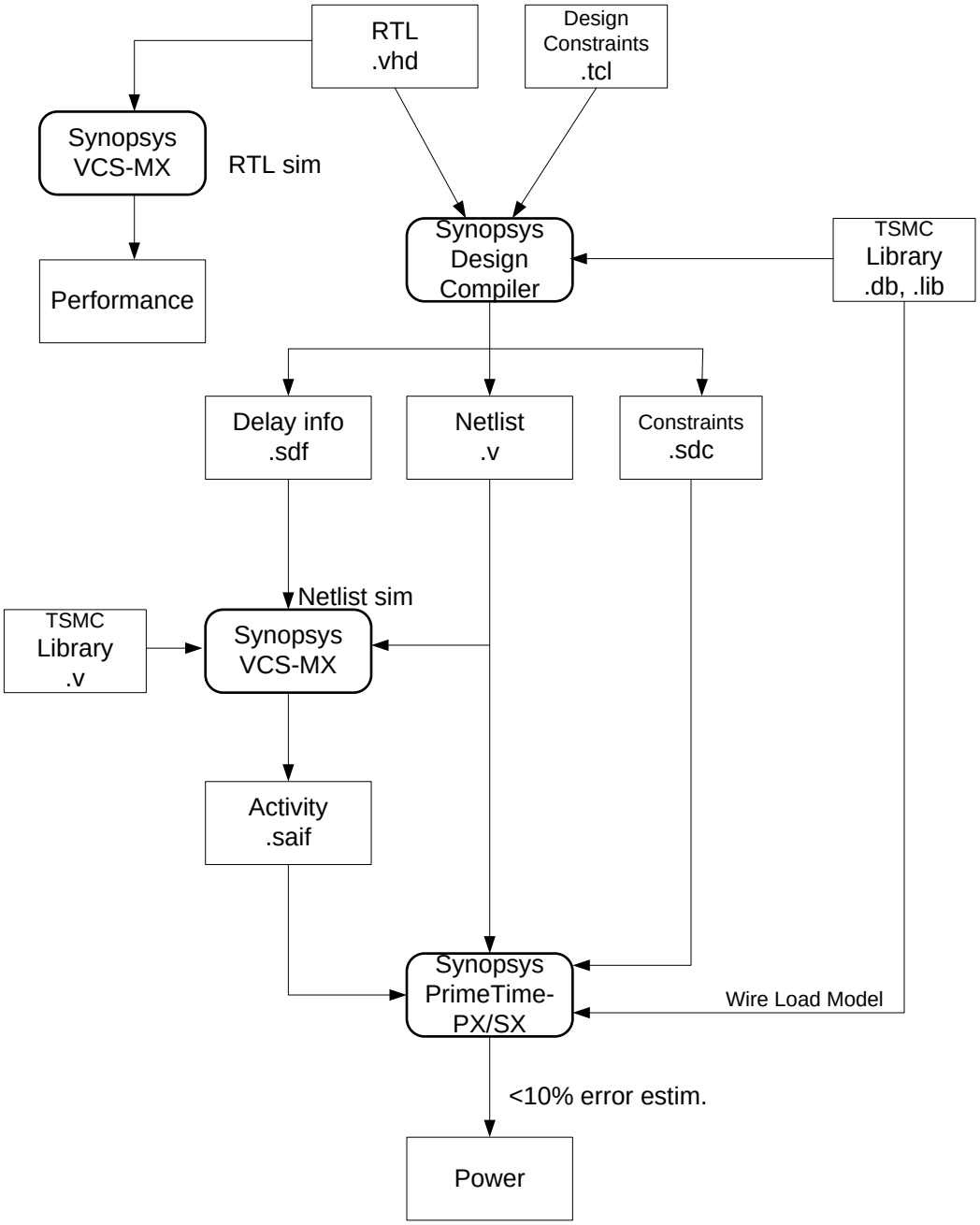


Figure 7.1 Synopsys front-end design and power analysis flow.

The synthesis process involving the Design Compiler comprises at least the following steps:

(i) ***Specify the working libraries***. The link, target, symbol, and synthetic libraries for the Design Compiler should be specified. The *link* and *target* libraries are technology libraries that define the semiconductor vendor's set of cells and related information, such as cell names, cell pin names, delay arcs, pin loading, design rules, and operating conditions. The *symbol* library defines the symbols for schematic viewing of the design. In addition, synthetic libraries specify the DesignWare Synopsys IPs that will be inferred during synthesis.

(ii) ***Read the design***. Reading the design consists of loading all the VHDL/Verilog design files into the Design Compiler environment.

(iii) ***Define design environment***. This step defines operating conditions (manufacturing process, temperature, and voltage), loads, drives, fanouts, and wire load models. Wire load modeling consists of estimating the effect of wire length and fanout on the capacitance, resistance, and area of nets. The Design Compiler uses these physical values to calculate wire delays and circuit speeds. Additionally, the PrimeTime-PX power estimator uses these models to estimate the power consumption of the wire parasitics. Wire load models are based on statistical information collected from each technology process and are developed by every Semiconductor vendor. The models include coefficients for capacitance, resistance, and area per unit length, and a fanout-to-length table for estimating net lengths (the fanout number determines the wire length).

(iv) ***Set design constraints.*** Constraints define the design goals for timing (clocks, clock skews, input delays, and output delays) and area (maximum area). The Design Compiler will try to meet these goals, but no design rules are violated by the process.

(v) ***Optimize the design***. This is the actual step where synthesis is done.

(vi) ***Analyze and Resolve Design Problems.*** **The** Design Compiler can generate numerous reports comprising results of design synthesis and optimization; for

example, area, constraint, and timing reports. These reports could be used to analyze and resolve any design problems or to improve the synthesis results.

(vii) ***Save the Design Database***. The design can be saved in various formats (Verilog/VHDL netlist file or design data file `.ddc`). Additionally, a Standard Delay Format (SDF) back-annotation file is saved that contains all the delay information of the cells and nets used during the gate-level simulation.

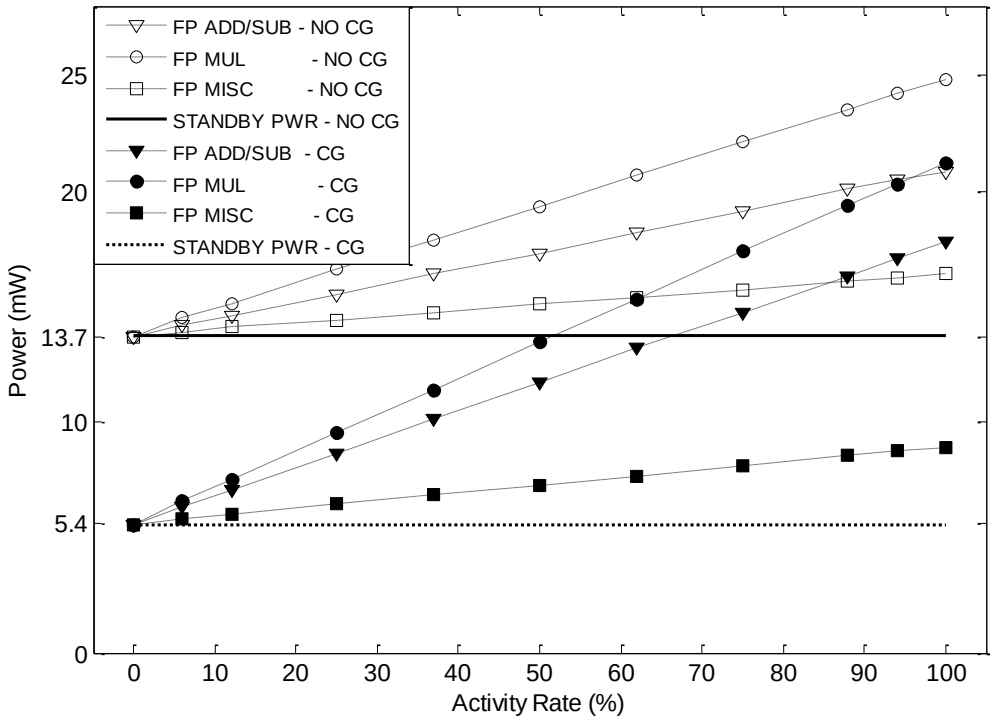


Figure 7.2 Power consumption of the VP Lane execution unit for the ADD/SUB, MUL and MISC operations under various activity rates. FP ADD/SUB - Single Precision Floating Point Add/Subtract; FP MUL - Single Precision Floating Point Multiply; FP MISC - Single Precision Floating Point Absolute, Negate, Move and IntraLane Shift operations; NO CG - No Clock Gating support during synthesis; CG - with Clock Gating support during synthesis; STANDBY PWR - Power consumption when no operation is performed. The lane execution unit is implemented in the 40 nm TSMC process with VDD=1.21V and low voltage threshold. The power consumption is measured at 1 GHz clock frequency and after the system reaches a steady state of operation.

Power estimation with Primetime-PX requires the following inputs: (i) the netlist generated by the synthesis process; (ii) the Switching Activity Interchange Format

(.SAIF) file generated during the gate-level simulation; (iii) the Synopsys design constraints; e.g., time constraints and the wire load model; and (iv) the vendor's technology libraries.

The RTL VHDL/Verilog models use clock gating extensively throughout the VP design and automatic clock gating capabilities provided by the Design Compiler to capture most of the remaining clocked elements. Thus, the power consumption associated with the clock distribution network can be substantially reduced. Figure 7.2 shows the benefits of clock gating for floating point execution units within a vector lane. The standby power consumption (i.e., the power consumption when no operation is performed) can be reduced by 60% when clock gating is enforced.

7.3 Design Exploration

The ASIC implementation targets the 40 nm TSMC High Performance process [TSMC 40nm, 2011]. The 40nm process provides more than twice the density at the same leakage level and more than a 40 percent speed improvement compared to TSMC's 65nm process. The High Performance process targets PC (personal computer), networking, and wired communication applications, and offers Multi-Voltage support with Low, Nominal and High Voltage thresholds (Vt).

Table 7.3 Description of Various TSMC High Performance 40nm Process Corners (PC)

Process corner	Description	
PC_01	Process: 40 nm Vendor: TSMC High Performance non-well biased with UPF (Unified Power Format) and Multi-Voltage support Temperature: 125 °C	VDD: 1.21 V Voltage Threshold: Low
PC_02		VDD: 1.21 V Voltage Threshold: Normal
PC_03		VDD: 1.21 V Voltage Threshold: High
PC_04		VDD: 0.99 V Voltage Threshold: Low

By changing the timing constraint on the VP blocks, the Design Compiler produces different logic topologies, synthesis mappings and gate sizes that trade-off area/power and delay. Designs synthesized with tight delay constraints use more aggressive mappings and larger gates, resulting in higher area and power figures. Figures 7.3a and 7.3b show the Pareto trade-off curves between performance and area/power consumption of the ALU data-path design module for different process corners that are listed in Table 7.3. Presenting the results in a trade-off oriented manner involving area/power and performance provides a more complete picture of the design exploration space to designers. The overall trade-off space spans approximately 4.5× in performance, from about 0.44 to 2 GFLOP/s, and 8× in power, from about 7.6 mW to 60 mW. Additionally, some conclusions can be made:

(i) For a given process corner, the area and power increase as the performance requirement increases.

(ii) Area Pareto points for the high speed process (1.21 V and Low Vt) dominate other Pareto process points even for low performance values.

(iii) At low performance, the low speed process corners dominate the high speed process corners in terms of power. For example, at 0.66 GFLOP/s, PC_03 dominates PC_01 and PC_02.

(iv) Finally, only PC_04 is able to provide a performance greater than 1.1 GFLOP/s since the other process corners are not able to meet these requirements.

According to conclusion iv, in order to provide throughputs over 1.1 GFLOP/s, the process corner adopted throughout the rest of the Chapter is PC_01 (that is, TSMC 40nm High Performance with VDD=1.21V and Low Vt).

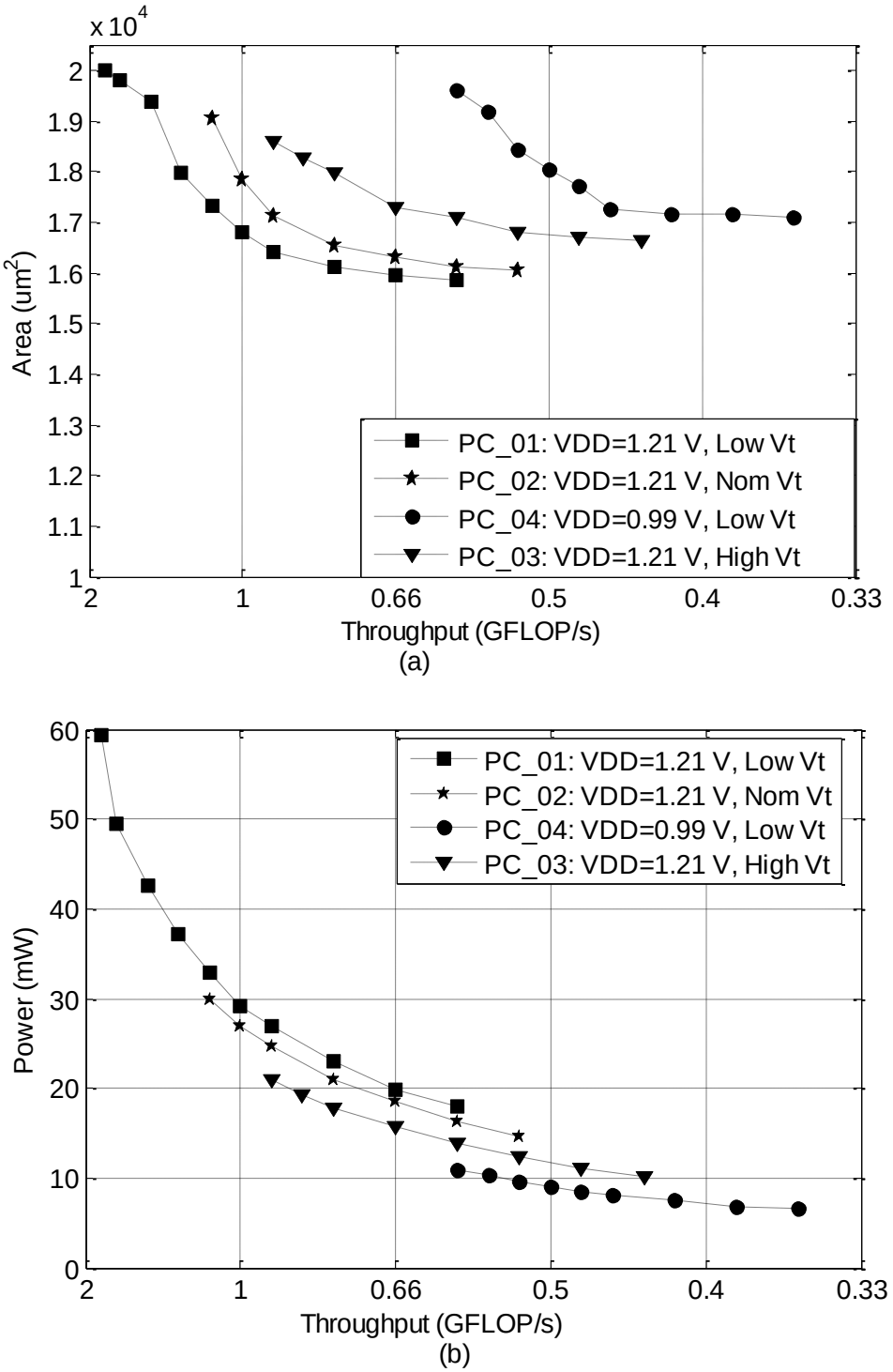


Figure 7.3 Pareto trade-off curves for the ALU module within a lane involving: (a) performance and area; (b) performance and power. Details for the PC_01 to PC_04 process corners are shown in Table 7.3.

7.4 ASIC Implementation Results

This section presents the timing, area and power results for the ASIC implementation of the Vector Processor.

Table 7.4 shows the maximum frequency for the main components of the VP given by synthesis for the TSMC 40 nm High Performance process. The wire load model chosen throughout the entire synthesis process is "TSMC512K_Lowk_Conservative", which is the most conservative load model; therefore, the maximum frequencies could be potentially improved further. The critical path delay for a lane corresponds to the floating point multiply module. However, a careful design with increased pipeline depth could further increase the maximum frequency.

Table 7.4 Maximum Working Frequency for the Main VP Components

Component		**Max Frequency (GHz)**
ALU	**ALU_CTRL**	2.08
	ALU_ADD/SUB	1.98
	ALU_MUL	1.78
LDST	**LDST_CTRL**	1.97
VC		2.12

Table 7.5 shows detailed area and power results for all the VP components. As depicted in Figure 7.4, more than three quarters of the lane area is occupied by the four SRAM banks used to implement VRF. Overall, only 13.7% of the VP area is taken by custom logic; the rest of the design is occupied by memory blocks within the VRF and Vector Memory. These results conform with other embedded designs [Balfour, 2010], where most of the chip area is occupied by regular SRAM. On the other hand, more than 66% of the power consumption is caused by the ALU and LDST units.

Table 7.5 Area and Power Results for Each VP Component, and Total VP Area for Various Configurations. The Standby Power is the Power Consumption when the VP is Idle (it Involves Leakage Power). The Maximum Power for Each Component Includes the Standby Power. The Percentage Figures are Relative to the First Module in the Hierarchy; i.e., ALU and LDST. The Power Consumption is Measured at 1.0 GHz Clock Frequency. The Total VP Area Includes the Vector Memory and One Equivalent Gate Comprises Four Transistors [TSMC 40nm, 2011]

Component		Area (μm^2)	Power (mW)		
			Leakage	Standby	Max
ALU		20659 (100%)	2.756	9.504	31.3
	ALU_CTRL	1951 (9.4%)	0.236	2.713	4.58
	ALU_ADD/SUB	5482 (26.5%)	0.717	2.311	21.20
	ALU_MUL	8126 (39.3%)	1.212	2.817	14.70
	ALU_MISC	1271 (6.2%)	0.144	0.239	3.62
	ALU_vc0_q	1571 (7.6 %)	0.177	0.308	0.617
	ALU_vc1_q	1571 (7.6 %)	0.182	0.308	0.617
LDST		7213.35 (100 %)	0.884	6.307	12.31
	LDST_CTRL	4081.91 (56.6 %)	0.66	5.68	11.21
	LDST_vc0_q	1563.0879 (21.7 %)	0.177	0.307	0.758
	LDST_vc1_q	1558.32 (21.6 %)	0.182	0.312	0.758
VRF - Latch based (one bank)		35637	3.891	3.891	8.979
VRF - SRAM (CACTI) (one bank)		28017	2.098	2.098	2.098+ 4.177/port
VC		3076	0.341	1.553	3.61
Scheduler		2427	0.281	1.134	
Crossbar Switch		14196	1.017	8.369	14
Vector Memory - SRAM (one bank)		82835	3.102	0	3.102+ 7.763/port

	TOTAL VP AREA	Gate Count
VP 2×2	0.466 mm^2	1166638
VP 4×4	0.911 mm^2	2276331
VP 8×8	1.798 mm^2	4495745
VP 16×16	3.573 mm^2	8934571
VP 32×32	7.125 mm^2	17812148

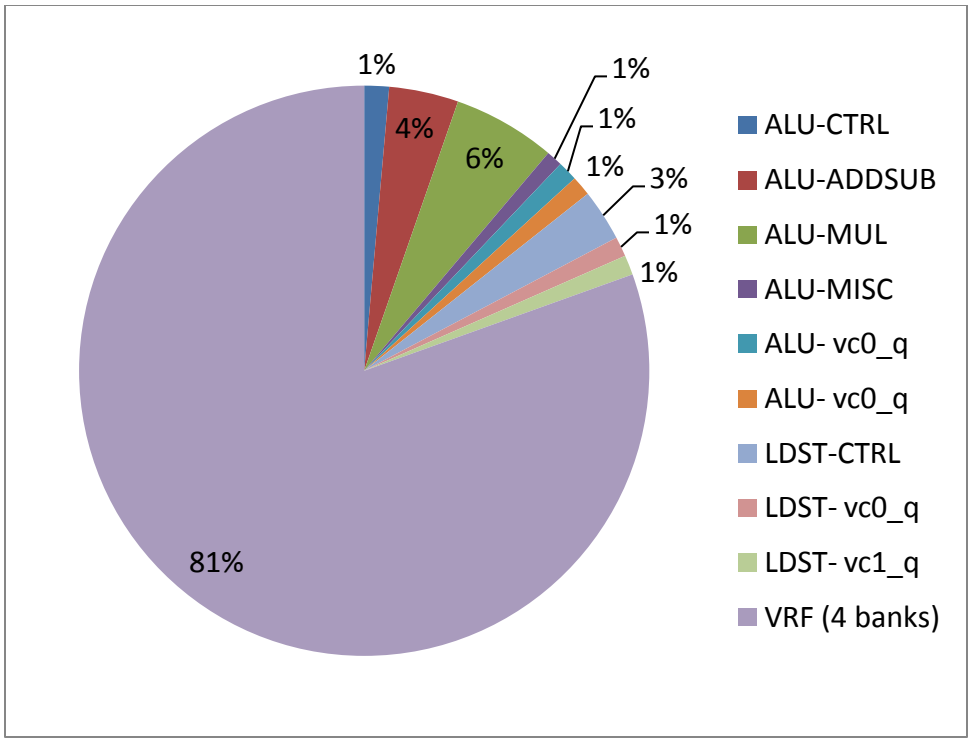


Figure 7.4 VP lane area breakdown. A lane has four VRF banks, each one containing 128 32-bit elements. The power consumption is measured at 1.0 GHz clock frequency.

Figure 7.5 shows the power consumption breakdown gathered from simulations of an 8×8 VP running various applications. As can be depicted, the FIR and FFT kernels exhibit higher dynamic power consumption for the ALU unit as compared with the LDST unit. The MM, LU decomposition and sparse matrix-vector multiplication kernels demonstrate high utilization of the LDST unit, and, thus, high power for the LDST controller and the Vector Memory banks. These conclusions are similar with the ones drawn in Section 5.2. Additionally, the dynamic power model for the ASIC design conforms to the dynamic power model developed in Section 5.2 and presented in Table 5.1. As in Chapter 5, the model assumes a fixed combination of Technology Process, Voltage, Frequency and Temperature, and is easy to extend since only constants change.

The leakage power is between 13.39% and 19.24% of the total power consumption for FIR32 and sparse matrix-vector multiplication kernel, respectively. One major power consumer is the clock distribution network even when the VP is idle. This power is produced by Flip-Flops (FFs) that cannot be clock gated, the clocking gates inferred by the synthesis tool when the clock gate option is enforced, and the wires associated with these cells

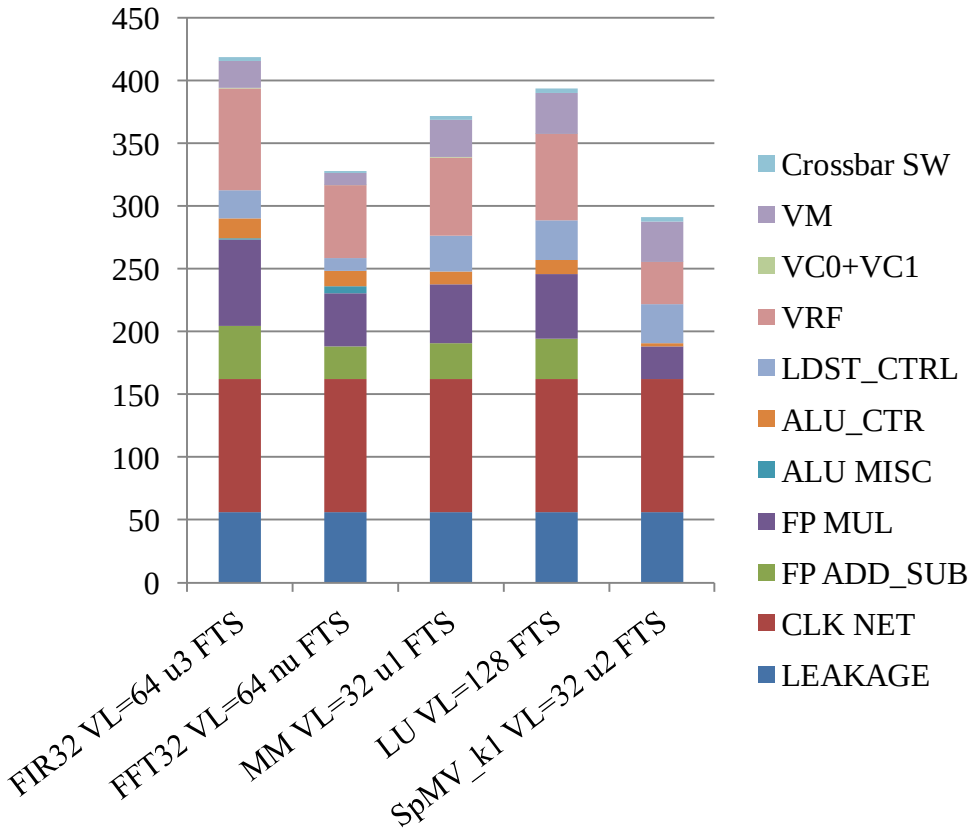


Figure 7.5 Power breakdown (in mW) for a Vector Processor with eight lanes and eight memory banks running different application kernels. Even if contained in each VP component, the leakage and clock distribution network power consumption are displayed separately. The power consumption is measured at 1.0 GHz clock frequency.

The clock power could be reduced by decreasing the number of FFs in the design; that is, by reducing the pipeline stages in the arithmetic units and controllers. If the leakage power contribution could be reduced by increasing the circuit speed to the

maximum working frequency, the clock network power will not since it is proportional to the clock frequency. A possible solution is to use, on top of fine-coarse clock gating implemented by synthesis tool, architecture-level clock gating, i.e., coarse-grained on-off control for clocks that feed whole design units which are not used for some periods of time (more than 4-10 clock cycles). Overall, the standby power is 162 mW and accounts for 38% to 54% of the total power consumption. In the best case, as depicted in Table 7.6, the standby power accounts for 33% of the total power consumption (see the MM FTS scenario). These numbers conform to power data for idle periods of 40/45 nm streaming processing systems [Radeon HD5450, 2010].

Table 7.6 Performance and Power Comparison for Various Application Kernels Running on the ASIC Implementation of the VP with Eight Lanes and Eight Memory Banks. The Applications are Presented in Chapter 3 (nu - no loop unrolling; u1- loop unrolled once). The Power Consumption is Measured After the System Reaches a Steady State

		Average utilization (%)		Execution Time (μs)	Dynamic Power (mW)	Total Power (mW)	Dynamic Energy (nJ)	Total Energy (nJ)	nJ/FLOP
		ALU	LDST						
FIR32 VL=128 nu	CTS	39.24	19.94	0.0207	106.21	268.53	2.19	5.55	0.086
	FTS	75.66	38.24	0.0107	204.78	367.10	2.20	3.94	0.061
	VLS	49.51	25.29	0.0164	134.00	296.32	2.19	4.86	0.075
FFT32 VL=32 nu	CTS	43.29	23.38	0.408	89.64	251.96	36.57	102.80	0.160
	FTS	76.28	42.39	0.230	157.95	320.27	36.40	73.82	0.115
	VLS	62.74	35.11	0.274	129.91	292.23	35.59	80.07	0.125
MM VL=128 u1	CTS	68.3	69.51	0.376	221.67	383.99	83.40	144.47	0.070
	FTS	97.32	98.91	0.264	315.86	478.18	83.46	126.36	0.061
	VLS	81.88	83.4	0.311	265.75	428.07	82.84	133.45	0.065
LU VL=128 nu	CTS	36.17	36.53	0.079	116.357	278.67	8.683	20.796	0.081
	FTS	72.05	72.92	0.0395	231.781	394.10	8.634	14.680	0.057
	VLS	47.23	47.67	0.059	151.936	314.25	8.641	17.873	0.070
SpMV_k1 VL=32 u1	CTS	9.35	38.2	422.25	68.53	230.85	28937	97477	0.581
	FTS	18.22	73.39	213.87	133.54	295.86	28562	63278	0.377
	VLS	14.79	60.11	252.52	108.40	270.72	27372	68358	0.407

Table 7.6 shows the performance, power and energy consumption for various execution scenarios. The same conclusions as in Section 4.2 hold also here. From the power perspective, the main two conclusions are:

(i) For a given application, the lowest dynamic energy is almost the same for all sharing contexts.

(ii) However, if the standby power is included, the advantage of FTS and VLS is substantial, especially for low average utilization (see the SpMV benchmark).

Additionally, from Figure 7.2 the energy per FLOP for the floating point units is around 20pJ/FLOP at the most (SPFP multiply unit). Table 7.6 shows that the best energy consumption achieved by the overall VP system is 61 pJ/FLOP. The rest of the 41 pJ or more are spent to supply data (controllers, VRF and memory) and instructions (VCs, instruction queues and instruction processing).

Using a linear approximation method, the values of the K coefficients for the dynamic power model can be found like in Section 5.2. They are shown in Table 7.7 along with the coefficients obtained previously for the FPGA implementation. If the FPGA coefficients are scaled to the ASIC frequency, then there is a 3× to 20× gap between the FPGA and ASIC implementations. The smallest gap is for the multiply FP unit because the Xilinx IP multiply core encompasses DSP48E slices. These are low power high speed ASIC multiply-add macros.

Table 7.8 shows the mean absolute error for the dynamic power model. The utilization of the lane units can be used to estimate the dynamic power consumption within a 5% confidence interval. Even if not shown here, the highest error deviation is given by the ALU and LDST controllers. The rest of the components have almost a linear dependence between the dynamic power and utilization. VRF and VM are not included since the dynamic power consumption is based on SRAM CACTI models. Also, the FFT kernels exhibit a higher error caused by conditional execution. As stated in Section 5.2,

the actual dynamic power of VRF and ALU for the FFT kernels is under the linear estimation curve.

Table 7.7 Comparison of Power Coefficients for the FPGA (from Table 5.1) and ASIC Implementation

		FPGA (µW/%) 125MHz	FPGA (µW/%) Scaled to 1GHz	ASIC (µW/%) 1GHz	ASIC/FPGA gap
ALU	$K^{INTSR}_{ALU\_CTRL}$	28	224	12.64	17.72×
	$K^{DATA}_{ALU\_CTRL}$	18	144	18.78	7.66×
	K_{ADD_SUB}	215	1720	112.93	15.23×
	K_{MUL}	71	568	182.91	3.10×
	K_{MISC}	18	144	33.6	4.28×
LDST	K^{INTSR}_{LDST}	34	272	18.08	15.04×
	K^{DATA}_{LDST}	55	440	55.37	7.94×
VRF	K_{VRF}	34	272	120	2.26×
VC	K_{VC}	240	1920	98.15	19.56×
VM	K_{MEM_BANK}	147	1176	57.43	20.47×

Table 7.8 Mean Absolute Error for Dynamic Power Estimation of the ASIC Implementation. The *w* Weights are Detailed in Table 5.1

	w_{ADD_SUB} / w_{MUL} / w_{MISC}	Mean Absolute Error (%)
FIR	0.48/0.48/0.04	3.45
FFT	0.36/0.36/0.27	7.38
MM	0.5/0.5/0	3.64
LU	0.5/0.5/0	2.97
SpMVM_k1	0/0.99/0.01	4.03
OVERALL		**4.29**

Finally, Figures 7.6a and 7.6b focus on the area and power scalability of the Crossbar Switch. This module is the only component in VP that is not linearly scalable. As can be seen, the area and power scales quadratically with the number of VP lanes. For sizes bigger than 16, the area scales more than quadratically because of the tight timing constraints (see section 7.3). For these cases, custom interconnect fabrics can be used [Who et al., 2011; Satpathy et al., 2011]. In [Satpathy et al., 2011] a 32×32 64-bit fully

connected crossbar is implemented in 65 nm. The frequency of 1026 MHz provides a total throughput of 2.1 Terabits/s consuming less than 500 mW of power and occupying an area of 0.35 mm^2.

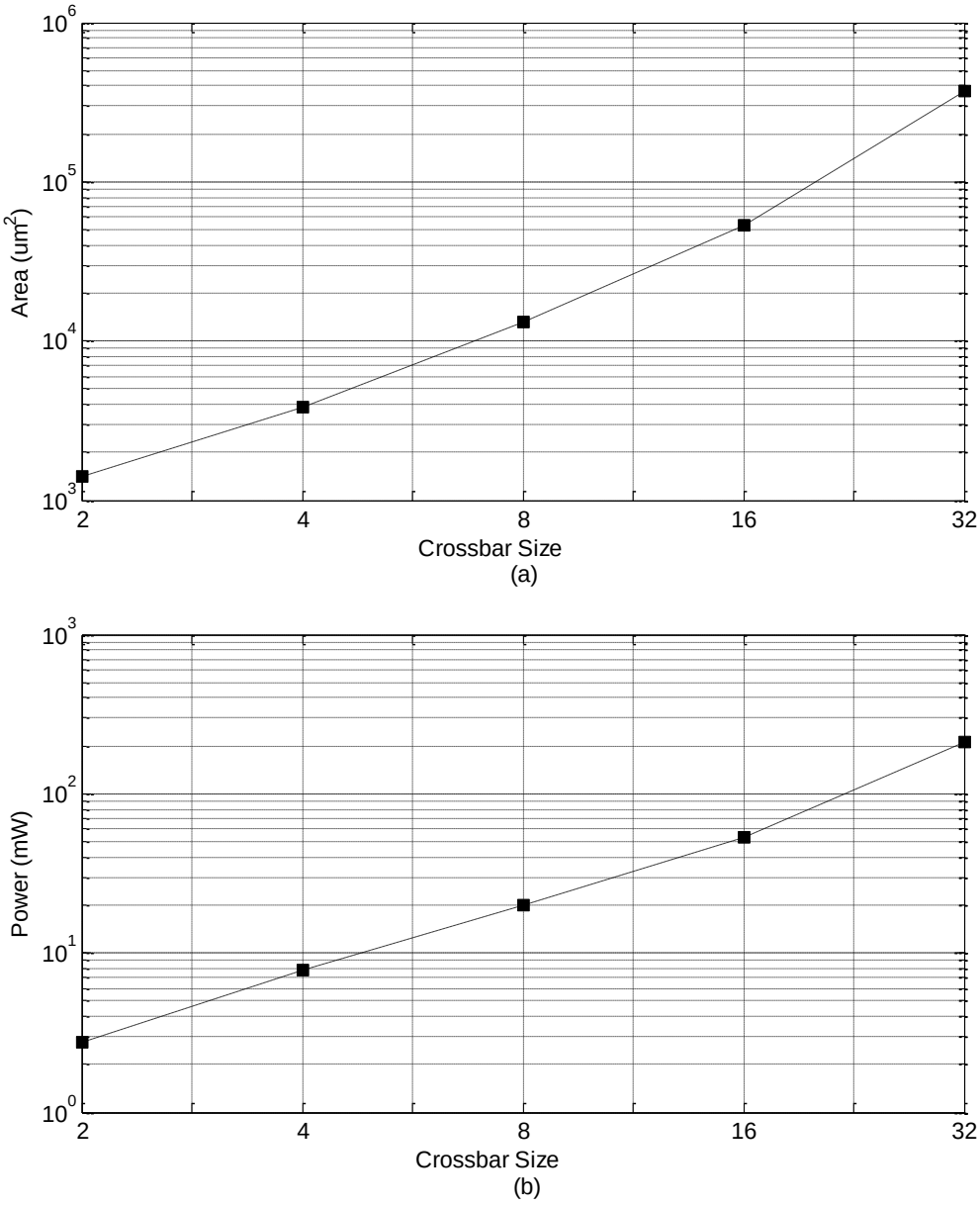


Figure 7.6 Area (a) and Power consumption (b) for an N×N VP crossbar switch, where N is the number of masters. The crossbar contains the arbiters and the logic that supports shuffle operations. The design is synthesized to meet the constraint of 1 GHz for the clock frequency. The power consumption is extracted under maximum LDST utilization.

7.5 Per VRF Bank Dynamic Power Gating

For the ASIC implementation, the lane VRF can be split into multiple banks. Individual banks can be powered down if they are not used. The total number of needed VRF banks to be used collectively by a pair of applications in the FTS context is given by Equation 7.1:

$$No\_Banks = \left\lceil \frac{VL_0 \cdot \text{NoVRegs}_0}{M \cdot K} \right\rceil + \left\lceil \frac{VL_1 \cdot \text{NoVRegs}_1}{M \cdot K} \right\rceil \quad (7.1)$$

where $\lceil \bullet \rceil$ is the ceiling function, VL_i ,and NoVRegs_i are the vector length and the number of vector registers requested by application *i*, for *i*=0 or 1; *M* is the number of lanes and *K* is the number of elements in each bank. For example, the current ASIC implementation has a VRF with four banks and *K*=128 elements in each bank; Table 7.9 shows the number of banks required by some scenario when the number of active VP lanes is eight.

Table 7.9 Number of VRF Banks Required by Each Scenario

Scenario	Number of required banks
CTS, FIR, VL=64, u3, NoVRegs=11	1
FTS, FIR, VL=64, u3, NoVRegs=11	2
FTS, FIR, VL=256, u3, NoVRegs=11	4
FTS, FIR, VL=256, u3, NoVRegs=11 & FIR, VL=64, u3, NoVRegs=11	3
FTS, FFT, VL=32, nu, NoVRegs=21	2
FTS, FFT, VL=64, nu, NoVRegs=21	4
FTS, MM, VL=64, nu, NoVRegs=7	2
CTS, LU, VL=128, u1, NoVRegs=6	1

As a consequence, the standby power of a lane (i.e., the leakage power plus the clock network power) will change with VP size changes in order to minimize the energy

consumption. Considering a relative threshold ($RTh_{M/2M}$) between the configurations with M and 2M lanes, the lane's standby power is:

M-lane configuration : $P_{SB}^{L} - M(TB - B)P_{SB}^{VRF\_BANK} - (L - M)\left(P_{SB}^{LANE} - P_{OFF}^{LANE}\right)$

2M-lane configuration : $P_{SB}^{L} - 2M(TB - B/2)P_{SB}^{VRF\_BANK} - (L - 2M)\left(P_{SB}^{LANE} - P_{OFF}^{LANE}\right)$

where *TB* is the total number of banks from VRF and *B* is the number of active banks in the *M*-lane configuration. Therefore, $RTh_{M/2M}$ becomes:

$$RTh_{M/2M} = \frac{2M}{M} * \frac{P_{SB}^{L} - M(TB - B)P_{SB}^{VRF\_BANK} - (L - M)\left(P_{SB}^{LANE} - P_{OFF}^{LANE}\right)}{P_{SB}^{L} - 2M(TB - B/2)P_{SB}^{VRF\_BANK} - (L - 2M)\left(P_{SB}^{LANE} - P_{OFF}^{LANE}\right)} \quad (7.2)$$

This gives a slight advantage to higher-lane configurations since less power is consumed due to the capability of powering down more memory banks.

7.6 Energy Minimization with Quality of Service (QoS)

In Section 6.2 the overall energy is minimized without taking into consideration a minimum level of performance required by the application. This section deals with minimizing the energy consumption given a performance constraint. That is, an application *i* (received from CPU *i*) will require a minimum performance level which, at the lane level, can be translated into a minimum utilization figure ($U_{ALU\_MIN}^{VCi}$).

$$\begin{cases} \min(E) \\ \text{subject to } U_{ALU}^{VC0} > U_{ALU\_MIN}^{VC0} \text{ and } U_{ALU}^{VC1} > U_{ALU\_MIN}^{VC1} \end{cases}$$

Based on each application request, it is the responsibility of the VP hardware manager, which includes the PG Controller, to decide how to resize the VP or to change

the weight of the Weighted Round Robin (WRR) Arbiter inside each lane (see Section 2.2.2).

There are two ways to achieve the required performance:

(i) Give more priority to a thread that does not currently meet its performance requirement.

(ii) If (i) does not provide the required performance for both threads, resize the VP by increasing the number of lanes.

Figure 7.7 shows part of the state machine displayed in Figure 6.4. The absolute threshold transition condition is appended with new conditions that assure the required QoS for each thread.

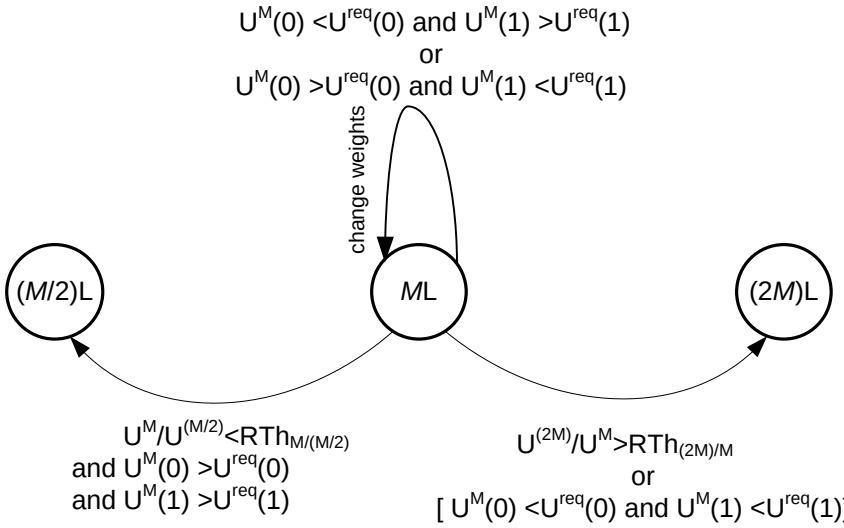


Figure 7.7 PG Controller state machine update for QoS support. $U^{M}(0)$ and $U^{M}(1)$ are the monitored utilizations corresponding to VC0 and VC1, respectively, and $U^{req}(0)$ and $U^{req}(1)$ are the required utilizations for the two threads.

7.7 Conclusions

This chapter presents the ASIC implementation of the Vector Processor. The FPGA to ASIC transition details are presented along with the Synopsys design flow, and time/area and power results. The main conclusions are:

(i) The dynamic power model developed in Section 5.2 also applies to the ASIC implementation with a produced error within a 5% confidence interval.

(ii) The standby power component, i.e., the power consumption when the VP is idle, accounts for more that 33% of the total power consumption; it can reach more than 55% when the utilization is low. The major component of the standby power consumption is the clock distribution network that cannot be gated.

(iii) For a given application, the lowest dynamic energy is almost the same for all sharing contexts. However, if the standby power is included, the advantage of FTS and VLS is substantial, especially under low average utilization (see the SpMV benchmark).

Just for the sake of comparison, Table 7.9 compares the peak performance, maximum power and power efficiency for several systems implemented in the 40 nm technology.

Table 7.10 Power Efficiency Comparison with Other Streaming Processors

	Peak Performance (Single-Precision) GFLOPs	**Total Power (W)**	**GFLOPs/W**
This work (8 Lanes VP)	8	0.480	16.66
Nvidia GeForce G210M GPU	72	14	5.14
AMD Radeon HD 5450 GPU	104	19.1	5.44
Nvidia Quadro 1000M GPU	268.8	45	5.97
Nvidia Tesla C2050 GPU	1030	238	4.32
IBM BlueGene/Q (65 nm) Supercomputer with 65,536 processors	170×10^3	85,000	2.01

Therefore, this ASIC work provides the best efficiency among well known streaming processors. However, it should be mentioned that in this comparison the power consumed by the buses, DMAs, off-chip memories and other components is not taken into account in the proposed system. However, this will not change the VP power model but rather will reduce the power efficiency of the system.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

This thesis presents a VP design methodology that can realize three architectural contexts for the implementation of shared vector coprocessors in multicores, in order to efficiently utilize silicon resources. Additionally, an energy minimization mechanism is developed. The central motivation of this work is to develop VP architectures that can yield high utilization of resources with low energy budgets.

Chapter 2 proposes three VP ***sharing*** architectures in detail and presents their implementation on an FPGA device. The first sharing architecture, coarse-grain temporal sharing (CTS) consists of temporally multiplexing sequences of vector instructions ideally arriving from different threads. However, providing a per-core exclusive access to the vector resources does not maximize their utilization. Fine-grain temporal sharing (FTS) consists of spatially multiplexing individual instructions issued by different threads, in order to increase the utilization of the functional units. Finally, vector-lane sharing (VLS) consists of simultaneously allocating distinct vector lanes or collections of them to distinct cores/threads. VLS provides better performance and energy results with kernels that have vector lengths smaller that the total number of lanes. Also, a guaranteed Quality-of-Service support for the VP is presented; more specifically, the Scheduler assigns coprocessor resources based on the priorities of the active threads.

Chapter 4 evaluates the performance and energy consumption for these coprocessor sharing contexts by implementing several floating-point applications (presented in **Chapter 3**) using an FPGA-based prototype. FTS exhibits the biggest

speedup and smallest energy consumption; it is followed by VLS. Moreover, under low resource utilization FTS doubles the speed-up and reduces the energy consumption by as much as 50% as compared to the case where a core (and its threads) has exclusive access to the vector coprocessor.

Chapter 5 presents performance models for these coprocessor sharing contexts as well as power estimation models based on observations deduced from the experimental results. These models suggest several techniques to increase the performance or reduce the energy consumption:

(i) Increase the data-level parallelism by increasing the vector length.

(ii) Increase the instruction-level parallelism at compile time by loop unrolling or other techniques.

(iii) Use multiple threads in a multiprocessor environment to increase the vector coprocessor utilization.

The analysis shows that the last technique can be superior to the former two combined. Therefore, the lack of adequate data-level parallelism in an application can be overcome by sharing the coprocessor resources among many cores and their threads.

Chapter 6 proposes two ***energy reduction techniques*** that employ power gating to dynamically control the width of a shared VP based on lane utilization. The motivation is that a rigid VP architecture shared by multiple cores in a dynamic environment cannot adjust its resources at runtime in order to achieve desired energy-performance levels. Based on a power estimation model introduced in Chapter 5 which suggests that, for a given vector kernel or combination of kernels, the dynamic energy does not vary substantially due to fixed workloads, this work proposes two power-gating techniques to control the number of active VP lanes in order to minimize the static energy. The first

technique, *DPGS*, uses apriori information of lane utilizations to choose the optimal number of lanes that minimizes the energy consumption for known kernels. *APGP*, on the other hand, uses embedded hardware utilization profilers in order to make runtime decisions about VP resizing. To find each time the optimal number of lanes that minimize the energy consumption, the current state of the VP is changed sequentially until an optimum VP state is reached for the current workload. Floating-point intensive benchmarking on an FPGA prototype shows that proposed techniques reduce the total energy by 30-35% while maintaining performance comparable to a multicore with the same amount of VP resources, where each core has exclusive access to its own dedicated VP. Additionally a trade-off mechanism is developed to increase the performance at the expense of increased energy. This allows an increase in the performance by about 18% while increasing the energy consumption by 22% for scenarios with low utilization kernels and by 13% for scenarios with high utilization kernels. Also, the work can be extended to find the optimal scaling factor ***s*** (i.e., thresholds multiplying factor) that minimizes the energy under a given performance constraint.

Finally, **Chapter 7** presents the ASIC implementation of the VP. The FPGA to ASIC transition details are presented along with the Synopsys design flow and time/area and power results. The main conclusions are:

(i) The dynamic power model developed in Section 5.2 applies as well to the ASIC implementation with an error within a 5% confidence interval.

(ii) The standby power component, i.e., the power consumption when the VP is idle, accounts for more that 33% of the total power consumption; it can reach more than 55% when the utilization is low. The major component of the standby power consumption is the clock distribution network that cannot be gated.

(iii) For a given application, the lowest dynamic energy is almost the same for all

sharing contexts. However, if the standby power is included, the advantage of FTS and VLS is substantial, especially under low average utilization (e.g., the SpMV benchmark).

(iv) The proposed VP provides a power efficiency of 16.66 GFLOPs/W, which is very high compared to commercial high-performance GPUs and supercomputers.

As a final conclusion, a lane-based rigid SIMD environment exposed to applications with diverse computational intensities (i.e., ratios between sustained computation and memory bandwidth) may produce energy profiles that may deviate from the minimum energy consumption. Thus, per application resizing of the VP size (i.e., varying the number of lanes and/or the number of arithmetic/LDST units within a lane) and/or the contained data storing resources (i.e., Vector Register File size and/or Vector Memory size) leading to states that can achieve minimal energy consumption will be a great advantage for future *green* SIMD/SIMT architectures. For example, memory-bound applications (i.e., having low computational intensity, like SpMV) may require limited computational resources that may be translated into a small number of lanes for energy minimization. On the other hand, applications with high computational intensity (e.g., FIR and MM) may minimize their energy consumption by increasing the VP size in number of lanes (increasing computational capability).

8.2 Future Work

There are many possible extensions and applications for the VP-based concepts and architectures presented in this dissertation.

CTS-FTS sharing scheme for multicore scalability. Future research should investigate the maximum number of thread/core contexts that can practically coexist in a

VP lane. In order to support VP access from more than two CPUs, two solutions can be derived: (i) a complete FTS context where the lane hardware supports more than two threads at a time; (ii) a mixed CTS-FTS scheme where any two out of N CPUs have simultaneous access to the VP at any given time.

Increasing further the number of scalar cores that have simultaneous access to a VP-based architecture using FTS may not always be a wise choice. Increasing substantially the number of thread/core contexts at the lane level will cause considerable hardware overheads and, more important, *energy overheads* associated with the arbitration logic to schedule threads, as well as to handle per thread storage and logic. According to the Equation 5.6 (second right hand term), adding more logic in VP lane instruction path will produce a larger deviation from the constant dynamic energy model, that is, a considerable part of dynamic energy (consumed for a given task) will increase linearly with number of lanes. Additionally, each scalar core or thread comes with its own contribution to the shared memory bandwidth connecting the off-chip memory and the Vector Memory. Scaling the design to include more than four cores will put a lower limit on the per core/thread available bandwidth which, eventually, will translate in per thread low utilization of the ALU units.

Since in most cases (as shown in Chapter 4) two threads are capable of utilizing properly the lane resources, the latter (ii) solution seems more practical. Therefore, VP access could be granted to only two processors at a time. The number of VCs will still be two but the interconnection network between CPUs and VCs will increase accordingly.

Dynamically sizing the Vector Memory in terms of number of banks. In Chapter 6 the energy minimization techniques were based on changing the number of

active lanes within a VP while keeping the same number of memory banks. For applications that require low Vector Memory footprint and have regular memory access, reducing the number of memory banks will not have significant impact on the performance while at the same time it will reduce the leakage power associated with the unnecessary memory blocks. A few questions have to be answered:

(i) How many banks have to be affected during power down/up based on the application's profiled information. Normally, the number of active banks has to exceed the number of active lanes. Otherwise, even in the case of a regular unit-stride memory access the utilization will be significantly affected.

(ii) When to modify the size of the Vector Memory. Changing the number of banks may require saving and restoring the VM data using the main memory. This is a time consuming task especially if it is applied to a large number of banks. The best solution is probably to perform this operation at the boundaries of large parallel regions for which the VM does not contain many live values.

Compiler support for transfers between VM and main memory. Being a ScratchPad Memory, the Vector Memory has to be explicitly managed by the programmer. Building compiler support to automate memory allocation, and transfers between the VM and main memory will facilitate ease of programming. Some of the existing heterogeneous memory management schemes [Avissar et al., 2001; Sjodin and Platen, 2001; Udayakumaran et al., 2006] could be adapted to realize this objective.

New Applications and ISA support. New high-performance embedded applications could be developed for performance and energy gains along with new instruction set architecture (ISA) support. For example, the VP instruction set could be enhanced with an instruction that enables intra-lane permutations. It will require some new features in the ALU unit and will be similar in functionality to a VMOVE instruction

with an involved index register that results in intra-lane vector element exchanges. In conjunction with the existing inter-lane permutation support (vector shuffle operation), the new ISA feature will be beneficial to applications with deterministic permutation patterns that do not require remote accesses. Therefore, permutations in programs will be divided into two categories: intra-lane and inter-lane permutations. Intra-lane permutations will have the advantage of being non-blocking, and will be implemented with a smaller execution latency and energy consumption. For example, FFT could benefit from this new feature since some of the butterfly permutation stages could be completely mapped to intra-lane permutations (of course, as long as the number of lanes is a power of two, which is the common practice).

Any Number of Vector Lanes. In order to support finer granularity processes that can promote better effectiveness of the energy minimization mechanism proposed in Chapter 6, the VP could be composed of any number of vector lanes; that is, this number may not always be a power of two. This approach is reasonable since, theoretically, the number of lanes that minimizes the energy consumption could take any value, depending on the application and the energy characteristics of the underlying hardware. However, additional hardware support that will increase the complexity of the lane is required. For example, one of the hardware enhancements will center around correct address alignment and computation in the LDST unit that will require a modulo-operation circuit where the divisor could be any number within a given range of numbers. Of course, as stated earlier in this paragraph, some applications may not require this new feature or may be slowed-down if the VP has a number of vector lanes which is not power of two.

Thread Scheduling toward Energy Minimization. In this research the VP resources are acquired by scalar cores immediately after a request, if VP resources are available. Future work may investigate and evaluate policies that allow the system to identify the best time to acknowledge a VP request in order to minimize the energy consumption and to meet the required Quality of Service. The decision will be based on the existing state of the VP and the concurrent acquire requests coming from all scalar cores.

Resize SIMD/SIMT resources towards a cool system. Another approach, somehow different than the energy minimization technique, is to reduce the power consumption density in a region of a chip populated by SIMD/SIMT resources. According to Equation 5.1, as the number of lanes (*M*) increases, the instantaneous utilization of the lane decreases and, thus, the power consumption/density. Therefore, one way to cool down hot spots, i.e., spots with high power consumption/density, is to increase the number of VP lanes. Further investigations could be done on the relation(s) between the number of active VP lanes, Temperature, and the performance of applications that are running at any instant time.

REFERENCES

Agarwal, A. 1992. "Performance tradeoffs in multithreaded processors," Parallel and Distributed Systems, IEEE Transactions on, vol. 3, no. 5, pp. 525-539, Sep. 1992.

Anis, M., Areibi, S., and Elmasry, M. 2003. "Design and optimization of multithreshold CMOS (MTCMOS) circuits," IEEE Trans. Computer-Aided Design Integrated Circuits Systems, vol. 22, no. 10, pp. 1324-1342, Oct. 2003.

Asanovic, K. 1998. "Vector Microprocessors," PhD thesis, University of California at Berkeley, 1998.

Avissar, O., Barua, R., and Stewart, D. 2001. "Heterogeneous memory management for embedded systems," In Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '01), ACM, pp. 34-43, New York, USA, 2001.

Azevedo, A. and Juurlink, B. 2009. "Scalar processing overhead on SIMD-only architectures," In Proceedings of 20th IEEE International Conference on Application-specific Systems, Architectures and Processors. IEEE, pp. 183-190, 2009.

Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. 2002. "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems," in CODES '02: Proceedings of the 10th International Symposium on Hardware/Software Codesign. New York, NY, USA: ACM, pp. 73-78, 2002.

Balfour, J. 2010. "Efficient embedded computing," PhD Thesis, Stanford, May 2010 (Figure 2.1).

Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S., and Sancho, J.C. 2008. "Entering the petaflop era: The architecture and performance of Roadrunner," In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08), IEEE Press, Piscataway, NJ, USA, Article 1, 11 pages.

Beldianu, S. F. and Ziavras S. G., 2011a. "On-chip vector coprocessor sharing for multicores," Parallel, Distributed and Network-Based Processing (PDP), 19th Euromicro International Conference on, pp. 431-438, 9-11 Feb. 2011.

Beldianu, S. F. and Ziavras S. G. 2011b. "Multicore-based vector coprocessor sharing for performance and energy gains," accepted for publication, ACM Transactions on Embedded Computing Systems, 2012.

Beldianu, S. F., Dahlberg C., Steele, T. and Ziavras, S. G. 2011c. "Versatile design of shared vector coprocessors for multicores," re-submitted to Microprocessors and Microsystems: Embedded Hardware Design after a minor revision.

Beldianu, S.F. 2012. "A complete RTL to Timing/Area/Power tutorial with Synopsys Tools," Technical Report, NJIT, 2012.

Beldianu, S.F., and Ziavras, S.G. 2012. “Performance-Energy Optimizations for Shared Vector Accelerators in Multicores” submitted to IEEE Transactions on Computers, 2012.

Bsoul, A.A.M., Wilton, S.J.E. 2010. “An FPGA architecture supporting dynamically controlled power gating,” Field-Programmable Technology (FPT), 2010 International Conference on, pp. 1-10, Dec. 2010.

Butts, J.A., and Sohi, G.S. 2000. “A static power model for architects,” Microarchitecture (MICRO-33) Proceedings. 33rd Annual IEEE/ACM International Symposium, pp. 191-201, 2000.

Chishti Z. and Vijaykumar T. N. 2008. “Optimal power/performance pipeline depth for SMT in scaled technologies,” IEEE Trans. Comput., vol. 57, no. 1, pp. 69-81, January 2008.

Chen T., Raghavan R., Dale J. N., and Iwata E. 2007. “Cell broadband engine architecture and its first implementation: a performance view,” IBM J. Res. Dev., vol. 51, no. 5, pp. 559-572, 2007.

Cho, J., Chang, H., and Sung, W. 2006. “An FPGA based SIMD processor with a vector memory unit,” In Proceedings of IEEE International Symposium on Circuits and Systems, IEEE, pp. 525-528, 2006.

Cho, S., and Melhem, R. 2008. “Corollaries to Amdahl's Law for Energy,” IEEE Comput. Archit. Lett., vol. 1, pp. 25-28, January 2008.

Chou C.H., Severance A., Brant A.D., Liu Z., Sant S., and Lemieux G. 2011. “VEGAS: soft vector processor with scratchpad memory,” In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11), pp. 15-24, ACM, New York, NY, USA, 2011.

Choy, N. C. K., Wilton, S. 2006. “Activity-based power estimation and characterization of DSP and multiplier blocks in FPGAs,” Field Programmable Technology, 2006. FPT 2006, IEEE International Conference on, pp. 253-256, Dec. 2006.

Colohan, C.B., Ailamaki, A.C., Steffan, J.G., and Mowry, T.C. 2007. “CMP Support for Large and Dependent Speculative Threads,” Parallel and Distributed Systems, IEEE Transactions on, vol. 18, no. 8, pp. 1041-1054, Aug. 2007.

Cray X1 2004, “Cray X1 Evaluation Status Report,” Cray Inc, 2004.

Dally, W.J. 1992. “Virtual-channel flow control,” Parallel and Distributed Systems, IEEE Transactions on, vol. 3, no. 2, pp. 194-205, Mar 1992.

Dunigan, T.H., Vetter, J.S., White, J.B., and Worley, P.H. 2005, “Performance evaluation of the Cray X1 distributed shared-memory architecture,” Micro, IEEE, vol. 25, no. 1, pp. 30-40, Jan.-Feb. 2005.

Eggers, S., Emer, J., Levy, H., Lo, J., Stamm, R., and Tullsen, D. 1997. “Simultaneous multithreading: A platform for next-generation processors,” IEEE Micro, vol. 17, no. 5, pp. 12-19, September 1997.

Esmaeilzadeh H., Blem E., Amant R.S., Sankaralingam K., and Burger D. 2011. "Dark silicon and the end of multicore scaling," In Proceeding of the 38th annual international symposium on Computer architecture (ISCA '11), ACM, New York, NY, USA, pp. 365-376, 2011.

Espasa, R., Valero, M. 1997. "Multithreaded vector architectures," High-Performance Computer Architecture, Third International Symposium on, pp. 237-248, 1-5 Feb 1997.

Federal HPC Rep 2004. "High-end computing revitalization task force - Federal Plan for High-End Computing," Technical report, Executive Office of the President, Office of Science and Technology Policy, May 2004. Report of the High-End Computing Revitalization Task Force.

Flachs, B., Asano, S., Dhong, S.H., Hofstee, H.P., Gervais, G., Kim, R., Le, T., Liu, P., Leenstra, J., Liberty, J., Michael, B., Oh, H.J., Mueller, S.M., Takahashi, O., Hatakeyama, A., Watanabe, Y., Yano, N., Brokenshire, D.A., Peyravian, M., Vandung T., and Iwata, E. 2005. "The microarchitecture of the synergistic processor for a Cell processor", ISSCC 2005 Digest of Technical Papers, Feb. 2005, pp. 134-135, 2005.

Frigo, M., and Johnson, S. G. 2005. "The design and implementation of FFTW3," In Proceedings of the IEEE, vol. 93, no. 2, pp. 216-231, 2005.

Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., and Tullsen, D.M. 1997. "Simultaneous multithreading: a platform for next-generation processors," IEEE Micro, vol. 17, no. 5, pp. 12-19, September 1997.

Gerneth, F. 2010. "FIR filter algorithm implementation using Intel SSE instructions: optimizing for Intel Atom architecture," Software White Paper on Intel Embedded Design Center, http://download.intel.com/design/intarch/papers/323411.pdf, (link accessed Jan. 2011).

Golub, G. H. and Van Loan, C. F. 1996. "Matrix computations 3rd Ed," Johns Hopkins, Baltimore, USA, 1996.

Green 500 2011, "The green 500 supercomputers list - November 2011," http://www.green500.org/lists/2011.

Gustafson, J. L. 1988. "Reevaluating Amdahl's law," Communications of the ACM vol. 31, no. 5, pp. 532-533, 1988.

Hagiescu, A. and Wong, W.F. 2011. "Co-synthesis of FPGA-based application-specific floating point SIMD accelerators," In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11), ACM, pp. 247-256, New York, USA, 2011.

Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., and Horowitz, M. 2011. "Understanding sources of inefficiency in general-purpose chips," Communications of the ACM, vol. 54, no. 10, October 2011.

Hiramoto T., and Takamiya, M. 2000. “Low power and low voltage MOSFETs with variable threshold voltage controlled by back-bias”, IEICE Trans. Electr., vol. E83, no. 2, pp. 161-169, 2000.

Hong, I., Kirovski, D., Qu, G., Potkonjak, M., and Srivastava, M.B. 1999. “Power optimization of variable-voltage core-based systems,” IEEE Trans. Computer-Aided Design Integrated Circuits Systems, vol. 18, pp. 1702-1714, 1999.

Intel IPP 2010. “Integrated Performance Primitives for Intel architecture - Reference manual,” Intel Corp., Dec. 2010. http://software.intel.com/en-us/articles/intel-ipp.

Intel MKL 2011. “Intel Math Kernel Library reference manual,” Intel Corp., Dec. 2011. http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation.

Ishihara, S., Hariyama, M., and Kameyama, M. 2011. “A low-power FPGA based on autonomous fine-grain power gating,” Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 19, no. 8, pp. 1394-1406, Aug. 2011.

Jejurikar, R., Pereira, C., and Gupta, R.K. 2004. “Leakage aware dynamic voltage scaling for real-time embedded systems,” Design Automation Conference, pp. 275-280, 2004.

Kao, J., Narendra, S., and Chandrakasan, A. 2002. “Subthreshold leakage modeling and reduction techniques,” IEEE/ACM Int. Conf. Computer-Aided Design, NY, pp. 141-148, 2002.

Keating, M., Flynn, D., Aitken,R., Gibsons, A., and Shi, K. 2007. “Low power methodology manual for system on chip design,” Springer Publications, NewYork, USA, 2007.

Kihm J., Settle, A., Janiszewski, A., and Connors D.A. 2005. “Understanding the impact of inter-thread cache interference on ILP in modern SMT processors,” The Journal of Instruction Level Parallelism (JILP), vol. 7, June 2005.

Kim, N.S., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J.S., Irwin, M.J., Kandemir, M., and Narayanan, V. 2003 “Leakage current: Moore’s law meets static power,” Computer, vol. 36, no. 12, pp. 68-75, 2003.

Kobayashi, H., Egawa, R., Takizawa, H., Okabe, K., Musa, A., Soga, T., and Shimomura, Y. 2008. “First experiences with NEC SX-9,” in M. Resch et al., editors, High Performance Computing on Vector Systems, pp. 3-11. Springer-Verlag, 2008.

Korthikanti, V.A., and Agha, G. 2010. “Towards optimizing energy costs of algorithms for shared memory architectures,” 22nd ACM Symp. Paral. Alg. Arch., NY, 2010, pp. 157-165.

Kozyrakis, C. and Patterson, D. 2002. “Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks,” In Proceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 283-293, 2002.

Kozyrakis, C. and Patterson, D. 2003a. “Overcoming the limitations of conventional vector processors,” SIGARCH Comput. Archit. News, vol. 31, no. 2, pp. 399-409, 2003.

Kozyrakis, C. and Patterson, D. 2003b. “Scalable, vector processors for embedded systems,” IEEE Micro., vol. 23, no. 6, pp. 36-45, 2003.

Krashinsky, B., Batten, C., Hampton, M., Gerding, S., Pharris, B., Casper, J., Asanovic, K. 2004. “The vector-thread architecture,” Micro, IEEE, vol. 24, no. 6, pp. 84-90, Nov.-Dec. 2004.

Kudlur, M. and Mahlke, S. 2008. “Orchestrating the execution of stream programs on multicore platforms,” SIGPLAN, no. 43, pp. 114-124, May 2008.

Kuon, I., Rose, J. 2007. “Measuring the gap between FPGAs and ASICs,” Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, no. 2, pp. 203-215, Feb. 2007.

Kurihara, K., Chaiken, D., and Agarwal, A. 1991. “Latency tolerance through multithreading in large-scale multiprocessors,” In Proceedings of the International Symposium on Shared Memory Multiprocessing, pp. 91-101, April 1991.

LaForest, C.E., and Steffan, J. G. 2010. “Efficient Multi-Ported Memories for FPGAs,” In Proceedings of 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 41-50, ACM, Monterey, CA, 2010.

Laudon, J., Gupta, A., and Horowitz M. 1994. “Interleaving: a multithreading technique targeting multiprocessors and workstations,” In Proceedings of the sixth international conference on Architectural support for programming languages and operating systems (ASPLOS-VI), pp. 308-318, ACM, New York, NY, USA, 1994.

Lemuet, C., Sampson, J., Francois J., and Jouppi, N. 2006. “The potential energy efficiency of vector acceleration,” SC 2006 Conference, Proceedings of the ACM/IEEE, pp. 73-90, Nov. 2006.

Lee, Y., Avizienis, R., Bishara, A., Xia, R., Lockhart, D., Batten, and Asanovic, K. 2011. “Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators,” In Proceeding of the 38th annual international symposium on Computer architecture (ISCA '11), pp. 129-140, ACM, New York, NY, USA, 2011.

Leverich, J., Monchiero, M., Talwar, V., Ranganathan, P., and Kozyrakis, C. 2009. “Power management of datacenter workloads using per-core power gating,” IEEE Comput. Archit. Lett., vol. 8, no. 2, pp. 48-51, July 2009.

Li J., and Martinez, J.F. 2005. “Power-performance considerations of parallel computing on chip multiprocessors,” ACM Trans. Arch. Code Optim., pp. 397-422, Dec. 2005.

Lin, Y., Lee, H., Woh, M., Harel, Y., Mahlke, S., Mudge, T., Chakrabarti, C., and Flautner, K. 2006. ”SODA: A low-power architecture for software radio,” In Proceedings of the 33rd Annual International Symposium on Computer Architecture. IEEE, pp. 89-101, Boston, MA, 2006.

Marongiu, A., and Benini, L. 2012. “An OpenMP compiler for efficient use of distributed Scratchpad Memory in MPSoCs,” in Computers, IEEE Transactions on, vol. 61, no. 2, pp. 222-236, Feb. 2012.

Martin, A.J., Nystroem, M., and Penzes, P. 2001. “ET2: A metric for time and energy efficiency of computation,” Tech. Rep. CaltechCSTR: 2001.007, Caltech Computer Science, 2001.

Matsutani, H., Koibuchi, M., Ikebuchi, D., Usami, K., Nakamura, H., and Amano, H. 2011. “Performance, area, and power evaluations of ultrafine-grained run-time power-gating routers for CMPs,” IEEE Trans. Comp.-Aided Des. Integr. Circuits Sys., vol. 30, no. 4, pp. 520-533, 2011.

McKeown, N., Mekkittikul, A., Anantharam, V., and Walrand, J. 1999. “Achieving 100% throughput in an input-queued switch,” Communications, IEEE Transactions on, vol. 47, no. 8, pp. 1260-1267, Aug 1999.

Milidonis, A., Porpodas, V., Alachiotis, N., Kakarountas, A.P., Michail, H., Panagiotakopoulos, G., Goutis, C.E. 2009. “Low-power architecture with scratch-pad memory for accelerating embedded applications with run-time reuse,” Computers & Digital Techniques, IET, vol. 3, no. 1, pp. 109-123, January 2009.

Mtx Market 2007, “Matrix market,” http://math.nist.gov/MatrixMarket/ (link accessed June 2011).

Muralimanohar, N., Balasubramonian, R., and Jouppi, N. 2007. “Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0,” In Proceedings of the 40th Annual International Symposium on Microarchitecture, pp. 3-14, December 2007.

Muralimanohar, N., Balasubramonian, R., and Jouppi, N. 2012. “CACTI 6.0: A Tool to Understand Large Caches,” Technical Report, HP Laboratories, 2012.

Musa, A. 2009. “High performance memory architecture for vector processors,” PhD Thesis, Tokyo Univ., January 16, 2009.

Nickolls, J., Dally, W.J. 2010. “The GPU computing era," Micro, IEEE, vol. 30, no. 2, pp. 56-69, March-April 2010.

Nvidia CUDA 2011. “NVIDIA’s next generation CUDA compute architecture: Fermi,” White Paper, Nvidia Corp., Santa Clara, 2011.

Oliker, L., Canning, A., Carter, J., Shalf, J., and Ethier, J. 2008. “Scientific application performance on leading scalar and vector supercomputing platforms,” The International Journal of High Performance Computing Applications vol. 22, no. 5, pp. 103-112, 2008.

Open Cores 2012, http://opencores.org/, March 2012.

Oplinger, J. T., Heine D. L., and Lam M. S. 1999. “In search of speculative thread-level parallelism,” In Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99). IEEE Computer Society, pp. 303-310, Washington, DC, USA, 1999.

Radeon HD5450 2010. "AMD Radeon HD5450 Series Specifications," AMD Comp. Oct. 2010. http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5450-overview/pages/hd-5450-overview.aspx#2.

Rahman, A., Das, S., Tuan, T., and Trimberger, S. 2006. "Determination of power gating granularity for FPGA fabric," Custom Integrated Circuits Conference, 2006, CICC '06, IEEE, pp. 9-12, 10-13 Sept. 2006.

Rintaluoma, T., and Silvén, O. 2010. "SIMD performance in software based mobile video coding," Embedded Computer Systems (SAMOS), 2010 International Conference on, pp. 79-85, 19-22 July 2010.

Rivoire, S., Schultz, R., Okuda, T., Kozyrakis, C. 2006. "Vector Lane Threading," Parallel Processing (ICPP 2006) International Conference on, pp. 55-64, 14-18 Aug. 2006.

Rogers, B.M., Krishna, A., Bell, G.B., Vu, K., Jiang, X., and Solihin, Y. 2009. "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09). ACM, pp. 371-382, New York, NY, USA, 2009.

Roy, S., Ranganathan, N., and Katkoori, S. 2009. "A framework for power-gating functional units in embedded microprocessors," IEEE Trans. Very Large Scale Integration Systems, vol. 17, no. 11, pp. 1640-1649, Nov. 2009.

Russell, R.M. 1978. "The CRAY-1 computer system," Commun. ACM, vol. 21, no. 1, pp. 63-72, January 1978.

Sanchez, F., Alvarez, M., Salami, E., Ramirez, A., and Valero, M. 2005. "On the scalability of 1- and 2-dimensional SIMD extensions for multimedia applications," In Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, pp. 167-176, 2005.

Sasanka, R., Adve, S.V., Chen, Y.K., and Debes, E. 2004. "The energy efficiency of CMP vs. SMT for multimedia workloads," In Proceedings of the 18th annual international conference on Supercomputing (ICS '04), pp. 196-206, ACM, New York, NY, USA.

Sasanka, R., Li M.L., Adve, S.V., Chen, Y.K, and Debes, E. 2007. "ALP: Efficient support for all levels of parallelism for complex media applications," ACM Trans. Archit., vol. 4, no. 1, Article 3, March 2007.

Satpathy, S., Dreslinski R., Ou, T., Sylvester, D., Mudge, T., and Blaauw, D. 2012 "SWIFT: A 2.1Tb/s 32x32 Self-Arbitrating Manycore Interconnect Fabric," Symposia on VLSI Technology and Circuits, Kyoto, Japan, June 2011.

Scogland, T., Lin, H., and Feng, W.-C. 2010, "A first look at integrated GPUs for green High-Performance Computing," Journal of Computer Science, Research and Development, Springer, vol. 25, no. 3-4, pp. 125-134, 2010.

Sengupta, D., and Saleh, R. 2007. "Generalized Power-Delay Metrics in Deep Submicron CMOS Designs," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, no. 1, pp. 183-189, Jan. 2007.

Sjodin, J., and Platen, C.V. 2001. “Storage allocation for embedded processors,” Compiler and Architecture Support for Embedded Computing Systems, ACM, pp. 15-23 New York, USA, 2001.

Soga, T., Musa, A., Shimomura, Y., Egawa, Y., Itakura, K., Takizawa, H., Okabe, K., and Kobayashi, H. 2009. “Performance evaluation of NEC SX-9 using real science and engineering applications,” In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09). ACM, New York, NY, USA, Article 28, 12 pages, 2009.

Sung, W., and Mitra, S. K. 1987. “Implementation of digital filtering algorithms using pipelined vector processors,” Proceedings of the IEEE. IEEE, vol. 75, no. 9, pp. 1293-1303, 1987.

Synopsys VCS-MX 2011. “VCS MX/VCS MXi User Guide vF-2011.09-SP2,” Synopsys Inc., Dec. 2011.

Synopsys DC 2011. “Design Compiler User Guide vF-2011.09-SP2,” Synopsys Inc., Dec. 2011.

Synopsys DC Optim. 2011. “Design Compiler Optimization Reference Manual vF-2011.09-SP2,” Synopsys Inc., Dec. 2011.

Synopsys PX 2011. “PrimeTime PX User Guide vF-2011.12,” Synopsys Inc., Dec. 2011.

Synopsys LP 2011. “Synopsy Low Power Flow User Guide vF-2011.09,” Synopsys Inc., Dec. 2011.

Top 500 List 2011. “Top 500 Supercomputers List,” November 2011. http://www.top500.org/

TSMC 40nm 2010. “TSMC Standard Cell Library tcbn40lpbwplvt,” TSMC Comp., 2010.

Tullsen, D.M., Eggers, S.J., and Levy, H.M. 1995, “Simultaneous multithreading: Maximizing on-chip parallelism,“ Computer Architecture Proceedings., 22nd Annual International Symposium on, pp. 392-403, 22-24 June 1995.

Udayakumaran, S., Dominguez, A., and Barua, R. 2006. “Dynamic allocation for scratch-pad memory using compile-time decisions,” ACM Trans. Embed. Comput. Syst., vol. 5, no. 2, pp. 472-511, May 2006.

Wang, W., and Mishra, P. 2011. “System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems,” IEEE Trans. Very Large Scale Integration Systems, pp. 1-9, March 2011.

Wang, Y., and Ranganathan, N. 2011. “An instruction-level energy estimation and optimization methodology for GPU,” IEEE 11th Int. Conf. Comp. Inf. Tech., pp. 621-628, Aug.-Sept. 2011.

Wang, H., Wang, P., Weldon, R.D., Ettinger, S.M., Saito, H., Girkar, M., Liao, S.S., and Shen J. 2002. “Speculative precomputation: Exploring use of multithreading technology for latency,” Intel Tech. J., vol. 6, no. 1, Feb. 14, 2002.

Wang, X., and Wang Y. 2011. "Coordinating power control and performance management for virtualized server clusters," Parallel and Distributed Systems, IEEE Transactions on, vol. 22, no. 2, pp. 245-259, Feb. 2011.

White, B. S., McKee, S., de Supinski, B., Miller B., Quinlan, D., and Schulz, M. 2005. "Improving the computational intensity of unstructured mesh applications," InProceedings of the 19th annual international conference on Supercomputing (ICS '05). ACM, pp. 341-350, New York, USA, 2005.

Who, M., Satpathy, S., Dreslinski, R., Kershaw, D., Sylvester, D., Blaauw, D., and Mudge, T. 2011. "Low power interconnects for SIMD computers," Proceedings Design, Automation and Test in Europe, DATE 11, Grenoble, France, pp. 600-605, March 2011.

Woh, M., Lin, Y., Seo, S., Mudge, T., and Mahlke, S. 2008. "Analyzing the scalability of SIMD for the next generation software defined radio," Acoustics, Speech and Signal Processing, 2008 (ICASSP 2008) IEEE International Conference on, pp. 5388-5391, March 31 2008-April 4 2008.

Woh, M., Seo, S., Mahlke, S., Mudge, T., Chakrabarti, C., and Flautner, K. 2010. "AnySP: Anytime anywhere anyway signal processing" IEEE Micro., vol. 30, no. 1, pp. 81-91, 2010.

Woo, D.H., and Lee, H.-H.S. 2008. "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era," IEEE Computer, vol. 41, no. 12, pp. 24-31, Dec. 2008.

Xilinx Inc. 2010a. "XPower Estimator User Guide. Xilinx," www.xilinx.com/support/documentation /user_guides (link accessed July 2011).

Xilinx Inc. 2010b. "MicroBlaze Processor Reference Guide, 2008," http://www.xilinx.com/support/ documentation/sw_manuals/mb_ref_guide.pdf (link accessed Sept. 2011).

Yang, H., and Ziavras S. 2005. "FPGA-based vector processor for algebraic equation solvers," In Proceedings of IEEE International Systems-On-Chip Conference. IEEE, pp. 115-116, Herndon, VA, 2005.

Yiannacouras, P., Steffan, J.G., and Rose, J. 2008. "VESPA: Portable, scalable, and flexible FPGA-based vector processors," In Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems. ACM, pp. 145-166, Atlanta, GA, 2008.

Yiannacouras, P., Steffan, J.G., and Rose, J. 2009. "Data parallel FPGA workloads: Software versus hardware," Field Programmable Logic and Applications, International Conference on, pp. 51-58, Aug. 31-Sept. 2 2009.

Yu, J., Eagleston, C., Chou, C. H.-Y., Perreault, M., and Lemieux, G. 2009. "Vector processing as a soft processor accelerator," ACM Transactions on Reconfigurable Technology and Systems, vol. 2, no. 2, Article 12, 34 pages, June 2009.

Yuffe, M., Knoll, E., Mehalel, M., Shor, J., Kurts, T. 2011 "A fully integrated multi-CPU, GPU and memory controller 32nm processor," Solid-State Circuits

Conference Digest of Technical Papers (ISSCC), 2011 IEEE International, pp. 264-266, 20-24 Feb. 2011.

Xilinx 2009. “Xilinx white paper: Power consumption at 40 and 45 nm”, Xilinx Inc. 2009.

Xilinx SDK 2011. “Xilinx Software Development Kit Help,” Xilinx Inc., 2011. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/SDK_doc/index.html (link accessed Dec. 2011).

Xilinx 2011. “Virtex-6 family overview,” Xilinx Inc., 2011. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf (link accessed Dec. 2011).

Xilinx MicroBlaze 2011. “MicroBlaze processor reference guide Embedded Development Kit EDK 12.3,” Xilinx Inc., 2011.