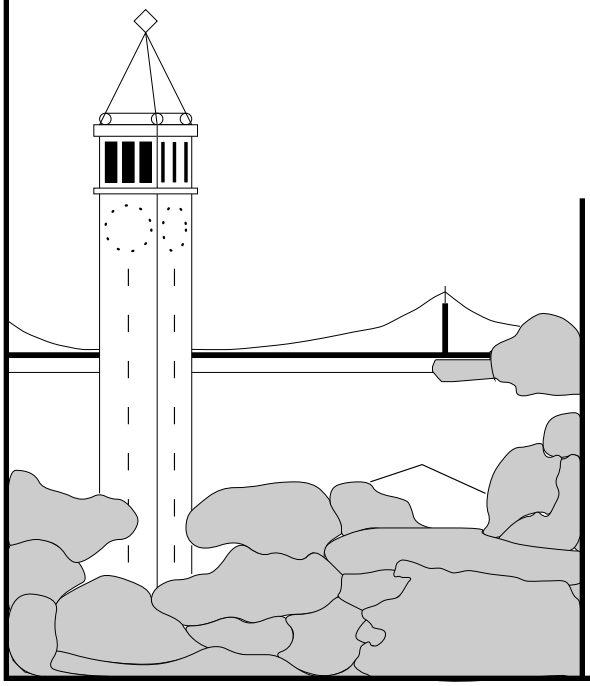
Scalable Vector Media-processors for Embedded Systems

Christoforos Kozyrakis



Report No. UCB/CSD-02-1183

May 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Scalable Vector Media-processors for Embedded Systems

by

Christoforos Kozyrakis

Grad. (University of Crete, Greece) 1996
M.S. (University of California at Berkeley) 1999

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David A. Patterson, Chair
Professor Katherine Yelick
Professor John Chuang

Spring 2002

Scalable Vector Media-processors for Embedded Systems

by
Christoforos Kozyrakis

Abstract

Scalable Vector Media-processors for Embedded Systems

by

Christoforos Kozyrakis
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David A. Patterson, Chair

Over the past twenty years, processor designers have concentrated on superscalar and VLIW architectures that exploit the instruction-level parallelism (ILP) available in engineering applications for workstation systems. Recently, however, the focus in computing has shifted from engineering to multimedia applications and from workstations to embedded systems. In this new computing environment, the performance, energy consumption, and development cost of ILP processors renders them ineffective despite their theoretical generality.

This thesis focuses on the development of efficient architectures for embedded multimedia systems. We argue that it is possible to design processors that deliver high performance, have low energy consumption, and are simple to implement. The basis for the argument is the ability of vector architectures to exploit efficiently the data-level parallelism in multimedia applications. Furthermore, the increasing density of CMOS chips enables the design of cost-effective, on-chip, memory systems that can support the high bandwidth necessary for a vector processor.

To test our hypothesis, we present VIRAM, a vector architecture for multimedia processing. We demonstrate that the vector instructions in VIRAM can capture the data-level parallelism in multimedia tasks and lead to smaller code size than RISC, CISC, and VLIW architectures. We also describe two scalable microarchitectures for vector media-processors: VIRAM-1 and CODE. VIRAM-1 integrates a simple, yet highly parallel, vector processor with an embedded DRAM memory system in a prototype chip with 120 million transistors. CODE uses a composite and decoupled organization for the vector processor in order to simplify the vector register file design, tolerate high memory latency, and allow for precise exceptions support. Both microarchitectures provide up to 10 times higher performance than alternative approaches without using out-of-order or wide instruction issue techniques that exacerbate energy consumption and design complexity.

To my parents,
Litsa and Manolis,
for their unconditional love and support.

Στους γονεις μου,
Λιτσα και Μανωλη,
για την απεριοριστη αγαπη και στηριξη τους.

Contents

List of Figures

List of Tables

Acknowledgements

Research in computer architecture requires a team effort. During the course of my studies, I was fortunate to work with a number of great people that influenced the direction and the quality of my work.

First, I would like to thank Dave Patterson, my thesis advisor, for his overall guidance, support, and friendship. He helped me develop a taste for research and encouraged me to pursue the problems and ideas I found most intriguing. Moreover, he showed me the importance of a balanced professional and personal life. I also enjoyed our frequent conversations on soccer, a rare delight for a European living in California.

I am especially grateful to Manolis Katevenis, my undergraduate advisor at the University of Crete, for initiating me into computer architecture. His enthusiastic teaching and good advice motivated me to pursue a graduate degree in the first place. Krste Asanovic, now a professor at MIT, introduced me to vector architectures and their potential with multimedia applications. Our numerous discussions had a strong influence on the microarchitecture and design of VIRAM-1. I also want to thank professors Katherine Yelick, John Wawrzynek, John Kubiatowicz, Christos Papadimitriou, and John Chuang at U.C. Berkeley for their help and feedback at various stages of my studies.

I am indebted to Sam Williams and Joe Gebis, my project partners and officemates. Without their persistence and selflessness, the VIRAM-1 prototype chip would not have been possible. They also had to deal with my stubborn character and Greek accent on a daily basis. I hope they enjoyed our long arguments on social, political, and economical issues as much as I did. By the way, we never agreed on anything. My gratitude to all the graduate students and staff involved with hardware or software development for the IRAM project: David Martin, Iakovos Mavroidis, Ioannis Mavroidis, Hiro Hamasaki, Brian Gaeke, Dave Judd, Rich Fromm, and Mani Narayanan. I also want to acknowledge the help from Paul Gutwin, Bill Tetzlaff, and Subu Iyer from IBM, as well as from Darren Jones from MIPS Technologies.

My favorite part of graduate school was the open-ended discussions on research and educational issues that I had with numerous people in the EECS department. John Hauser, Tim Callahan, Kees Visser, Jim Beck, David Oppenheimer, Eylon Caspi, Joe Yeh, and Kim Keeton facilitated several insightful conversations and should be blamed for all the time I wasted talking in the corridors of Soda Hall instead of working.

Special thanks to Iason Vasiliou for being a truly good friend during my six years in Berkeley. He often had to force me to have some fun and forget about the hard work and stress of graduate school. Maria Moreno, Kostas Adam, Nikos Chronis, Stelios Perissakis, Manolis Terrovitis, Regina Soufli, Kristina Varga, Pamela Erickson, Amanda Cramp, and all the players of the Tzatziki Turbos, the soccer team of Greek students in U.C. Berkeley, have also been great friends and good companions as well. As for the last year of my studies, it was Vicky Kalivitis that brought me the peace of mind necessary to conclude my work.

Finally, I want to thank my parents, Litsa and Manolis, and my two beautiful sisters, Maria and Natasa, for more than I can say with words.

The IRAM project at U.C. Berkeley was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract DABT63-96-C-0056, by the California State MICRO Program, and by the Department of Energy. IBM, MIPS Technologies, Cray Inc., and Avanti Corp. have made significant software or hardware contributions to the IRAM project. This thesis was also supported by a U.C. Regents fellowship (1996-97) and an IBM Research Ph.D. fellowship (2001-02).

Chapter 1

Introduction

"There is at the back of every artist's mind,
a pattern or type of architecture."
Gilbert Chesterton

Over the past twenty years, microprocessor designers have concentrated on accelerating engineering applications on workstation systems. This approach has lead to development of the superscalar and VLIW architectures for exploiting the instruction-level parallelism (ILP) available in applications with complex control flow. ILP processors have taken advantage of the exponential improvements in the density and speed of circuits in semiconductor chips in order to deliver exponentially increasing performance.

Recently, however, the focus in computing has shifted from engineering to multimedia applications and from workstations to embedded systems. In this new computing environment, the energy consumption and design complexity of ILP architectures renders them ineffective, despite their theoretical generality and flexibility. Moreover, it is becoming gradually more difficult for ILP processors to translate improvements in circuits technology to proportional gains in application performance.

This thesis focuses on the development of efficient microprocessors for embedded multimedia systems. We argue that it is possible to design processors that deliver high performance for multimedia tasks, have low energy consumption, and are simple to implement and scale with modern CMOS technology. The basis for the argument is the existence of data-level parallelism in multimedia applications and the ability to exploit it efficiently with a vector architecture. Furthermore, the increasing density of semiconductor dies enables the design of cost-effective, on-chip, memory systems that can support the high bandwidth necessary for a vector processor.

Thesis Contributions

The main contributions of this dissertation are the following:

- We introduce the VIRAM vector instruction set architecture (ISA) for embedded multimedia systems. The vector instructions in the VIRAM ISA can express the data-level parallelism in multimedia application in an explicit and compact manner.

- We present the microarchitecture, design, and evaluation of the VIRAM-1 media-processor. VIRAM-1 integrates a simple, yet highly parallel, vector processor with an embedded DRAM memory system. It demonstrates that a vector processor can provide high performance for multimedia tasks, at low energy consumption, and low design complexity.

- We propose the CODE vector microarchitecture for the VIRAM ISA that combines composite organization with decoupled execution. The simplified vector register file and the ability to tolerate high memory latency allow CODE to extend the performance and energy advantages of VIRAM-1 across a larger design space. It can also support precise exceptions with a minimal impact on performance.

- We demonstrate that embedded DRAM is a suitable technology for the memory system of vector media-processors. Embedded DRAM provides the high memory bandwidth required by a vector processor at low energy consumption and moderate access latency.

Thesis Outline

The outline of the rest of this thesis is as follows.

Chapter 2 provides the background and motivation for this work. It discusses the characteristics and requirements of multimedia applications and embedded systems. It also argues that the superscalar and VLIW architectures for high performance processors are not suitable for embedded multimedia systems.

Chapter 3 describes the function and characteristics of the EEMBC benchmark suite for embedded processors. We focus mostly on the consumer and telecommunications benchmarks, which are representative of multimedia tasks.

Chapter 4 introduces the VIRAM vector instruction set. It describes a set of enhancements to traditional vector architectures that provide support for multimedia processing and virtual memory. We demonstrate that the vector instructions in VIRAM can capture more than 90% of the dynamic instruction count of the EEMBC benchmarks. We also show that the use of vector instructions leads to significant advantages in terms of code size over CISC, RISC, and VLIW architectures.

Chapter 5 presents and evaluates the VIRAM-1 prototype microprocessor. We describe the VIRAM-1 pipeline structure and how it interacts with embedded DRAM. We also present its scalable design based on the concept of vector lanes. We demonstrate that VIRAM-1 outperforms superscalar and VLIW processors by at least a factor of 2, despite its low clock frequency due to its focus on low energy consumption.

Chapter 6 introduces the CODE microarchitecture for vector media-processors. We discuss its two basic elements, composite organization and decoupled execution, and how they allow for performance, energy, and complexity improvements over VIRAM-1. We describe the issue logic and operation control in CODE and discuss its implementation based on two orthogonal concepts: vector cores and vector lanes. Finally, we use the EEMBC benchmarks to derive the optimal value for the key parameters of the CODE issue logic.

Chapter 7 tackles the problem of precise virtual memory exceptions in vector microprocessors. We introduce an alternative definition for precise exceptions for vector instructions that places relaxed requirements on processor implementations. We also describe a set of minor modifications to the issue logic of CODE that implements precise exceptions. We demonstrate that the support for precise virtual memory exceptions has negligible impact on performance for the multimedia benchmarks.

Chapter 8 examines the use of embedded DRAM for the memory system of vector media-processors. We introduce the basics of embedded DRAM technology and discuss the impact on performance and energy consumption of various design parameters such as the number of banks and sub-banks, the address interleaving scheme, and the type of processor to memory interconnect. We demonstrate that the CODE microarchitecture works well with embedded DRAM as it can tolerate high memory latency if sufficient bandwidth is available. We also show that the use of multiple banks and sub-banks in the memory system is crucial, especially for applications with non sequential access streams.

Chapter 9 provides a detailed performance evaluation of CODE for the EEMBC benchmarks. Assuming equal die area, CODE outperforms VIRAM-1 by 26% for the consumer benchmarks and 12% for telecommunications benchmarks. We also demonstrate that CODE can exploit additional hardware resources by operating on independent vector instructions on multiple vector cores or by executing several element operations for each vector instruction in parallel on multiple vector lanes.

Finally, Chapter 10 concludes the thesis and suggests directions for future work.

Chapter 2

Background and Motivation

> "One's mind has a way of making itself up in the background,
> and it suddenly becomes clear what one means to do."
> *Arthur Benson*

The main drivers for microprocessor technology have traditionally been workstation and server systems running engineering applications. The requirements of such systems have lead to the development of the superscalar and VLIW architectures that exploit the instruction-level parallelism (ILP) available in applications with complex control flow. In addition, the exponential growth in CMOS semiconductor technology has allowed the design of faster, larger, and increasingly complicated processors. In the last few years, however, we have experienced significant changes in the requirements and underlying assumptions for general-purpose microprocessors. In this chapter, we discuss the applications, systems, and technology trends that set the background and motivate the research work in this thesis.

Section 2.1 presents the requirements of two emerging trends in computing: multimedia applications and embedded systems. It also discusses the semiconductor technology and manufacturing challenges that threaten to limit the potential of future processor designs. Section 2.2 argues that superscalar and VLIW processors do not match the requirements of embedded multimedia in deep sub-micron CMOS technology. Finally, Section 2.3 introduces the basic requirements for efficient microprocessors for embedded multimedia systems and sets the research goals for this thesis.

2.1 Applications, Systems, and Technology Trends

To develop successful microprocessor architectures, we must examine carefully the requirements of the applications that the processors will run and the characteristics of the computer systems that the processors will go into. In addition, we should keep in mind the capabilities and, most important, the limitations of the underlying manufacturing technology for semiconductor chips.

2.1.1 Multimedia Applications

The continuing improvements in circuits technology and recent algorithmic innovations have enabled the use of real-time, media data such as video, sound, and animation. Applications with multimedia features such as 3-D graphics, video or visual imaging, speech or handwriting recognition, and high fidelity music, are already among the most popular and consume the majority of processing cycles on desktop systems. They have the ability to greatly improve the usability, quality, productivity, and enjoyment of computer systems. They also expand the applicability of computer-based products from the office environment to every aspect of our lives. Hence, it is

common knowledge that the influence of multimedia applications on computing will only increase in the future [BG97, Gro98, Dal98, Kil98].

Multimedia applications exhibit a set of distinguishing characteristics [DD97]:

- Data-level parallelism is inherent in multimedia programs as they typically repeat a small set of operations over a sequence of input pixels, video frames, or sound samples. This form of parallelism is explicit in the algorithmic description of multimedia functions.
- They operate mostly on narrow data types, as 8-bit or 16-bit numbers are sufficient to encode the limited input range of human vision and hearing.
- They require real-time response guarantees. Most multimedia applications rely on real-time qualitative perception. Hence, the sustained performance under worst-case conditions is much more important than the peak performance or the accuracy of arithmetic results. With video decoding, for example, the rate of 30 frames per second defines the minimum acceptable and maximum required performance level. It is preferable to produce a few erroneous pixels per frame rather than drop bellow the required frame rate at any point in time.
- Due to the streaming nature of multimedia applications, their input data exhibit limited temporal locality.

In contrast, most engineering workloads use 64-bit numbers for maximum arithmetic accuracy and demonstrate good temporal and spatial locality on their data accesses. Their complex control flow includes short data dependence distances that limit the potential for extracting data-level parallelism. Since the goal with engineering applications is minimum time to completion, they place emphasis on peak performance over real-time response guarantees.

The apparent differences in the characteristics of engineering and multimedia applications indicate that a processor optimized for the former is unlikely to be efficient for the latter.

2.1.2 Embedded Systems

In parallel with the appearance of multimedia applications, the focus in systems development has been switching from the desktop to the embedded domain. Embedded systems include portable devices such as personal digital assistants (PDAs), digital cameras, palmtop computers, and cellular phones, as well as entertainment systems such as video game consoles, set-top boxes, and DVD players [Lew98]. In the last few years, there has been a rapid growth in the variety and functionality of such embedded consumer products, driven mostly by their huge market potential. Even though one or two desktop computers are sufficient for most households, a single person may own and use several embedded devices.

The characteristics and requirements of embedded systems are considerably different from those of desktop machines:

- They require low energy consumption. Portable devices must operate for a long time using conventional battery technology. In addition, electronics for embedded devices are limited to cheap cooling systems and packages.
- Embedded systems store application code in some form of non-volatile memory like ROM or Flash because hard disks are too expensive for most consumer products. Compact code size lowers the system cost because the application can use a smaller ROM or Flash chip.
- Due to the consumer nature of embedded products, they call for low development and manufacturing cost. Hence, the electronic components for embedded systems must be easy to design in the first place and easy to scale for follow-up products.

- To reduce the overall size for embedded devices, it is desirable to use highly integrated chips that incorporate a large number of the processing, memory, and IO components on a single die.

As desktop and workstation systems become more cost-oriented, some of the requirements of embedded systems become more general. However, the conventional wisdom has been that power consumption is a secondary issue for desktop systems. Code size has also been irrelevant due to the availability of hard disks in PCs and workstations and the success of instruction caches with the code for engineering applications.

2.1.3 Technology Constraints

Until recently, it was widely believed that the continuing validity of Moore's law [Moo65] meant that we can proceed with the design of larger and increasingly complex processors without significant concerns about the underlying technology. Even though the capacity of semiconductor chips still grows at the exponential rate predicted in Moore's law, there are certain limitations that could prohibit the conversion of increased capacity to increased performance in future microprocessors.

The first technology constraint is the exponentially increasing performance gap between microprocessors and DRAM memory [HP02]. While processor chips have been optimized for performance, DRAM chips have been targeting maximum capacity and minimum cost. The consequence is that main memory accesses are becoming increasingly slower from the processor's perspective. No matter how fast the processor can execute arithmetic operations or how many operations it can execute in parallel, it cannot deliver a high level of sustained performance if it cannot access fast the input and output data for the applications [WM95]. We provide a further discussion of the processor-memory performance gap in Chapter 8.

An additional problem of semiconductor technology is the latency of long, on-chip wires. As the feature size of CMOS processes shrinks, transistors and logic gates become faster. However, as we also increase the amount of hardware resources in microprocessor chips, the propagation latency of long wires that implement cross-chip communication remains constant [HMH01]. Therefore, the relative cost of computation versus communication decreases with every generation of CMOS technology and global interconnect patterns within a processor chip are becoming progressively more expensive. The consequence for processor architectures that rely on global, low latency communication of their subcomponents will be either slower clock frequency or repeated stalls during global communication events.

The final technology constraint refers to processor development costs. As the capacity and functional complexity of microprocessors increase at exponential rates, the same holds for their design and verification complexity. For complicated architectures with limited component modularity, the design and verification costs and cycles can easily exceed the manufacturing costs and cycles [AEJ+02]. Complicated designs require large development teams that are expensive assemble and difficult to manage. They also place a heavy burden on the CAD tools for automated design and the equipment for testing semiconductor products.

2.2 The Case against Superscalar and VLIW Processors for Embedded Multimedia Processing

The prevailing approaches to high performance processors have been the superscalar and VLIW architectures that exploit the instruction-level parallelism available in applications with wide-issue organizations [SS95, Fis83]. These architectures have delivered exponential growth in performance over the past two decades. Many expect that the performance of ILP processors will keep improving at the current rate indefinitely and that superscalar and VLIW designs can provide an

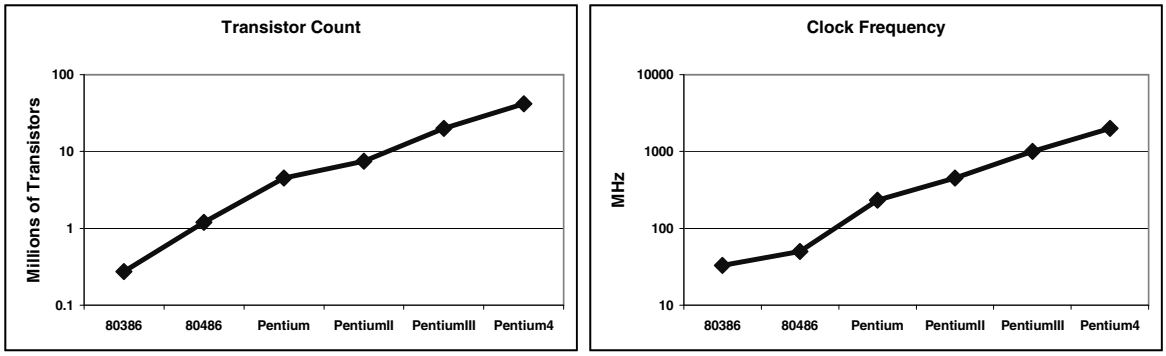


Figure 2.1: The evolution of the number of transistors and the clock frequency across six generations of x86 microprocessor by Intel. Both graphs use a logarithmic scale for the y axis. The four Pentium processors use superscalar organizations and are able to issue and execute more than one instruction per clock cycle.

efficient computing platform for all applications and all systems. In this section, we argue that ILP processors are running out of steam. They are increasingly inefficient even for the engineering applications they were developed for. Moreover, they are poor matches to the requirements of multimedia applications running on embedded systems.

Figure 2.1 presents the evolution of the number of transistors and the clock frequency in six generations of superscalar microprocessors from Intel. As the feature size for CMOS technology has shrunk, we have been able to increase the number of transistors in microprocessor chips by a factor of 2.8 per generation. The additional hardware resources enabled the increase in the number of instructions issued and executed per cycle by using additional functional units, larger caches, and larger branch predictors. At the same time, faster gates, better circuit techniques, and deeper pipelines have allowed clock frequency to increase by a factor of 2.2 per generation. Each generation can execute basic operations twice as fast as the previous one. Therefore, one would expect an overall performance improvement for superscalar processors by a factor of approximately 6.1 (2.8 x 2.2) per generation.

Figure 2.2 presents the performance of the same microprocessors as measured with the integer SPEC benchmarks for engineering workloads. Sustained performance has increased by nearly 300 times, which implies an improvement factor of 3.1 per generation. This is half of the expected rate. In addition, the rate of improvement has dropped to approximately 2.0 for the three most recent microarchitectures (Pentium II to Pentium 4). Figure 2.2 also presents the evolution of power consumption. Despite the feature shrinks and the continuous reductions in the power supply voltage, power dissipation has been doubling with each generation and is approaching fast the dissipation limit for air-cooled systems.

The conclusion from Figures 2.1 and 2.2 is that superscalar processors are becoming increasingly ineffective with turning increased circuits capacity and high clock frequency into sustained performance, even for the engineering workloads they have been optimized for. As the amount of instruction-level parallelism in applications is inherently limited [Wal91a] and difficult to extract [AHBK00], scaling superscalar designs requires expensive investments in die area, circuits design, and engineering effort for diminishing returns in performance. The same holds for VLIW processors since they also rely on instruction-level parallelism.

The picture for ILP processor becomes even worse if we consider the requirements of multimedia applications. Superscalar processors use a strictly sequential instruction stream that hides any data-level or instruction-level parallelism. In order to discover any parallelism and exploit it for higher performance, the hardware must use complicated issue logic that is wasteful in terms of

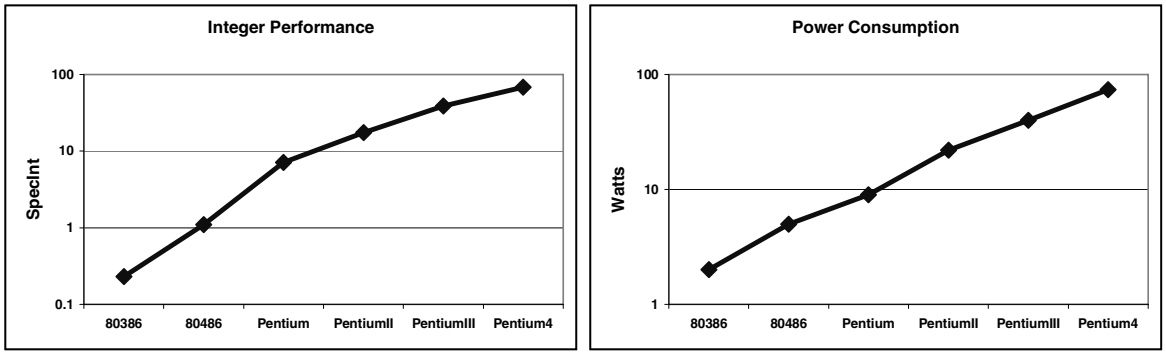


Figure 2.2: The evolution of integer performance and the power consumption across six generations of x86 microprocessors by Intel. Both graphs use a logarithmic scale for the y axis. Integer performance refers to the score for the integer benchmarks in the SPEC suite.

both hardware resources and energy consumption. VLIW processors can capture some amount of data-level parallelism in the long instructions at the cost of increased code size (see Chapter 4).

Both superscalar and VLIW processors typically provide limited support for narrow data types. They also use hierarchies of caches in order to bridge the processor-memory performance gap. However, caches rely on the premise of temporal locality, which is not available in the streaming data for multimedia applications. Superscalar and VLIW processors rely on probabilistic techniques such as caching, hardware or software speculation, and out-of-order execution. Such techniques make it difficult to provide guarantees for real-time response, as there is a large impact on performance associated with cache misses and incorrect speculation.

ILP processors are poor matches for embedded systems as well. Superscalar processors consume excessive amounts of energy to discover which instructions in a sequential stream are independent and can execute in parallel. VLIW processors, on the other hand, consume energy on fetching and decoding instructions from bloated executables. They both waste energy on cache references for multimedia applications with limited temporal locality.

ILP processors rely on global communication patterns for instruction issue and result forwarding. In most cases, propagation of critical signals across the chip must occur in a single clock cycle. This property makes superscalar and VLIW designs vulnerable to the latency problem of long wires in CMOS technology.

Finally, high performance designs for superscalar and VLIW processor suffer from increased design complexity. The design nonrecurring expenses (NRE) for such microprocessors routinely reach the tens of millions of dollars [AEJ+02]. In addition, their design and verification cycles are measured in years despite the use of hundreds of engineers. For reference, the manufacturing NRE for mask production and probe-card design are just reaching the $1 million mark and manufacturing cycle times for microprocessor chips are measured in weeks. The trend of increasing complexity is expected to continue for superscalar and VLIW designs as they implement wider and increasingly complicated control functions.

2.3 Research Goals

Given the applications, systems, and technology trends, we consider the following as the fundamental characteristics for an efficient processor architecture for embedded multimedia systems:

- It expresses the **data-level parallelism** in multimedia applications in an **explicit and compact** way.

- It exploits data-level parallelism to deliver **high performance at low energy consumption**.
- It provides sufficient performance under worst-case conditions in order to simplify guarantees for **real-time response**.
- It leads to **modular hardware implementations** with mostly local interconnect that are easy to design, verify, and scale.
- It is easy to program efficiently using **high-level languages and compilers**.

The above features define the basic research goals for the architecture and microarchitecture techniques presented in the following chapters.

Chapter 3

Multimedia Benchmarks

"In theory, there is no difference between theory and practice.
But, in practice, there is."
Jan van de Snepscheut

The development and evaluation of a new processor microarchitecture is impossible without measuring its efficiency for a set of applications. Ideally, we would like to evaluate the processor using full-size, end-user applications running within the environment of a complete product. Such an evaluation is rarely possible however, especially during the development stages of a new microarchitecture. The chip-level simulators used to explore, tune, and verify a proposed design are too slow to allow experimentation with complete applications running on top of an operating system. Furthermore, we typically design new processors for the next generation of user software, hence the exact characteristics of the applications are not always available during development. Therefore, processor designers evaluate and compare processors using a benchmark suite: a collection of short programs that represent the applications of interest and stress the key processor features necessary to run them efficiently. This chapter describes the benchmarks used in this thesis to develop and evaluate a set of microarchitectures for vector media-processors for embedded systems.

Section 3.1 introduces the EEMBC benchmark suite for embedded processors. Section 3.2 describes the kernels included in the consumer and telecommunications categories of the EEMBC suite, which we use in the rest of this thesis. Section 3.3 discusses some basic characteristics of the EEMBC benchmarks, their advantages and shortcomings. Finally, Section 3.4 presents related work in the area of benchmarking embedded and multimedia processor architectures.

3.1 The EEMBC Benchmark Suite

The EEMBC (EDN Embedded Microprocessor Benchmark Consortium) suite is a collection of benchmarks for the evaluation of microprocessors for a wide range of embedded applications [Hal99b]. Table 3.1 summarizes the benchmarks included in the suite. The goal of EEMBC is to become the "yardstick" for embedded processors in a similar way that the SPEC [Hen00] and TPC [PF00] suites are the standard benchmarks for desktop and server processors respectively.

The field of embedded processors is extremely diverse as it includes anything from 8-bit and 16-bit, low-cost microcontrollers to powerful, 32-bit and 64-bit microprocessors. There are more than ten instruction sets in use with embedded processors today and, even though some are more popular than the rest, there is no dominant approach. Furthermore, there is no dominant microarchitecture. One can find an embedded chip that implements any possible hardware technique: RISC, CISC, VLIW, SIMD, and MIMD [HP02]. The fundamental reason for the diversity is the wide assortment of applications for embedded processors, ranging from industrial control systems to cellular phones

Consumer Category	
RGB to CMYK Conversion	RGB to YIQ Conversion
High-pass Gray-scale Filter	JPEG Compress
JPEG Decompress	
Telecommunications Category	
Autocorrelation	Convolutional Encoder
Bit Allocation	Fast Fourier Transform
Viterbi Decoder	
Networking Category	
Dijkstra Routing	Packet Flow
Patricia Table Lookup	
Office Automation Category	
Bezier Curve Calculation	Dithering
Image Rotation	
Automotive & Industrial Category	
Table Lookup & Interpolation	Tooth to Spark
Angle to Time Conversion	Pulse-width Modulation
Remote Data Request	Road-speed Calculation
Infinite Impulse Response Filter	Finite Impulse Response Filter
Bit Manipulation	Basic Arithmetic
Pointer Chasing	Matrix Arithmetic
Cache Buster	Inverse Discrete Cosine Transform
Fast Fourier Transform	

Table 3.1: The five categories in the first release of the EEMBC suite and the benchmarks they include. All benchmarks are coded in C. They can execute on a wide range of embedded processors and microcontrollers. This thesis focuses on the consumer and telecommunications categories.

and digital cameras. Each domain of embedded applications has different requirements in terms of performance, power consumption, and cost. The high sales volume of embedded products motivates vendors to develop processors customized to the needs of each specific domain.

To allow for fair comparisons within the diverse space of embedded processors, the EEMBC suite includes five benchmark categories, one for each major embedded application domain:

- The **consumer** category includes algorithms used in digital-cameras, set-top-boxes, and personal digital assistants (PDAs).
- The **telecommunications** class contains basic kernels from modem, ADSL, and wireless applications.
- The **networking** group features workloads from network devices such as switches and routers.
- The **office automation** category includes functions that represent office machinery such as printers, fax machines, and word processors.
- The **automotive & industrial** class contains tasks derived from industrial controllers and automotive applications such as engine and airbag control.

All benchmarks are coded in standard C and include multiple reference sets of input and output data.

The main metric for the EEMBC suite is execution throughput: the number of times a processor can repeat a specific benchmark within one second (iterations/second). Higher throughput scores mean higher performance. EEMBC reports throughput individually for each benchmark. It also summarizes each category with a composite metric, which is proportional to the geometric mean of the individual benchmark scores. An additional metric is the static code and static data sizes in bytes. Naturally, smaller code and data sizes are preferable. The static data size does not include dynamically allocated memory. However, static memory use is a representative metric for embedded applications, because embedded programs typically allocate all the necessary buffer space in a static manner to avoid the overhead of dynamic memory management.

EEMBC allows two modes of measurement. For the "out-of-the-box" mode, the evaluation reflects the results achieved with straightforward compilation of the original benchmark code. This mode allows no optimizations other than what the compiler can achieve using various flags. The "optimized" mode allows modification of the benchmark code, use of intrinsic language extensions and optimized libraries, data re-ordering or restructuring, even hand-tuning in assembly. Practically, the only modification not allowed in the optimized mode is changing the underlying algorithm of the benchmark. To ensure that the optimized or modified versions of each benchmark still produce correct results, EEMBC requires that the produced outputs are either bit identical or within a predefined error margin from the provided reference outputs.

3.2 Benchmark Description

For the microarchitectures presented in this thesis, we focus on the consumer and telecommunications categories of the EEMBC suite. The two benchmark groups represent the typical workload of modern media-processors in consumer devices that combine multimedia applications with high-bandwidth, wired or wireless connectivity.

The following subsections summarize the function and the characteristics of the 10 consumer and telecommunications benchmarks.

3.2.1 Consumer Benchmarks

The consumer category includes five fixed-point, multimedia benchmarks:

- **RGB to CMYK Conversion** (`Rgb2cmyk`): The benchmark converts a digital image to the CMYK format used by most printers. All three of the RGB components of each input pixel are necessary to compute the four CMYK components of the output pixel. The benchmark explores the processor's ability to perform basic matrix arithmetic and fixed-point or trigonometric operations.

- **RGB to YIQ Conversion** (`Rgb2yiq`): The benchmark converts a digital image to the YIQ format that complies with NTSC television standards. The output can be used as an overlay on a standard TV picture. The benchmark applies a special 3x3 matrix to each RGB pixel. It explores the processor's ability to perform basic matrix arithmetic and fixed-point or trigonometric operations.

- **High-pass Gray-scale Filter** (`Filter`): The program receives a dark or blurry gray-scale image and sharpens it with a high-pass filter or smoothens it with a low pass filter. The filters apply a 3x3 convolutional kernel that requires the values for the eight neighbors of an input pixel in order to calculate its output value. The benchmark explores the processor's ability to perform matrix arithmetic.

- **JPEG Compress** (`Cjpeg`): It implements the JPEG standard for "lossy" compression of digital images [Wal91b]. It first performs a discrete-cosine-transform (DCT) on 8x8 image

Benchmark	Iterations per second	Code Size (KBytes)	Data Size (KBytes)
`Rgb2cmyk`	95.64	1.8	230.8
`Rgb2yiq`	41.52	1.6	230.8
`Filter`	57.51	2.0	77.1
`Cjpeg`	10.48	58.6	777.9
`Djpeg`	13.34	58.7	1,042.9

Table 3.2: The characteristics of the EEMBC consumer benchmarks on the NEC VR5000 processor [Tur98]. The VR5000 is a dual-issue MIPS processor running at 250MHz. Its 32-KByte first-level instruction and data caches are two-way set associative. In early 2002, the VR5000 is representative of embedded processors used in high-end consumer products.

blocks. The DCT coefficients are subsequently quantized and compressed using a variable-length Huffman code.

- **JPEG Decompress (`Djpeg`)**: It decompresses a JPEG image to retrieve the original RGB format. It recovers the quantized DCT coefficients from the compressed data and performs the inverse transformations on the 8x8 blocks. Just like the compression part, this benchmark explores the processor's ability to perform pre-loading, handle frequent branches, and execute arithmetic operations on short vectors.

Table 3.2 presents the execution throughput, code and data sizes of the consumer benchmarks for a 250MHz, dual-issue MIPS processor for consumer and telecommunications applications. Most benchmarks have small code size that fits completely in an 8-KByte cache. Even though `Cjpeg` and `Djpeg` have larger code sizes, they result to negligible miss rates with an 8 KByte instruction cache due to the high degree of locality in their dynamic instruction streams [FWL99]. The data sizes for the benchmarks are considerably larger as they are proportional to the size of the input images. Even though there is little temporal locality in the data accesses, there is significant spatial locality that allows for efficient prefetching.

3.2.2 Telecommunications Benchmarks

The telecommunications category includes five fixed-point benchmarks:

- **Autocorrelation (`Autocor`)**: This telephony kernel compresses 8-kilosample/second voice data into a much smaller data stream. It uses autocorrelation to determine the short-term redundancy of the speech signal, due to the filtering by the vocal tract. The autocorrelation coefficients are processed with a code-excited linear prediction (CELP) algorithm to find the filter that closely matches the vocal tract's transfer function. The benchmark explores the processor's ability to handle dot-products.

- **Convolutional encoder (`Convenc`)**: The benchmark implements a key function for a V.xx modem. It creates an output data stream with error detection and correction capabilities using a linear shift register and table look-ups. The program explores the processor's ability to perform bit-wise exclusive or operations and table lookups.

- **Bit allocation (`Bital`)**: This benchmark simulates a key function for asynchronous digital subscriber line (ADSL). The processor must distribute data into a series of "frequency bins." The ADSL modem then modulates and transmits these bins over the telephone line. The benchmark explores the processor's ability to progressively spread a stream of data over a series of buffers using on a "water-level" algorithm.

Benchmark	Iterations per second	Code Size (KBytes)	Data Size (KBytes)
`Autocor`	206,378	1.1	0.05
`Convenc`	2,555	1.6	0.05
`Bital`	11,222	1.5	1.66
`Fft`	3,656	5.4	3.37
`Viterbi`	1,038	3.8	3.80

Table 3.3: The characteristics of the EEMBC telecommunications benchmarks on the NEC VR5000 processor [Tur98].

- **Fast Fourier transform (`Fft`)**: This program performs the fixed-point, fast Fourier transform on 256 complex points for an ADSL application. It converts time domain data into frequency domain information. The benchmark explores the processor's ability to perform complex mathematical and memory access functions.

- **Viterbi decoder (`Viterbi`)**: The benchmark implements the Viterbi decoder used in modem and wireless applications [Vit67]. It receives an input packet encoded with an IS-136 1/2 rate convolutional encoder. It uses a series of add-compare-select (ACS) steps and a back-tracking stage to recover the original information in the presence of transmission errors. The benchmark explores the processor's ability to perform bit-wise operations, comparisons, and table lookups.

Table 3.2 presents the execution throughput, code and data sizes of the telecommunications benchmarks for a 250MHz, dual-issue MIPS processor for consumer and telecommunications applications. Both code and data sizes are small. Data references have limited temporal locality but plenty of spatial locality. The major performance limitation for `Viterbi`, `Bital`, and the `Convenc` is the existence of dependencies in their output data streams, which can prohibit parallel execution. For `Fft`, the bottleneck is typically the complex memory access pattern for the butterfly permutations.

3.3 Discussion

As we already noted in the introduction of this chapter, we use benchmarks to evaluate microprocessors because of the difficulties associated with using full-size applications during the development stages of a design or with porting full applications to a large number of competitive processors. Therefore, any benchmark suite is bound to have some advantages and some basic shortcomings. It is important for processor designers to keep both in mind in order to correctly translate the benchmarking results and understand their limitations.

The EEMBC suite is the first real attempt to develop a non-trivial benchmarking methodology for embedded processors. Its major advantage is the coverage of a large number of embedded application domains. It allows designers to focus their attention to the domain(s) in which they intend to employ their processor without worrying about their benchmark scores in other application areas. EEMBC reports results both with a composite metric for each category and individual scores for each benchmark. The former enables quick comparisons and the latter allows for detailed studies. The benchmarks in the EEMBC suite are not synthetic. They are directly derived from real embedded applications and represent their computationally or memory intensive components. Hence, there is increased confidence that the benchmark results are meaningful. Finally, the EEMBC benchmarks are small and easy to port, tune, and run across a wide range of embedded platforms, regardless of the operating system or IO environment.

On the other hand, the EEMBC benchmarks are not complete applications. Hence, they do not capture interference between kernels in the same application or the overhead of the remaining part of the code. In addition, they cannot evaluate the contribution to performance of system-level components such as DMA engines, streaming buffers, integrated memory, external memory interface, interrupt handling logic, and so on. Embedded processors typically include on same die a variety of memory and IO interfaces and their actual performance can vary significantly depending on the way the applications use the additional hardware. Furthermore, the energy and power consumption are not currently included in the EEMBC reports. Even though energy and power consumption are not as easy to measure or estimate as execution time, they are important metrics for embedded processors. For several embedded applications, it is the performance to power consumption ratio that determines the most appropriate processor.

In terms of the results and comparisons presented in this thesis, one should note the following deficiency of the EEMBC suite. The EEMBC procedure for measuring performance specifies that benchmarks should execute for a large number of times (typically a few hundred) in order to obtain reliable timing information. However, repeated execution along with the tiny data sets for several benchmarks allow caches to capture the data sets and the performance results to overlook the cost of accessing main memory. This situation of warmed-up data caches would not happen with real embedded applications, where a processor would never decode the same network data twice and would rarely attempt to compress the same image more than once. The EEMBC measurement procedure gives processors with cache hierarchies an unfair advantage over processors that implement data prefetching or streaming buffers, the mechanisms most appropriate for handling streaming multimedia data.

The microarchitectures we present in this thesis do not use caches. They employ architecture and microarchitecture techniques that implement prefetching. In contrast, the various commercial processors we compare against use cache hierarchies and benefit from the measurement deficiency.

3.4 Related Work

The EEMBC suite and process for certification of results follow the example of the SPEC [Hen00] and TPC [PF00] organizations for benchmarking desktop and server systems respectively. SPEC and TPC have been continuously updating their benchmarks and procedures for over a decade and have won wide respect within their domains. The EEMBC consortium has the potential to play a similar role for embedded processors but is still in its very early stages.

Before the release of the EEMBC suite, the standard benchmark for embedded processor was Dhrystone, a short synthetic program representative of system programming with integer arithmetic from the early 1980s [Wei84]. Despite its little relevance to the workload of modern embedded systems, most vendors still quote Dhrystone MIPS as a performance metric, the ratio of their processor's execution time over that of the VAX 11/785 [AF88]. Other benchmarks used for embedded processors are Whetstone [GW76] and CPU2. Whetstone is a FORTRAN program that runs a set of loops with integer, boolean, and floating-point arithmetic operations. It performs many iterative calls to programmed subroutines and in-line transcendental functions. CPU2 is also a FORTRAN program that includes portions of frequently used single and double-precision floating-point kernels.

Digital signal processors (DSPs) have typically used separate benchmarks from other embedded designs. The BTDI suite is a popular collection of DSP benchmarks that includes FFT, FIR, IIR, and other related kernels [EB98]. The DSPstone is a similar suite with academic origin [ZMM94]. Recently, DSP vendors have shown interest in the EEMBC suite because it allows comparisons between DSP chips and embedded processors with DSP capabilities.

In the last few years, some academic groups have attempted to define benchmarks for evaluating multimedia processors. The UCLA Mediabench includes a number of image, video,

and voice handling applications available in the public domain [LPMS97]. The Berkeley Multimedia Workload extends the Mediabench suite both in terms of number of applications and size of the input data [SS01a]. None of these suites has become popular outside the academic environment. First, they represent the multimedia applications of workstations and desktop PCs and not the applications in embedded devices. In addition, their code relies on the services of a full-size operating system, which is not always available in embedded systems. The groups that proposed these suites did not attempt to define and provide support for an acceptable process for measurement and certification of results.

Finally, the MiBench is a recent academic effort for a free version of the EEMBC suite [GRE+01]. MiBench introduces a new benchmark category for security applications such as encryption and digital signatures. In the five original categories, MiBench includes representative kernels but not necessarily the same with EEMBC. In addition, MiBench focuses on high-end embedded processors (32-bit or 64-bit) and not on 8-bit or 16-bit microcontrollers. Consequently, the MiBench input data are at least one order of magnitude larger than the input data in EEMBC. Hence, MiBench results are not directly comparable to EEMBC results. In early 2002, MiBench results are not available for any commercial or research processor.

3.5 Summary

In this chapter, we presented the EEMBC suite for comparing embedded processors in a variety of application domains. We will use the consumer and telecommunications benchmarks of EEMBC to evaluate the vector microarchitectures proposed in this thesis. The five consumer benchmarks represent image processing applications in digital cameras. The five telecommunications programs represent applications for ADSL and wireless communication.

The EEMBC benchmarks capture the basic kernels for the corresponding applications domains and exercise the execution component of an embedded microprocessor. However, their small and static data sets are not representative of streaming input data in multimedia applications and fail to stress the streaming capabilities of the memory system. Nevertheless, the EEMBC suite is currently the only realistic option for evaluating a new microprocessor with a variety of embedded applications and comparing it to number of commercially available designs.

Chapter 4

Vector Instruction Set Architecture for Multimedia

"When you do the common things in life in an uncommon way,
you will command the attention of the world."
George Washington Carver

The instruction set (ISA) is the portion of an architecture that is visible to software. It defines the register and memory state of the architecture and a set of instructions that can operate on the state. Certain instruction sets allow software to express a single operation on the state with one instruction. Vector instruction sets, on the other hand, allow software to express multiple independent operations with one instruction, which makes it easier to implement efficient hardware. This chapter focuses on adjusting vector architectures to multimedia processing and introduces the VIRAM instruction set, a vector architecture developed for embedded media processors.

Section 4.1 provides a brief overview of traditional vector instruction sets used with supercomputing applications. In Sections 4.2 and 4.3, we introduce a set of architectural enhancements that target the characteristics of multimedia programs and general purpose systems. Section 4.4 presents the VIRAM architecture with a summary of its state and instructions. In Section 4.5, we analyze the use of the VIRAM instruction set with multimedia benchmarks and compare it with alternative architectural approaches such as VLIW and SIMD. Section 4.6 utilizes the experience from using the VIRAM ISA with the multimedia benchmarks in order to evaluate some of the basic decisions we made during its development. Finally, Section 4.7 overviews related work in instruction set architectures for multimedia processing.

4.1 Introduction to Vector Instruction Set Architectures

Parallelism is the key to achieving high performance in modern microprocessors, for it allows hardware to accelerate the execution of an application by processing multiple of its operations concurrently. Vector architectures provide a set of instructions for explicitly representing the data-level parallelism in an application to the hardware used to execute it. They have been used commercially for nearly three decades in the areas of scientific and high performance computing [Rus78, MU84, Jon89, HL96]. This section reviews the fundamental concepts in register-based, vector architectures and provides the background for the instruction set issues addressed in the rest of this thesis. Hennessy and Patterson provide a longer introduction to vector processing in Appendix G of [HP02].

Vector instruction sets include the basic arithmetic and memory instructions for operating

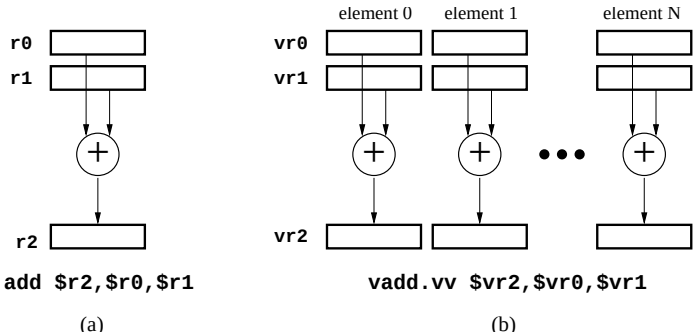


Figure 4.1: The difference between scalar and vector instructions. A scalar instruction (a) defines a single operation on a pair of scalar operands. An addition instruction reads two individual numbers and produces a single sum. On the other hand, a vector instruction (b) defines a set of identical operations on the elements of two linear arrays of numbers. A vector addition instruction reads two vector operands, performs element-wise addition, and produces a similar array of sums.

on individual numbers, similar to those found in all RISC architectures. In addition, vector architectures define high-level instructions that operate on linear arrays of numbers, known as vectors. Figure 4.1 shows a vector instruction that specifies two vectors as input operands and produces a result vector by executing the same operation on each pair of elements from the input arrays. In other words, a single opcode and set of operands define a large number of identical, yet independent, operations on the elements of two arrays. A vector instruction set typically defines vector instructions for all logical, integer, and floating-point operations. Vector architectures store array operands in a vector register file, in the same way a register array holds the operands for scalar instructions in RISC architectures. A vector register file is a two-dimensional storage array, where each row holds all the elements for a single vector. The number of elements per register is a property either of the ISA itself or of each processor that implements it.

A compiler can set the degree of data-level parallelism expressed by vector instructions using the vector length register (`VL`). Each instruction reads this control register as a default input operand and uses it to determine the number of element operations to execute. For example, an addition instruction with vector length set to five will only add the first five element pairs of the input operands and will not affect the remaining elements of the output vector register. Depending on the amount of parallelism available in the application, the compiler can set the vector length register to any value between one and the maximum number of elements stored per register, known as the maximum vector length (`MVL`). If the data-level parallelism in the application exceeds that of one vector instruction at maximum vector length, software can use strip-mining, a technique that expresses a long sequence of identical operations as a loop of vector instructions [GBS94]. If all vector instructions in a sequence are supposed to execute the same number of element operations, the compiler needs to set the vector length register only once at the beginning of the sequence.

In addition to instructions for integer and floating-point operations, vector architectures define instructions for exchanging data between vector registers and the memory system. Similar to arithmetic instructions, vector memory operations perform a large number of element accesses and operate under vector length control. Table 4.1 describes the three addressing modes supported by virtually all vector architectures: unit stride mode fetches data from sequential locations; strided mode addresses data in memory locations separated by a constant distance (stride); and indexed mode uses the corresponding elements of a vector register as pointers to memory for exchanging data for each element operation. The three modes allow applications to express their data-level parallelism with vector operations on consecutive elements in the vector register file, regardless of the exact layout of data in memory. As with scalar loads and stores, vector memory instructions

Mode	Example instructions	Meaning	When used
Unit stride	`vld $vr0,addr`	`for i=0 to VL` `  vr0[i]←Mem[addr+i*d]`	Sequential accesses; stepping through arrays within a loop
Strided	`vlds $vr0,addr,str`	`for i=0 to VL` `  vr0[i]←Mem[addr+i*d*str]`	Accessing columns of arrays; accessing color components of pixels
Indexed	`vldx $vr0,$vr1,addr`	`for i=0 to VL` `  vr0[i]←Mem[addr+vr1[i]]`	Accessing sparse matrices; pointer chasing

Table 4.1: The three addressing modes for vector memory accesses. The variable `d` designates the size of the data item in memory (1, 2, 4, or 8 bytes). A vector instruction set typically defines one load and one store instruction per addressing mode and data size combination. When the size of elements in registers is larger than the size of data in memory, the instruction implies a conversion using sign extension or truncation of the most significant bits.

come in several variations to support all the possible sizes of data in memory (1, 2, 4, and 8 bytes). The typical alignment restriction is that the memory address for each vector element must be a multiple of the memory data size in bytes. There are no alignment restrictions for the whole vector.

There are a few vector instructions that deviate from the element-wise execution model. The most common exceptions are instructions that allow shuffling of elements within a vector register. Although there is no uniform support across all architectures, commonly supported primitives include operations for shifting element locations within a vector register (insert and extract) or packing and unpacking of elements using a bit-mask (compress and expand). There are also instructions that allow manipulation of a few control registers such as the vector length and support the exchange of data between that vector register file and the scalar register file that holds operands for scalar instructions.

It should be clear from the brief introduction that vector architectures have several important advantages for applications with a high degree of data-level parallelism:

- A single vector instruction defines a large number of independent operations. By executing multiple element operations concurrently, a vector processor can keep multiple, deeply pipelined datapaths busy without the need for high instruction fetch and decode bandwidth. Since element operations are explicitly independent, there is no need to implement hardware for hazard checks within one vector instruction.

- Vector memory instructions have a known access pattern. Prefetching techniques can accelerate sequential or strided access for a large number of elements. The latency for initializing a memory access can be amortized by fetching multiple elements in parallel or in a pipelined manner. In other words, if sufficient memory bandwidth is available, memory latency is exposed once for the entire vector, instead of once for each element in the vector.

- The roles for the processor (hardware) and the compiler (software) are truly complementary and allow each one to focus on what it can do best. The compiler discovers the data-level parallelism available in the application code and expresses it with vector instructions. The hardware uses the explicit knowledge of parallelism to execute multiple element operations concurrently for vector arithmetic and memory instructions.

- A single vector instruction is equivalent to an entire loop. Hence, applications require less overhead instructions for incrementing loop indices or branching. The result is compact code size and reduced dynamic instruction count.

4.2 Vector Architecture Enhancements for Multimedia

The data-level parallelism available in multimedia is similar to that in scientific applications, as they both include computationally intensive kernels that repeat the same set of operations on their input data. Still, in order to express multimedia applications efficiently using vector instructions, a set of enhancements to traditional vector architectures is necessary. The modifications target some of the distinctive characteristics of multimedia programs, such as the use of narrow data types, fixed-point arithmetic, and reduction operations.

4.2.1 Support for Narrow Data Types

Unlike scientific applications where double-precision (64-bit) floating-point numbers are the predominant data type, multimedia programs process video or audio streams using narrower data. Pixels and audio samples are typically stored in memory using sequences of 8-bit or 16-bit data. In addition, human vision and hearing have such a limited range of inputs that 16-bit and 32-bit accuracy is generally sufficient during data processing. Therefore, a vector instruction set for multimedia must also define vector operations on 16-bit and 32-bit numbers in a way that allows for efficient implementation.

A vector architecture can support all three data types within a single vector register file by allowing different registers to store elements of different size. In order to use hardware resources efficiently regardless of the data width, the storage space for a 64-bit element can hold multiple narrower elements in every vector register. Similarly, segmented 64-bit datapaths for arithmetic operations can execute multiple narrower operations in parallel. Therefore, four 16-bit vector elements fit in the space for a single 64-bit one, and a segmented 64-bit adder can calculate four 16-bit element additions in parallel.

The compiler selects the width of vector elements and operations using the virtual processor width register (`VPW`) [HT72]. Just like vector length, this control register is a default input operand to all vector instructions. It selects between 64-bit, 32-bit, or 16-bit data types and instructs the hardware to translate vector elements and operations accordingly. By properly setting `VPW` before accessing each register, the compiler can store vectors of different element sizes in the same register file. When the virtual processor width selects a narrow data type, the maximum vector length (`MVL`) increases. A processor with maximum vector length of 16 elements for 64-bit data can store up to 32 elements per vector register for 32-bit data, or 64 elements for 16-bit data. As discussed in Chapter 5, switching to a narrower `VPW` also increases the peak performance of a vector processor.

An alternative mean to the `VPW` register for selecting the width of vector elements and operations would be to encode it within the opcode of every vector instruction. All recent SIMD extensions for RISC and CISC architectures employ this method for treating 64-bit scalar registers as short vectors of narrower data. This approach consumes a large amount of opcode space, as every operation requires one opcode for each supported data type. Given the large number of operations in modern instruction sets and the variances necessary to select important options (signed or unsigned arithmetic, vector or scalar operands, and so on), it is difficult to use this method without limiting the number of operations supported or significantly complicating the instruction encoding. The `VPW` approach, on the other hand, requires only one new control register and no additional opcode space. The run time overhead for setting the control register is also small. Because multimedia kernels typically use a single data type width for all arithmetic operations within each loop, a compiler can set the `VPW` register once per loop to the widest data type used by any loop instruction.

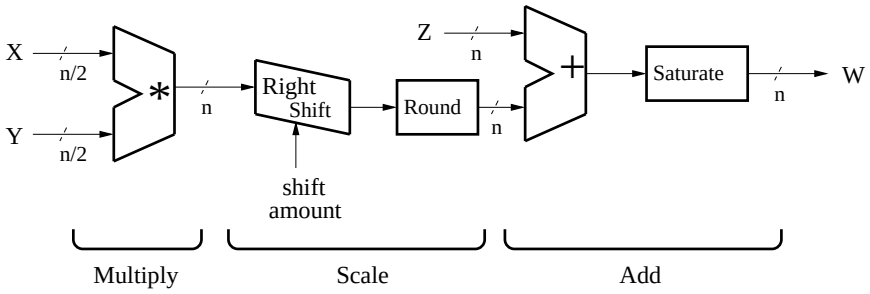


Figure 4.2: The multiply-add model for a vector architecture for multimedia. All operands (X, Y, Z, W) are vector register elements. The variable `n` designates their width (`VPW`) and the selected accuracy of the overall result. The multiplier uses only half of the bits from X and Y, either the upper or the lower part, in order to produce an n-bit result. Shifting right by a programmable amount can scale the multiplication result to any fixed-point format. Rounding after scaling and saturating after adding improve the final accuracy. If Z and W are elements of the same register, the operation becomes multiply-accumulate.

4.2.2 Support for Fixed-point Arithmetic

Apart from narrow data types, multimedia applications frequently use saturated and fixed-point arithmetic. Saturation replaces modulo arithmetic to reduce the error introduced by overflow and underflow in signal and image processing algorithms. Fixed-point operations allow decimal calculations within narrow integer formats. They require less hardware resources and are faster to execute than floating-point operations.

Digital signal processor (DSP) architectures have always provided support for saturation and fixed-point arithmetic. Their instruction sets include special versions of add, subtract, and shift instructions that saturate the result in the case of overflow. Fixed-point support is more complicated as the result of a multiply-accumulate operation, the most common fixed-point primitive, has twice as many bits as the input operands. DSP architectures use one of the following two methods in order to store as many result bits as possible and achieve high accuracy for the overall computation [LBSL97]. The first approach introduces one or more special registers, called accumulators, that can store more bits than the regular architecture registers. For example, a 16-bit DSP architecture may include a 40-bit accumulator. Successive multiply-accumulate operations store their wide result in an accumulator. The alternative approach suggests the use of a pair of consecutive registers in a register file to store the wide result. Both approaches are efficient in terms of hardware resources, but complicate significantly the task for compiling high-level code for such architectures. Register allocation and instruction scheduling are particularly difficult in the presence of a few extended-precision registers or register pairs, that can be manipulated, loaded, or stored using special instructions and complicated ordering rules. It is not surprising that most DSP chips perform significantly better using hand-optimized assembly code rather than compiled executables [EB98, Lev00].

Figure 4.2 presents the multiply-add model that allows a vector architecture to provide flexible support for fixed-point numbers of arbitrary format using regular vector registers for all input and output operands. There are three basic steps to this model: first multiply the upper or lower halves of the first two inputs; then scale the multiplication result to the desired fixed-point format by shifting and rounding; finally perform saturating add with the third input operand. The architecture can provide a separate instruction for each step or a single instruction for the whole operation. In latter case, if there are not sufficient bits in the instruction encoding for the four register operands, the third input and the output can be the same vector register, turning the operation into a multiply-accumulate. Control registers can provide the shift amount for scaling and

the rounding mode, since the fixed-point format rarely changes halfway through a computation.

The arithmetic accuracy of the multiply-add model is a function of the data type width (`VPW`) selected during its execution. With a large `VPW`, vector elements can hold more decimal bits leading to a small rounding error. On the other hand, with a small `VPW`, vector registers hold more elements and wide datapaths execute more narrow operations in parallel. Therefore, there is a trade-off between accuracy and performance, and the compiler can select the appropriate setting for each application. For example, consider an application that multiplies and accumulates 256 vector pairs of 8-bit data in memory. Executing the multiply-add operations with `VPW` set to 32-bit leads to full accuracy with no decimal bits rounded off, regardless of the exact data values. On the other hand, if the application does not require full accuracy or the data values are relatively small, the application can execute twice as fast by setting `VPW` to 16-bit. Since most multimedia applications can tolerate small deviations in arithmetic precision, with careful selection of the shift amount and the rounding mode, narrow data types are often appropriate to use with the proposed multiply-add model.

Regardless of the type of registers used for multiply-add operations, programming language issues prohibit compilers from extensively using fixed-point instructions. Languages like C and C++ do not provide straight-forward methods for expressing fixed-point arithmetic or saturation in the application code. Consequently, calls to special library functions and assembly programming are currently the only ways to exploit fixed-point support in the instruction set. Once the recently drafted extensions that add fixed-point data types to the C language become widely used [Org99], compilers will likely overcome this limitation. An alternative approach is to use compiler extensions that allow users to describe application-level characteristics to the compiler [ECCH00]. The programmer can instruct the compiler how fixed-point arithmetic is described using existing language semantics, which allows the compiler to use fixed-point instructions whenever possible.

4.2.3 Support for Element Permutations

Reduction or butterfly primitives are frequent in multimedia applications that include dot-products or transformations like FFT. These primitives are difficult to express with vector instructions because either they produce a scalar output instead of a whole vector, in the case of reductions, or they don't operate on corresponding elements from the input vectors, in the case of FFT. To address this inadequacy, we introduce three vector instructions that perform a restricted set of permutations on the elements of a vector register.

Figure 4.3 presents the `vhalf` permutation instruction for implementing reduction primitives. The instruction splits the elements of a vector into two registers. Adding the two registers at half the vector length performs the first step of an add reduction. By iteratively applying the permutation, the vector length reduction, and vector addition of the two registers, we can reduce the original vector to a single element sum. Replacing the addition with other instructions, such as multiply or exclusive or, allows the implementation of a variety of arithmetic and logical reductions.

Figure 4.4 presents the operation of the `vhalfup` and `vhalfdn` permutation instructions for vectorized butterflies. Each instruction copies elements between two registers towards one direction using a programmable distance. Along with the corresponding vector addition, the two instructions implement one stage of a floating-point or fixed-point FFT. Even though a single instruction could implement both permutation patterns [Asa98], individual instructions are preferred because they require less hardware resources and simpler control for interlocks.

A single instruction that can execute any random permutation of vector elements could replace the three proposed instructions. Certain SIMD extensions for RISC architectures include a general permutation instruction for this purpose [Phi98]. Yet, due to the generality of a random permutation, such an instruction has complicated control and is either slow or requires an expensive crossbar structure for fast execution. On the other hand, the three simpler permutation instructions implement only the small set of permutation patterns that are frequent in multimedia applications.

Figure 4.3: An add reduction of a vector with eight elements using the `vhalf` permutation instruction. The instruction reads the second half of the source register and copies the elements at the beginning of the destination register. The vector length register, which must be a power of two, indicates the number of elements in the register at any time. A reduction step consists of `vhalf` followed by a vector addition of the source and destination registers of the permutation at half the vector length. Three iterative steps are necessary to reduce the initial vector to a single element value.

Due to their regularity, they are easy to implement and execute fast with modest hardware resources (see Chapter 5).

Another alternative is to implement the permutations with the shuffling instructions in traditional vector architectures (extract, insert, compress, and expand) or with strided and indexed memory operations. The shuffling instructions are more complicated to implement than the three simple permutations and do not directly provide the desired functionality. The use of memory instructions puts unnecessary pressure on the memory system, the most significant performance bottleneck for most modern processors. The modest hardware resources needed for the permutation instructions should be easier to provide than increased memory system performance.

To utilize the permutation instruction for reductions, a compiler must recognize linear recurrences on operations like addition, maximum and minimum, and logical exclusive or. On the other hand, recognizing the many algorithmic descriptions for FFT is a more difficult task. Hand-optimized libraries with FFT routines are an easier way to use the butterfly permutation instructions.

4.2.4 Support for Conditional Execution

Several multimedia kernels include conditional statements, such as if-then-else constructs, in the main loop body. Without special architectural support, the branch instruction necessary to implement a conditional statement prohibits the use of vector instructions.

A vector architecture can support vectorized execution of conditional statements using a flag register file, which stores vector registers with single bit elements [Asa98]. Each vector instruction uses one of the flag register as a source of masks for conditional execution of its element

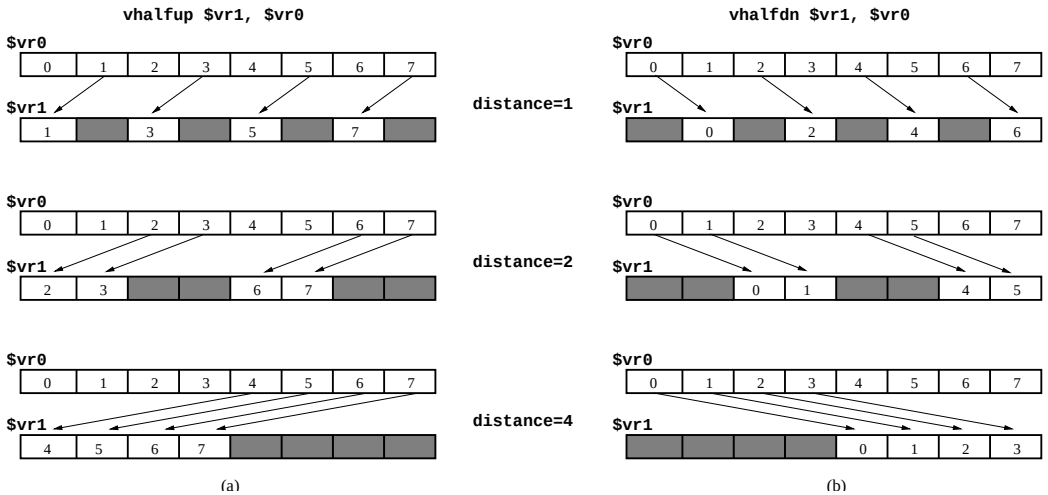


Figure 4.4: Example use of the `vhalfup` (a) and `vhalfdn` (b) permutation instructions. Each instruction performs one half of a butterfly permutation by copying elements from the input to the output register based on the distance stored in a control register. They only differ in the direction of the element move. The instructions do not modify the remaining elements of the output register, which are marked with gray. The vector length must be a power of two for both instructions.

operations. An operation writes its result back to the register file only if the corresponding bit in the flag register is set. Comparisons between elements of vector registers, memory loads, and logical operations can produce the proper values for flag registers. If there are not enough bits in the encoding of every instruction to specify any flag register, instructions can select from one of two default flag registers using a single opcode bit. Logical operations between flag registers can move values to and from the default flags, if more than two flag registers are necessary as with the case of nested conditional statements.

Masked execution of instructions using flag registers is not the only way to support conditional execution for vector registers. In [SFS00], Smith, Faanes, and Sugumar present a thorough study of all the alternatives for conditional execution, including methods based on conditional merges, indexed memory operations, and vector shuffling instructions (compress - expand). The last two approaches compress vectors so that no masked elements are included in a register. The study reviews the ease of use with a variety of programming constructs, as well as the complexity and speed of potential hardware implementations for each alternative. It concludes that masked execution of vector instructions is simpler to use and implement, especially in highly parallel vector processors. Even though masked execution is not always the fastest alternative for certain sparse matrix kernels, it is sufficiently fast for all programming constructs, including nested if statements.

4.3 Vector Architecture Enhancements for General Purpose Systems

With vector architectures for scientific computing, system issues are not a major consideration. In this environment, a single application may run uninterrupted for hours and an auxiliary computer handles input-output and other system activities. For vector processors employed in embedded systems, however, this is not necessarily the case. Even though the scalar component of a vector architecture implements most system features, a few potential issues, such as virtual memory

and context switch time, call for the introduction of system concepts in the heart of the vector instruction set.

4.3.1 Support for Virtual Memory

Virtual memory is likely the single most important system feature for architectures for desktop computers and servers. It enables address space protection, data and code sharing, large address spaces with paging to disk, sparse address spaces, and memory-mapped IO. For embedded systems, several of these features are not particularly important, because such systems tend to be simpler and used for a small set of tasks. However, as embedded software development becomes an increasingly difficult task, some embedded systems require an operating system as complicated as Linux. For this case, address space protection is necessary and the architecture must support virtual memory.

There are several alternatives and choices in providing architectural support for virtual memory: pages or segments, software or hardware assisted exceptions, and so forth [JM98]. For this work, we have selected to support paged virtual memory using a software managed translation lookaside buffer (TLB) as defined in the MIPS architecture [Hei98]. Still, all issues would be similar with any other choice. If virtual memory is enabled in a specific system, the TLB must perform protection checks and translate from virtual to physical all addresses generated for vector load and store instructions. There are two issues to address regarding TLB accesses. First, with vector processors attempting to generate multiple addresses per cycle in order to fetch many elements in parallel, the TLB can become an additional performance bottleneck. In Chapter 5, we discuss a vector TLB implementation that addresses this problem. The second issue is the state of the processor when an address lookup misses in the TLB or generates a protection violation. Since this is a software managed TLB, in either case, a virtual memory exception is generated and execution continues with an operating system handler.

Ideally, the state of a processor after any exception is precise. All the instructions before the faulting one have completed their execution and all other instructions, including the faulting one, have not updated the architecture state. In practice, this is hard to implement because modern processors execute multiple instructions in parallel in a pipelined and, some times, out of order manner. It is even more complicated for vector architectures because every instruction defines a large number of operations that may execute over a long period of time and any one of them can cause a exception. In Chapter 5, we present a simple vector processor with support for imprecise, yet restartable, memory exceptions. Several instructions before or after the faulting one may be partially completed at the time the software handler initiates, but the processor provides sufficient mechanisms for the handler to run and execution of the interrupted program to resume correctly. In Chapter 7, we revisit the definition of precise state for a vector processor and discuss a mechanism for implementing precise exceptions and its cost in terms of performance and hardware resources.

4.3.2 Support for Arithmetic Exceptions

Arithmetic instructions can also produce exceptions, especially those operating on floating-point numbers. For vector supercomputers, architectural support for detecting and servicing floating-point exceptions at full execution speed is critical, because the accuracy of applications like weather prediction often depends on the ability to correctly handle denormals, overflow, and underflow. For multimedia applications, on the other hand, elaborate support for fast arithmetic exceptions is not an important requirement. In most cases, arithmetic exceptions are disabled altogether, since a few incorrect pixels in a video frame are preferred over the impact on real-time performance induced by running exception handlers whenever exceptions occur. Support for arithmetic exceptions is necessary only for application debugging, during which reduced execution speed is not an issue.

Asanovic has proposed a method for keeping track of arithmetic exceptions using the flag register file [Asa98]. Element operations that generate exceptions set the corresponding bits in flag registers, with a separate register used for each type of integer or floating-point exception. The application can periodically check these flag registers and raise the proper exceptions if any bits are set. Coupled with a processor mode for executing a single vector instruction at the time, this method provides sufficient support for application debugging.

4.3.3 Support for Context Switching

With each vector register containing tens of elements, the amount of architectural state in a vector processor is considerable. Saving and restoring all the vector state can increase the cost of context switching, despite the presence of a high bandwidth memory system for vector loads and stores. To alleviate this impact, it is important to reduce both the number of context switches that manipulate vector state and the amount of state saved or restored each time.

In a well-balanced system for real-time applications, input-output handlers and periodic operating system routines are the most common reasons for interrupting the execution of a program. These handlers use scalar instructions and rarely update any vector state. To avoid saving and restoring vector registers is such cases, we need a mechanism for marking vector state and instructions as "unusable" before switching out a vector application. If the new application, whether a handler or a user program, tries to issue a vector instruction or access vector state, an exception is generated to instruct the operating system to save and restore the vector state. Otherwise, the previous application will find its vector state intact when it resumes execution. A control register that stores the identification number for the last process that updated vector state is necessary for this technique. On issuing a vector instruction, we compare this register to the identification number of a currently running process and generate an exception if they are different. Chapter 5 discusses how this method allows a vector processor to continue executing vector instructions for an application, while an interrupt handler runs using scalar instructions.

To reduce the amount of vector state involved in a context switch, the architecture can define valid and dirty bits for each vector register [PMSB88]. The valid bits indicate registers with useful data that the operating system must restore during context switches. The dirty bits indicate registers with data updated since the last context switch that the operating system must save. For a system with 32 vector registers, a couple of control registers are sufficient to store all dirty and valid bits. Writes on a vector register should automatically set the corresponding valid and dirty bits. The compiler should produce code that clears the valid bit every time it deallocates a register, and the operating system must clear all dirty bits when saving the state for a vector process. The hardware can also keep track of the largest vector length used by an application in order to reduce the number of elements saved and restored for each register [Asa98].

4.4 The VIRAM Instruction Set Extension for MIPS

To explore in practice the ideas presented in Sections 4.2 and 4.3, we developed the Vector IRAM (VIRAM) instruction set extension for the MIPS-64 RISC architecture [MIP01]. Figure 4.5 presents the vector state introduced by the VIRAM architecture, including the vector and flag register files. The instruction set does not define the maximum number of elements per vector or flag register. An implementation can select the proper `MVL` value based on the performance requirements and available hardware resources. A read-only control register makes this value available to software at run-time, hence with proper strip-mining of vectorized code, the same binary executable can run on every processor implementation of the architecture.

Table 4.2 presents a summary of the VIRAM instructions. There are 91 instructions that, due to the various instruction options, occupy 660 opcodes in the coprocessor 2 space of MIPS-64.

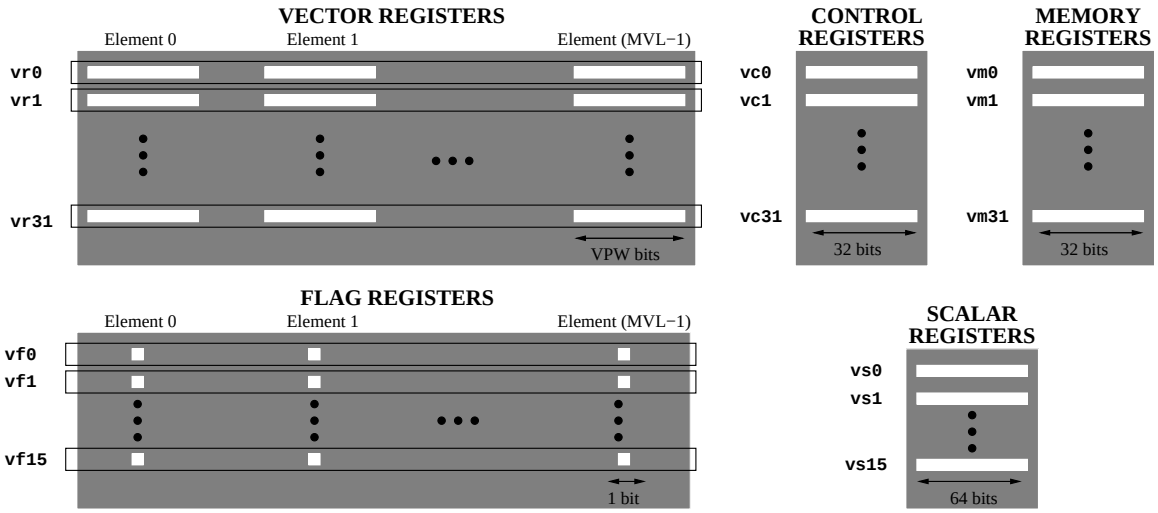


Figure 4.5: The architecture state in the VIRAM instruction set. The vector and flag register files have 32 and 16 registers respectively. The control register file holds values such as the vector length (`VL`) and the virtual processor width (`VPW`). Two additional scalar register files hold scalar data for vector instructions and memory addresses, strides, and increments for vector load and store operations. This state is in addition to any state defined by the MIPS-64 architecture.

The majority of opcodes define arithmetic operations for integer, fixed-point, and floating-point numbers. Vector loads and stores allow an arbitrary post-increment to the scalar register that holds the base address. All arithmetic and memory operations can select one of the first two flag registers to provide masks for conditional execution. For a detailed presentation of all the instructions set features, refer to the VIRAM instruction set manual [Mar99].

Defining VIRAM as a coprocessor extension instead of a stand-alone instruction set allows us to take advantage of any orthogonal developments in the MIPS architecture and use the large amount of development tools available for it. Since coprocessor specifications at the instruction set level are similar for most RISC architectures, we can easily port the vector extensions to any other architecture. We can also use any MIPS processor core with coprocessor support as the basis for a VIRAM processor implementation.

The penalty for this flexibility is the need for the three scalar register files in the vector architecture state (see Figure 4.5). Vector instructions must access scalar registers for control values, memory addresses, and scalar operands. If the scalar and vector hardware are implemented separately and communicate only through the coprocessor interface, vector instructions cannot directly access the register files in the scalar core. Move to and from coprocessor instructions are the only way to move scalar data to the register files in the vector hardware for vector instructions to use. Even though the area for the three register files may be negligible for most implementations, the move instructions can introduce significant run-time overhead if not carefully optimized by the compiler. If close integration of the scalar and vector hardware is an option, however, the vector instructions can read scalar values directly from the MIPS-64 register file, hence, the extra register files and the move instructions are no longer necessary.

The vectorizing compiler used with the VIRAM architecture is based on the Cray PDGS system for vector supercomputers [Cra00]. It has C, C++, and Fortran front-ends and performs an extensive list of optimizations including outer-loop vectorization. We developed a back-end that generates executable code for VIRAM implementations. To simplify the back-end development, we included in the architecture several traditional vector instructions not obviously necessary with

Mnemoic	***Operation***
Vector Integer Arithmetic	
`vabs`	Absolute value
`vadd`	Add
`vsub`	Substract
`vmullo`	Multiply lo
`vmulhi`	Multiply hi
`vdiv`	Divide
`vmod`	Modulo
`vsra`	Arithmetic right shift
`vcmp`	Compare
`vmin`	Minimum
`vmax`	Maximum
Vector Floating-Point Arithmetic	
`vabs.{s,d}`	Absolute value
`vadd.{s,d}`	Add
`vsub.{s,d}`	Subtract
`vmul.{s,d}`	Multiply
`vmadd.{s,d}`	Multiply add
`vmsub.{s,d}`	Multiply subtract
`vnmadd.{s,d}`	Negative multiply add
`vnmsub.{s,d}`	Negative multiply subtrace
`vdiv.{s,d}`	Divide
`vrecip.{s,d}`	Reciprocal
`vsqrt.{s,d}`	Square root
`vrsqrt.{s,d}`	Reciprocal square root
`vneg.{s,d}`	Negate
`vcmp.{s,d}`	Compare
`vcvt`	Convert
`vtrunc`	Truncate
`vround`	Round
`vceil`	Celling
`vfloor`	Floor
Vector Fixed-Point Arithmetic	
`vsat`	Saturate
`vsadd`	Saturating add
`vssub`	Saturating subtract
`vsrr`	Saturating shift right
`vsls`	Saturating sift left
`vxumul`	Multiply upper halves
`vxlmul`	Nultiply lower halves
`vxumadd`	Multiply add upper halves
`vxumsub`	Multiply subtract upper halves
`vxlmadd`	Multply add lower halves
`vxlmsub`	Multply subtract upper halves

Mnemonic	***Operation***
Vector Logical	
`vand`	And
`vor`	Or
`vxor`	Exclusive Or
`vnor`	Nor
`vsll`	Shift left logical
`vsrl`	Shift right logical
Vector Load and Store	
`vld`	Unit stride load
`vst`	Unit stride store
`vlds`	Strided load
`vsts`	Strided store
`vldx`	Indexed load
`vstx`	Indexed store
`vstxo`	Ordered indexed store
`vfld`	Flag load
`vfst`	Flag store
Vector Processing	
`vins`	Insert
`vext`	Extract
`vcompress`	Compress
`vexpand`	Expand
`vmerge`	Merge
`vfins`	Flag insert
`vhalf`	Reduction permutation
`vhalfup`	Left butterfly
`vhalfdn`	Righ butterfly
Flag Logical	
`vfand`	And
`vfor`	Or
`vfxor`	Exclusive Or
`vfnor`	Nor
`vfclr`	Clear
`vfset`	Set
Flag Processing	
`viota`	Iota
`vciota`	Continuous iota
`vfpop`	Population count
`vfff1`	Find first one
`vffl1`	Find last one
`vfsetbf`	Set before first one
`vfsetif`	Set including first one
`vfsetof`	Set only first one

Table 4.2: The VIRAM instruction set summary. Instruction options, such as signed or unsigned arithmetic and the data size in memory, have been omitted for simplicity.

multimedia applications, such as the compress and expand shuffling operations. The compiler is able to recognize linear recurrences on operations like addition and logical exclusive or, and generate vector code for the reduction using the permutation instructions. On the other hand, recognizing the many algorithmic descriptions for FFT is a difficult task for the compiler. Instead, we provide a set of hand-optimized FFT routines as an easier way to use the butterfly permutation instructions available.

The VIRAM instruction set was in development from 1997 to 2000. The original definition was missing certain features, which we added during the design of the VIRAM-1 processor implementation (see Chapter 5). We introduced post-increment for vector loads and stores to reduce the overhead of coprocessor moves and the permutation instructions for vectorizing dot-products and FFTs. The need to support arbitrary fixed-point formats motivated the current form of the vector multiply-add model. In terms of implementation difficulty, most instruction set features, including support for multiple data sizes, proved straight-forward. Vector memory operations were the most difficult to implement, due to their interaction with the memory system and the importance of high performance loads and stores for a vector processor. The regularity of the permutation instructions made their control logic simpler than we initially expected. On the other hand, the traditional vector instructions for shuffling elements were extremely difficult to implement and debug.

4.5 Instruction Level Analysis of Multimedia Benchmarks

Before exploring the efficiency of hardware implementations of the VIRAM architecture, it is important to perform an instruction level characterization of the multimedia benchmarks. Specific information on the use of the instruction set features in each benchmark is the key to understanding potential performance deficiencies and proposing hardware mechanisms to overcome them.

4.5.1 Benchmark Vectorization

All benchmarks in the consumer and telecommunications categories of the EEMBC suite are vectorizable to some extent. With the exception of `Fft`, the VIRAM compiler is able to vectorize each benchmark in a way similar to the one an experienced assembly programmer would use.

`Rgb2cmyk` and `Rgb2yiq` contain a single loop that operates separately on every pixel in the image. They are trivial to vectorize using strided accesses to separate the color components of each pixel. On the other hand, `Filter` contains two nested loops that iterate across the rows and columns of the image respectively. We vectorized the loop for columns because it allows the use of unit stride accesses and requires fewer loads per pixel. Because digital images contain thousands of pixels, `Rgb2cmyk`, `Rgb2yiq`, and `Filter` perform arithmetic and memory operations on long vectors with tens of elements.

Unlike the other EEMBC benchmarks that are single-function kernels, `Cjpeg` and `Djpeg` are larger applications with tens of subroutines. They both perform 2-D discrete cosine transforms (DCT) on 8×8 pixel blocks, which account for more than half of the overall run-time. We vectorized the two constituent, 1-D DCTs across the rows and columns of the block respectively. The vector length in the DCT code is 8. Alternatively, we could perform outer-loop vectorization and process tens of 8×8 blocks concurrently in order to generate longer vectors. However, outer-loop vectorization of DCT would require global changes to the function structure and the buffering scheme in the two benchmarks. Apart from DCT, we vectorized several image manipulation routines (color conversion, up-sampling, down-sampling) and functions that emulate file-system operations. These functions operate on whole rows or columns of images and have long vectors. For `Cjpeg`, we also vectorized the discovery of runs of zero coefficients in the function for Huffman encoding.

For the `Autocor` telecommunications benchmark, we vectorized the inner loop that performs a dot-product on the input data for each time-delay unit defined by the outer-loop. With

	Op Count ($\times 10^3$)	Inst Count ($\times 10^3$)	% Vector Op	% Vector Inst	% Scalar Op	% Scalar Inst	VPW	Vector Length Avg (Max)
Rgb2cmyk	1,000.2	11.4	99.6%	68.4%	0.4%	31.4%	16 (100%)	128.0 (128)
Rgb2yiq	1.386.0	36.0	98.9%	59.9%	1.1%	40.1%	32 (100%)	64.0 (64)
Filter	1,147.7	20.1	99.2%	53.2%	0.8%	46.8%	16 (100%)	106.0 (128)
Cjpeg	13,682.3	5,427.4	64.8%	11.5%	35.2%	88.5%	16 (27%) 32 (73%)	41.4 (128) 13.3 (64)
Djpeg	11,997.3	4,505.3	67.2%	12.7%	32.8%	87.3%	16 (39%) 32 (61%)	98.1 (128) 13.0 (64)
Autocor	54.5	4.1	94.7%	29.2%	5.3%	79.8%	32 (100%)	43.4 (64)
Convenc	8.4	0.3	97.1%	20.3%	2.9%	79.7%	16 (100%)	128.0 (128)
Bital	192.8	13.1	95.7%	36.9%	4.3%	63.1%	32 (100%)	38.4 (64)
Fft	22.5	0.7	98.9%	63.2%	1.1%	36.8%	32 (100%)	63.7 (64)
Viterbi	147.6	19.6	92.1%	40.1%	7.9%	59.9%	16 (100%)	17.8 (128)

Table 4.3: The dynamic instruction set counts for the multimedia benchmarks. The first two columns present the total number of operations and instructions executed in each benchmark. The four following columns present the percentages (%) of operations and instructions that execute in vector and scalar mode respectively. The VPW column specifies the element width used in each benchmark. Because Cjpeg and Djpeg perform both 16-bit and 32-bit operations, we also provide in parenthesis the percentage of vector operations that used each VPW value. The last column presents the average vector length along with the maximum vector length for the VPW in each benchmark. For Cjpeg and Djpeg, we report the average vector length for each VPW value separately.

Convenc, we used indexed memory accesses to vectorize the inner loop that examines each input data word and updates the output branch-words. In Bital, we vectorized across the number of output buffers. A reduction is also necessary in each iteration of the outer-loop to count the number of bits allocated thus far. For Viterbi, we vectorized the generation of branch metrics and the forward sweeping of decoding states that includes the add-compare-select operations. However, we could not vectorize the back-tracking loop that produces the final output due to dependencies. Finally, we vectorized Fft manually in assembly using the intra-register permutation instructions for the butterfly stages.

The vector length in the telecommunications benchmarks depends on the size of the input data and the values of the control parameters for each kernel. For the datasets supplied by EEMBC, all benchmarks excluding Viterbi operate on relatively long vectors, with a few tens of elements each. In Viterbi, the vector length is determined by the number of decoding states, which is typically 8 or 16.

4.5.2 Dynamic Instruction Set Use

Table 4.3 presents the dynamic execution counts for the ten multimedia benchmarks. It differentiates between operations and instructions. A scalar instruction defines a single operation. Hence, for a scalar architecture the terms operation and instruction are interchangeable. On the other hand, a vector instruction specifies a number of element operations equal to the value of the vector length at the time. For a vector architecture, the instruction count indicates only the number of instructions fetched and decoded by the processor. It is the operation count that specifies the actual workload for each benchmark.

The first interesting point of Table 4.3 is the degree of vectorization for the benchmarks;

	% Arithmetic					% Misc.		% Load			% Store			*Arith.*
	±	≪	×	∑	≥	**FL**	**VP**	**U**	**S**	**X**	**U**	**S**	**X**	*Mem.*
`Rgb2cmyk`	31	–	–	–	15	–	–	–	23	–	–	31	–	0.9
`Rgb2yiq`	–	17	17	32	–	–	–	–	17	–	–	17	–	2.0
`Filter`	53	7	13	–	–	–	–	19	–	–	7	1	–	2.7
`Cjpeg`	29	8	14	–	3	2	1	12	11	1	15	4	–	1.3
`Djpeg`	31	8	11	3	3	–	–	20	4	1	10	8	1	1.3
`Autocor`	8	–	–	30	–	–	4	58	–	–	–	–	–	0.6
`Convenc`	57	–	–	–	–	–	–	31	–	–	–	12	–	1.3
`Bital`	38	13	–	–	13	22	2	6	–	–	6	–	–	5.3
`Fft`	35	9	18	–	–	–	14	10	–	7	5	2	–	2.6
`Viterbi`	40	9	–	–	9	1	–	16	4	–	5	16	–	1.4
Average	32	7	7	7	4	3	2	17	6	1	5	9	0	1.9

Table 4.4: The distribution of vector operations for the multimedia benchmarks. All columns except the last one present percentages (%) of the total number of vector operations. The four major classes of vector operations are arithmetic, miscellaneous (misc.), load, and store. The operations categories for the arithmetic class are: simple arithmetic and logical (±), shift (≪), multiply (×), multiply-add (∑), and comparisons (≥). The miscellaneous class includes flag operations (FL) and operations for permutations and vector shuffling (VP). The load and store operations are divided into unit-stride (U), strided (S), and indexed accesses (X). The last column presents the ratio of vector arithmetic to memory (load-store) operations for each benchmark.

in other words, the percentage of the total number of operations defined by vector instructions. For most benchmarks, the degree of vectorization exceeds 90%, which demonstrates the effectiveness of the vector architecture with expressing the data-level parallelism in the benchmarks. `Cjpeg` and `Djpeg` exhibit the lowest degrees of vectorization because they execute Huffman coding and decoding using mostly scalar instructions. Still, approximately 65% of operations in `Cjpeg` and `Djpeg` are due to vector instructions. It is also interesting to notice that the number of vector operations is high despite the low percentage of vector instructions. For example, only 20% of the instructions in `Convenc` are vector, but they define 97% of the overall operations.

The second noteworthy point in Table 4.3 is the average vector length. Even though long vectors are not necessary, they are desirable. Instructions that operate on long vectors can keep functional units in a vector processor busy for several clock cycles. Hence, the existence of long vectors translates to reduced instruction issue bandwidth requirements. For five benchmarks (`Rgb2cmyk`, `Rgb2yiq`, `Filter`, `Convenc`, and `Fft`), the average vector length is almost equal to the maximum. For `Autocor` and `Bital`, the average vector length is limited to approximately 60% of the maximum because of the frequent use of dot-products that progressively reduce the size of vectors. `Cjpeg`, `Djpeg`, and `Viterbi` operate mostly on short vectors with 13 to 18 elements on the average. It is interesting to note that the distribution of average vector lengths for the multimedia benchmarks is similar to the distribution reported for supercomputing applications [Esp97].

Table 4.4 describes the distribution of vector operations in each benchmark. Simple arithmetic operations and unit stride load accesses are the most repeatedly used categories across all benchmarks. Nevertheless, for almost every operation category in the instruction set, we can find at least one benchmark that makes frequent use of it. The permutation instructions are rarely used in general, but their existence is critical for the vectorization of reductions and butterflies in `Autocor`, `Bital`, and `Fft`. The only operation categories in the VIRAM architecture not used with the EEMBC benchmarks are divides and indexed stores. However, divides are frequent in 3-D

	Stride in Bytes (% op)			
Rgb2cmyk	3 (23%)	4 (31%)		
Rgb2yiq	3 (34%)			
Filter	320 (1%)			
Cjpeg	2 (1%)	3 (4%)	4 (7%)	32 (3%)
Djpeg	2 (2%)	3 (7%)	32 (1%)	
Convenc	2 (12%)			
Fft	8 (2%)			
Viterbi	688 (4%)	4 (16%)		

Table 4.5: The distribution of strides for the multimedia benchmarks. We express stride as the distance in bytes between the memory locations for two consecutive elements in a strided load or store access. The figure in parenthesis next to each stride value presents the number of memory operations that use this stride as a percentage (%) of the total number of vector operations. Unit stride accesses are not included. The distance between two consecutive elements in unit stride accesses is zero.

graphics applications, which are not represented in the EEMBC suite.

The last column of Table 4.4 presents the ratio of arithmetic to memory operations. This ratio is important for balancing the mix between arithmetic and load-store functional units in implementation of the VIRAM architecture. All benchmarks excluding `Autocor` and `Rgb2yiq` perform more than one arithmetic operation per memory access. The average ratio is approximately two arithmetic operations per memory access. For supercomputing benchmarks, the average ratio is typically closer to one [Esp97].

Finally, Table 4.5 presents the distribution of strides for the benchmarks with stride accesses. Small strides, between 2 and 4 bytes, are the most frequent across all benchmarks. Larger strides are less common and occur in kernels that process columns of two-dimensional data structures, such as `Cjpeg` and `Djpeg`.

4.5.3 Code Size

The static code size of applications is generally important for any kind of system because it affects the size, performance, and power consumption of the instruction cache. For the embedded domain, there is an additional reason for low code size. Most embedded systems store executable code in some form of non-volatile memory such as ROM or Flash. Compact code size lowers the system cost because the application can use a smaller ROM or Flash chip.

Tables 4.6 and 4.7 present the code size of the multimedia benchmarks for VIRAM and a number of alternative architectural approaches such as RISC, CISC, VLIW, and DSP. For VIRAM, we report the code size achieved both with direct compilation and after tuning the basic kernels of each benchmark in assembly. Assembly tuning is beneficial for performance because the VIRAM compiler produces unscheduled code. However, it is also advantageous with code size. Because the PDGS system was originally written for the Cray-1 architecture that had only 8 vector registers, the VIRAM compiler is often confused and produces unnecessary code that spills vector registers. It also attempts to use disjoint sets of register for different classes of operations, which leads to needless copying of vector registers. Finally, the scalar portion of the code is sub-optimal both in terms of scheduling and code density. The compiler fails to move the calculation of constants outside the loop body (code motion) and does not perform extensive elimination of common sub-expressions. However, the compiler is extremely efficient with vectorization. Its shortcomings are acceptable for an academic research project and straight-forward to improve for commercial use.

	Vector (VIRAM)		CISC (x86)	RISC (MIPS)	VLIW (Trimedia)	
	cc	**as**	**cc**	**cc**	**cc**	**cc-opt**
`Rgb2cmyk`	0.67 (0.9)	0.27 (0.4)	0.72 (1.0)	1.78 (2.5)	2.56 (3.6)	6.14 (8.5)
`Rgb2yiq`	0.52 (0.6)	0.41 (0.5)	0.89 (1.0)	1.57 (1.8)	4.35 (4.8)	34.56 (38.6)
`Filter`	1.32 (1.4)	0.70 (0.7)	0.94 (1.0)	1.99 (2.1)	4.67 (4.9)	3.58 (3.8)
`Cjpeg`	60.25 (2.0)	58.88 (1.9)	30.01 (1.0)	58.65 (1.9)	114.94 (3.8)	180.03 (6.0)
`Djpeg`	70.30 (2.0)	68.44 (1.9)	35.96 (1.0)	58.17 (1.6)	117.44 (3.3)	163.00 (4.5)
Average	(1.4)	(1.1)	(1.0)	(2.0)	(4.1)	(12.3)

Table 4.6: Static code size comparison for the consumer benchmarks. Code sizes are reported in KBytes. Next to each code size, we present in parenthesis its ratio to the code size for the x86 CISC architecture, where larger means bigger code. We differentiate between code generated with direct compilation (*cc*), with significant restructuring of the C code (*cc-opt*), and with assembly tuning of important kernels (*as*). The code sizes for architectures other than VIRAM are those from official EEMBC reports [Lev00]. The optimized code for the Trimedia VLIW architecture includes SIMD instructions [SRD96].

Table 4.6 compares the code size of VIRAM to the other architectures for the consumer benchmarks. The x86 CISC architecture produces the most compact code for consumer benchmarks because it includes variable length instructions and memory to memory operations. Compiler code for VIRAM is 1.4 times larger on the average than code for the x86 CISC architecture. Tuning in assembly reduces the ratio to x86 down to 1.1. VIRAM code is actually smaller than x86 for `Rgb2cmyk`, `Rgb2yiq`, and `Filter`. `Cjpeg` and `Djpeg` include thousands of lines of code for error checking and for emulating file-system services, which are not vectorized and lead to large code size. RISC code for the MIPS architecture is 2.0 times larger than x86. VLIW code with direct compilation for the Trimedia architecture is 4.1 times larger than x86. If the C code is modified for maximum performance on a VLIW processor, the ratio between VLIW and x86 bloats to 12.3.

Table 4.7 compares the code size of VIRAM to the other architectures for the telecommunications benchmarks. The DSP architecture by Analog Devices (ADI) produces the most compact code for consumer benchmarks because it includes features such as zero-overhead loops and special addressing modes for reductions and FFT. Compiler code for VIRAM is 4.6 times larger on the average than code for DSP. Tuning in assembly reduces the ratio down to 2.0. Code for x86 is 7.5 times larger than DSP, mostly because of the poor code density for `Convenc` and `Fft`. RISC code for MIPS is 8.0 times larger than DSP. The VLIW architecture in this case is the TMS320C6 [Tru97], which implements VLIW instructions on top of a basic DSP architecture. Straight-forward compilation produces code 5.8 times larger than DSP. C-level optimizations for VLIW lead to code sizes 7.8 times larger than DSP.

The compact code size of VIRAM is due to four basic reasons. First, each vector instruction captions a large amount of data-level parallelism. Therefore, there is little need for static scheduling techniques such as software pipelining and loop unrolling [Muc97], which increase instruction-level parallelism but lead to bloated code size. This advantage is particularly obvious when comparing VIRAM to the VLIW approaches, especially if the VLIW code is tuned for performance. Optimized VLIW code is 2 to 10 times larger than code for VIRAM. Of course, VLIW code incurs the additional overhead of empty slots in long instructions [Fis83]. The second beneficial factor for VIRAM is that vector memory instructions capture operations for address handling such as indexing, scaling, and auto-increment. RISC, CISC, and VLIW architectures must execute several instructions around each load or store to capture this functionality. Furthermore, the permutation instructions in VIRAM allow compact descriptions of dot-products and butterfly primitives. In contrast, CISC, RISC,

	Vector (VIRAM)		CISC (x86)	RISC (MIPS)	VLIW (TI TMS320C6)		DSP (ADI)
	cc	**as**	**cc**	**cc**	**cc**	**cc-opt**	**cc**
`Autocor`	1.0 (3.7)	0.3 (1.1)	0.5 (1.9)	1.1 (3.9)	0.9 (3.2)	1.5 (5.1)	0.3 (1.0)
`Convenc`	0.7 (6.2)	0.3 (3.3)	0.8 (7.5)	1.6 (15.4)	1.1 (10.6)	2.0 (19.2)	0.1 (1.0)
`Bital`	1.0 (2.9)	0.6 (1.7)	0.7 (1.9)	1.5 (4.2)	2.3 (6.5)	1.4 (4.0)	0.4 (1.0)
`Fft`	– (–)	0.7(1.1)	15.7 (23.0)	5.5 (8.0)	3.0 (4.3)	3.5 (5.2)	0.7 (1.0)
`Viterbi`	2.6 (5.7)	1.1(2.5)	1.3 (3.0)	3.8 (8.4)	1.9 (4.2)	2.6 (5.6)	0.4 (1.0)
Average	(4.6)	(2.0)	(7.5)	(8.0)	(5.8)	(7.8)	(1.0)

Table 4.7: Static code size comparison for the telecommunications benchmarks. Code sizes are reported in KBytes. Next to each code size, we present in parenthesis its ratio to the code size for the DSP architecture by Analog Devices (ADI), where larger means bigger code. We differentiate between code generated with direct compilation (*cc*), with significant restructuring of the C code (*cc-opt*), and with assembly tuning of important kernels (*as*). The code sizes for architectures other than VIRAM are those from official EEMBC reports [Lev00]. The TMS320C6 VLIW architecture includes several features of DSP architectures [Tru97].

and VLIW architectures require complicated loops to express the necessary permutation patterns. Finally, the use of vector instructions allows us to eliminate the instructions for maintaining the loop index and branching in several small loops.

It is interesting to notice that VIRAM produces smaller code than MIPS, the RISC architecture it is based on. The vector instructions we added to the MIPS architecture function as useful "macro-instructions" that describe a large number of operations with a 32-bit instruction word.

4.5.4 Basic Block Size

Table 4.8 presents the average basic block size for the multimedia benchmarks. A basic block is a sequence of instructions that does not include branches, jumps, or other instructions for control transfer [HP02]. The average basic block size ranges from 4 to 29 instructions with an average of 14.0. Previous studies for scalar architectures have reported basic block sizes for consumer and telecommunications benchmarks between 4 and 7 [GRE+01]. There are two reasons for this difference. Vectorization removes the loops for routines with short vectors, such as 2-D DCT or small reductions. Hence, the vector code includes fewer branches. In addition, vectorization introduces extra coprocessor instructions in each basic block for moving scalar values between the scalar and vector components of a vector processor.

However, the most interesting point of Table 4.8 is the average basic block size in terms of operations. It represents that average amount of work between two branch instructions. The number of operations per basic block ranges from 35 for benchmarks with short vector lengths (`Cjpeg` and `Djpeg`) to 1666 for kernels that process long vectors. The average size is 502.1 operations. The large basic block size implies that the latency of resolving branches is not a critical performance issue for vector processors. Even if it takes 10 cycles to determine whether a branch is taken in a long vector pipeline, the cost is amortized over hundreds of vector operations and becomes negligible. Hence, a vector processor can use a long pipeline and does not require hardware such as branch predictors or branch target buffers.

On the other hand, a processor that implements a scalar architecture has a very small number of instructions (operations) between every two branches. Therefore, it must use a rather short pipeline and include sophisticated branch prediction logic in order to keep the branch latency and its impact on performance low. Superscalar implementations exacerbate the problem because

	Basic Block in Instructions	Basic Block in Operations	Op/Inst Ratio
`Rgb2cmyk`	18.0	1666.3	92.6
`Rgb2yiq`	28.9	1161.1	40.2
`Filter`	4.0	533.3	133.3
`Cjpeg`	12.4	35.3	2.8
`Djpeg`	14.8	45.4	3.1
`Autocor`	7.6	114.6	15.1
`Convenc`	3.9	128.9	33.1
`Bital`	10.7	314.5	29.4
`Fft`	24.8	903.5	36.4
`Viterbi`	15.0	117.7	7.8
Average	14.0	502.1	39.4

Table 4.8: The average basic block size for the multimedia benchmarks in VIRAM. The first column presents the basic block size in terms of instructions. The second column presents the basic block size in terms of operations. The last column presents the ratio the basic block size in operations to the size in instructions.

they can execute the few instructions in each basic block faster.

4.5.5 Comparison to SIMD Extensions

Most commercial versions of RISC, CISC, and VLIW architectures have introduced SIMD extensions for handling the data-level parallelism in multimedia benchmarks [Ric96, SRD96, Phi98, Int00]. Even though, SIMD extensions are similar to vector instructions, there are significant differences that limit the benefits of SIMD.

SIMD extensions define short vectors of narrow data within the scalar registers of the corresponding architectures and a set of instructions to perform arithmetic operations on them. With 64 to 128 bits per register, a SIMD instruction can operate on up to 4 32-bit elements or up to 8 16-bit elements [SS01b]. However, Table 4.3 (page 30) shows that the multimedia benchmarks have significantly higher degrees of data-level parallelism. Even for `Cjpeg` and `Djpeg`, vectors of 13 32-bit elements are available. By capturing with each instruction only a small percentage of the data-level parallelism available, a processor with SIMD extensions must execute a larger number of instructions than a vector processor. Consequently, the processor with SIMD extensions requires higher instruction issue bandwidth and consumes more power. In addition, the basic block size with SIMD extensions is significantly smaller than that with vector instructions, hence sophisticated branch resolution support is also necessary.

An additional shortcoming of SIMD extensions is the lack of support for accessing vectors in memory. A compiler for SIMD must use scalar load and store instructions to emulate the functionality of unit stride, strided, and indexed vector accesses. To handle stride, indexing, and alignment issues, a variety of pack, unpack, rotate, and merge instructions are also necessary. The overhead of these instructions can often cancel any performance or code size benefits from using SIMD arithmetic. On the other hand, the load and store instructions in a vector processor handle strides, indexing, and alignment in hardware. They also encapsulate the memory access pattern and allow efficient implementation of prefetching hardware.

4.6 Evaluation of the VIRAM ISA Decisions

In this section, we use the experience from the EEMBC benchmarks to revisit some of the basic decisions in the VIRAM instruction set and discuss their effectiveness. It is likely that some of the following statements could be be somewhat different with another benchmark set. However, we consider the consumer and telecommunications benchmarks in the EEMBC suite to be representative of multimedia applications, hence it is meaningful to draw conclusions from them.

The Use of the Coprocessor Model

Defining VIRAM as a coprocessor extension instead of a stand-alone instruction set introduces some overhead due to the instructions that move data between the scalar register files in the MIPS core and the scalar register files in the vector unit. For the EEMBC benchmarks, however, the additional move instructions account for less than 6% of the static code size and less than 2.5% of the dynamic operations count. Hence, the use of the coprocessor model does not have a significant effect on code size or benchmark performance.

Specifying Data Width with the VPW Register

The use of the `VPW` control register for specifying the data width for vector operations reduces by 3 times the number of opcodes necessary for vector instructions when compared to an ISA that encodes the data width in each instruction word. In addition, the overhead for setting the `VPW` register with coprocessor move instructions is negligible. None of the EEMBC benchmarks could use more than one data width for the various instructions in each loop. Therefore, setting the `VPW` once per loop does not create the potential for wasting performance or energy in processor implementations of the VIRAM architecture.

The Use of Vector Registers

VIRAM defines 32 vector registers. However, only two EEMBC benchmarks, `Cjpeg` and `Djpeg`, reference more than 15 vector registers in their code. Hence, with proper use of the valid and dirty bits, we have to save less than half of the vector architecture state during context switches of vector applications. In addition, `Cjpeg` and `Djpeg` use rather short vectors with 13 32-bit elements on the average in the functions for DCT transformations that reference 25 vector registers. If an implementation can track the maximum vector length used by an application, we can significantly reduce the amount of vector state involved in context switches for these two benchmarks by only saving the valid elements in each register. Therefore, despite the large amount of vector state defined in VIRAM, the actual overhead for context switches of vector applications can be kept low.

The Use of Flag Registers

Only three of the EEMBC benchmarks make use of the support for conditional execution available in the VIRAM ISA (`Cjpeg`, `Bital`, `Viterbi`). None of these programs references more than 4 flag registers at any time. Thus, the 16 flag registers defined in VIRAM are more than sufficient. In addition, the three benchmarks use 3 or less distinct sets of masks per loop iteration for conditional execution of arithmetic and memory instructions, including the fully set mask for unconditional execution. Thus, the use of a single bit in each VIRAM opcode to identify one of two default flag registers as the source of masks for conditional execution is a reasonable compromise between efficient use of opcode space and ease of use. The overhead for moving up to three sets of masks in and out of the two default flag registers is small.

Shuffling vs. Permutation Instructions

None of the ten benchmarks uses the traditional instructions for shuffling elements in a vector register (insert, extract, compress, and expand). Hence, from the EEMBC benchmarks alone, one could conclude that the shuffling instructions are not necessary in a vector architecture for multimedia. On the other hand, the simpler permutation instructions of VIRAM were instrumental in vectorizing the reductions in `Autocor` and `Bital`, and the butterflies in `Fft`.

Address Post-increment for Load-Store Instructions

All ten benchmarks use the address post-increment ability of vector load and store instructions for most of their memory accesses. We could increment the base address register for vector instructions using coprocessor moves and scalar add instructions. However, specifying the increment operation with each vector load and store instruction reduces static code size by up to 20% for some of the smaller benchmarks and helps minimize the overhead of communication over the coprocessor interface. In addition, address post-increment is easy to use in the compiler.

4.7 Related Work

Although the majority of work on vector architectures has focused on super-computing environments, certain research groups have investigated their advantages with multimedia applications.

The T0 extension to the MIPS-II architecture was one of the first vector instruction sets for multimedia [AJ97]. It has been a strong influence to the VIRAM architecture. Developed for speech processing applications, T0 provides vector instructions for fixed-point operations on 32-bit data and supports conditional execution using conditional merges. It does not support multiple data sizes, element permutations, or running an operating system. The T0 vector microprocessor implemented the T0 instruction set [WAK+96]. Lee extended the T0 architecture to support vector floating-point operations for a set of architectural and compiler studies using simulations [SL99].

The popularity of multimedia applications has also motivated researchers to support them within instruction sets other than vector. SIMD extensions to existing architectures and streaming instruction sets are the most prominent approaches.

Virtually all RISC, CISC, and VLIW architectures include SIMD extensions for exploiting sub-word parallelism in multimedia programs [Ric96, SRD96, Phi98, Int00]. Several commercial processors have implemented the corresponding extensions. Even though there are great differences in their features, the trend is towards introducing new wide register files for multimedia data and supporting an ever-increasing variety of operations for both integer and floating-point arithmetic. We discussed the general disadvantages of SIMD extensions when compared to vector architectures in Section 4.5.5. Due to their shortcomings and the frequent updates to their features, few compilers can generate code for SIMD extensions. In addition, several studies have demonstrated that simple vector processors can outperform complicated out-of-order designs with SIMD extensions for multimedia by factors larger than 3 [SL99, Asa98].

The MOM matrix instruction set is a proposal for a SIMD variant that supports operations on fixed-size, two-dimensional arrays of narrow numbers [CVE99]. It has been motivated by video processing algorithms, where motion estimation operates on 8×8 or 16×16 arrays of pixels. MOM inherits all the deficiencies of SIMD extensions and has limited use outside video processing. It has never been implemented in a commercial or research chip.

The Imagine stream architecture defines a two-level register file hierarchy with the lower level associated with each functional unit [RDK+98]. Microcoded instructions configure the operation executed in each unit and the way functional units are connected. A program executes by streaming data through the processor and having each unit apply its operation on them. Imagine

provides a programmer with flexible control over a large number of register and execution resources at the cost of complicated programming. To generate code for a stream processor, a compiler must likely adopt a vector or SIMD approach for modeling the parallelism in the application, and then map the vector or SIMD operations one the stream hardware. Stream architectures are practical for specialized accelerators with a separate memory system, but provide limited support for general-purpose environments running an operating system.

4.8 Summary

In this chapter, we presented the VIRAM instruction set for multimedia processing on general purpose systems. Vector instructions already provide a compact way to express data-level parallelism. The additional features required for multimedia applications are support for narrow data types and fixed-point numbers, instructions for permutations of vector elements, and a method for conditional execution of element operations. To simplify software development and use with complex operating systems, the instruction set includes support for virtual memory and semantics that allow fast context switches.

The instruction level analysis demonstrates that the degree of vectorization for the multimedia benchmarks with the VIRAM architecture exceeds 90% in most cases. Even for applications that are partially vectorized and include short vectors such as `Cjpeg` and `Viterbi`, the degree of vectorization is higher than 65%. The code size for VIRAM executables is comparable to that of the most compact architectural approach in both consumer and telecommunications benchmarks. VIRAM code is 2 time smaller than code for RISC architectures and up to 10 times smaller than code for VLIW processors.

In the next two chapters, we will discuss two micro-architectures that implement the VIRAM instruction set. Chapter 5 presents a simple micro-architecture with wide functional units, which uses deep pipelining and embedded DRAM technology. Chapter 6 introduces a composite and decoupled organization, which is the basis of a scalable family of vector processors.

Chapter 5

The Microarchitecture of the VIRAM-1 Processor

"What we have to learn, we learn by doing it."
Aristotle

A high-end microprocessor typically relies on high frequency operation in order to deliver high performance. Consequently, it consumes a large amount of power. Moreover, it requires hundreds of engineers for a period of three to five years for development, verification, and testing. This chapter presents the microarchitecture and implementation of VIRAM-1, a processor that exploits the explicit parallelism expressed by vector instructions to provide high performance at low energy consumption and reduced design complexity.

Section 5.1 highlights the goals of the VIRAM-1development. In Section 5.2, we present in details the microarchitecture, focusing mostly on the vector and memory hardware. Section 5.3 describes the implementation methodology that allowed a group of six graduate students to design VIRAM-1 within an academic environment [1]. Section 5.4 presents a performance analysis of VIRAM-1 for the multimedia benchmarks. Finally, in Section 5.5 we discuss the lessons learned from the VIRAM-1 development, its basic advantages and some further challenges we need to address.

5.1 Project Background

The motivation for implementing VIRAM-1 was to provide practical experience and realistic insights in developing vector microprocessors for the emerging domain of multimedia processing in embedded systems. As explained in Chapter 2, the characteristics of this application domain are significantly different from those of desktop and server systems. Therefore, we embarked on a full processor implementation in order to provide a convincing proof of concept and expose the whole spectrum of issues in microarchitecture and design methodology. In addition, a working processor prototype provides an attractive platform for compiler and software development, which are both necessary for a complete system demonstration.

The design objectives for VIRAM-1 were directly derived from the project motivation discussed in Chapters 1 and 2. We aimed at developing a vector processor that provides high performance for multimedia tasks at low energy consumption. To simplify the software development of real-time applications, we wanted performance to be predictable and not to rely on probabilistic

[1] The development of VIRAM-1 was joint work with Joseph Gebis, Samuel Williams under the guidance of David Patterson and Katherine Yelick at U.C. Berkeley. Hiroyuki Hamasaki, Ioannis Mavroidis, and Iakovos Mavroidis also made significant contributions to the VIRAM-1 design.

techniques such as caching and speculation. An equally important goal was to develop a simple and scalable design. Unlike superscalar hardware, the vector processor should be easy to build with a small group of designers and scaling to the next generation implementation should be an incremental effort, not a complete redesign. Finally, by utilizing the mature compiler technology for vector architectures, we aimed at developing an efficient processor with a software development model based on high-level languages and automatic compilation.

To maintain the project focus on the most important goals and simplify the design task, certain potential features became secondary or were precluded. The performance and energy characteristics of VIRAM-1 rely on architectural and microarchitectural features and not on specialized circuitry for high clock frequency or low power consumption. We did not engage in advanced circuitry development because of its complexity and the fact that any performance or power benefits it introduces are orthogonal to those achieved with architectural methods. We also focused on single processor systems, providing no special support for shared memory or message-passing in a multiprocessor environment. Finally, we did not integrate a large number of IO interfaces on the same die with the processor. Even though most embedded processors include a variety of interfaces, pushing their functionality to an external chip-set allowed for faster prototyping. A future, commercial implementation of the VIRAM-1 microarchitecture could reverse any of these choices with little impact to the basic characteristics reported in this thesis.

5.2 The VIRAM-1 Organization

The VIRAM-1 microarchitecture [Koz99, KGM+00] relies on two basic technologies: vector processing and embedded DRAM. Each technology contributes a set of complimentary features towards meeting the overall design goals.

The vector architecture allows for high performance for multimedia applications by executing multiple element operations in parallel. The control logic for parallel execution has reduced complexity and energy overhead because element operations within a vector instruction are independent by definition. In Section 5.3, we also present a modular implementation for vector hardware, which reduces design complexity and improves scalability.

Embedded DRAM technology enables the integration of a large amount of DRAM memory on the same die with the processor. It provides the high bandwidth memory system required for a vector processor [ST92] in a cost effective way. Vector memory instructions can hide long latencies for element transfers by utilizing high bandwidth for sequential and random accesses. The high density of embedded DRAM reduces the energy consumption in the memory system by decreasing or eliminating the number of transfers that must access off-chip memory through high capacitance board busses. The embedded DRAM memory system can also match the modularity of the vector processor if organized as a collection of independent banks.

Figure 5.1 shows the overall block diagram for VIRAM-1 with its four basic components: the MIPS scalar core, the vector coprocessor, the embedded DRAM main memory, and the external IO interface. We discuss each component in details in the subsequent subsections.

5.2.1 Scalar Core Organization

VIRAM-1 uses the m5Kc scalar core by MIPS Technologies [Hal99a]. It is a single issue, in-order core with a six-stage pipeline, which implements the MIPS-64 architecture [MIP01]. It contains 8-KByte, first-level instruction and data caches and a 32-entry TLB. It also includes a coprocessor interface to which we have attached a floating-point unit for scalar operations on single precision numbers.

The coprocessor interface is also the only way the scalar core connects to the vector hardware. Consequently, vector instructions cannot access scalar registers in the MIPS core and addi-

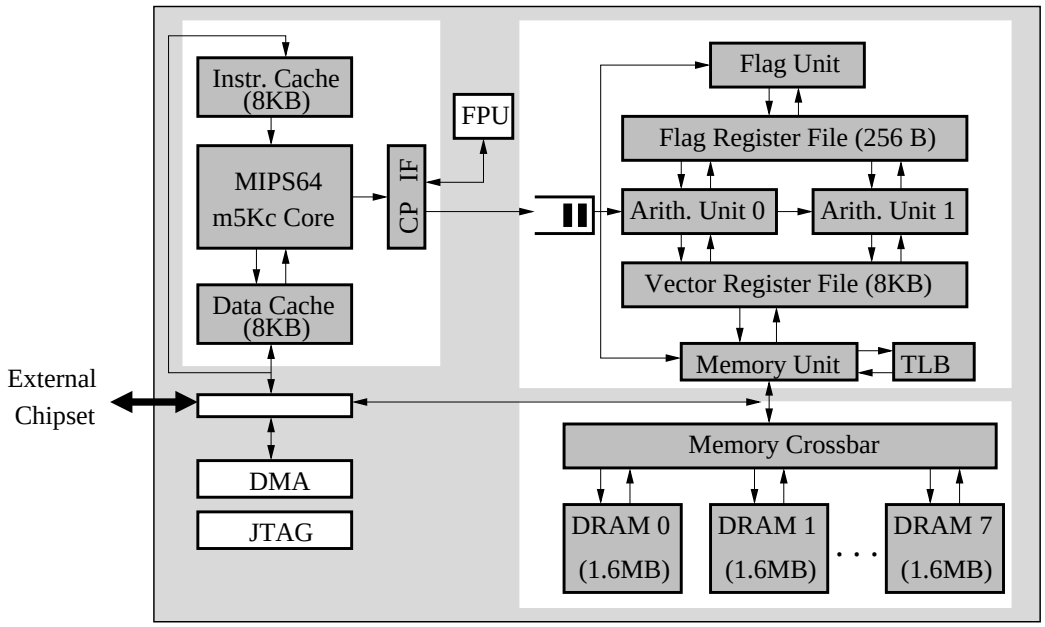


Figure 5.1: The block diagram of the VIRAM-1 vector processor.

tional scalar register files are necessary in the vector hardware, as we discussed in Chapter 4. In addition, the two components cannot share the TLB or the logic for memory instructions. Apart from saving die area, sharing these resources would also simplify exception processing and memory consistency for scalar and vector accesses. However, the clear separation of vector and scalar hardware allowed us to start developing the vector hardware long before selecting the specific scalar core. It also allowed independent verification and testing of the two, which was simpler and faster.

5.2.2 Vector Coprocessor Organization

The vector hardware executes the VIRAM instructions and connects to the MIPS core as coprocessor 2. An instruction queue decouples the vector coprocessor from the scalar core, allowing vector execution to proceed even when the scalar core stalls due to cache misses or external interrupts.

Figure 5.1 also shows the internal organization of the vector coprocessor. There are four functional units, two for arithmetic instructions, one for flag operations, and one for vector load and store accesses. Arithmetic unit 0 can execute integer, fixed-point, and single-precision float-point operations. Due to area restrictions, arithmetic unit 1 has neither floating-point hardware nor an integer multiplier. However, arithmetic unit 1 implements the element permutation instructions. Each arithmetic unit includes four 64-bit partitioned datapaths, which allows the execution of four 64-bit, or eight 32-bit, or sixteen 16-bit element operations in parallel. In other words, for narrower data types (VPW), VIRAM-1 can sustain a larger number of operations per cycle.

The memory unit handles vector and flag load or store instructions. On every clock cycle, it can exchange up to 256 data bits with the memory system. It generates up to four memory addresses per cycle for strided and indexed accesses. For sequential accesses, a single address is sufficient for a 256-bit transfer. The memory unit contains a 32-entry TLB, similar to the one in the MIPS core. Even though the TLB itself has a single access port, it can translate four addresses in parallel using a four-entry, four-ported micro-TLB, which provides caching for the most frequently accessed TLB entries. The micro-TLB is controlled by hardware, while software manages any misses or exceptions

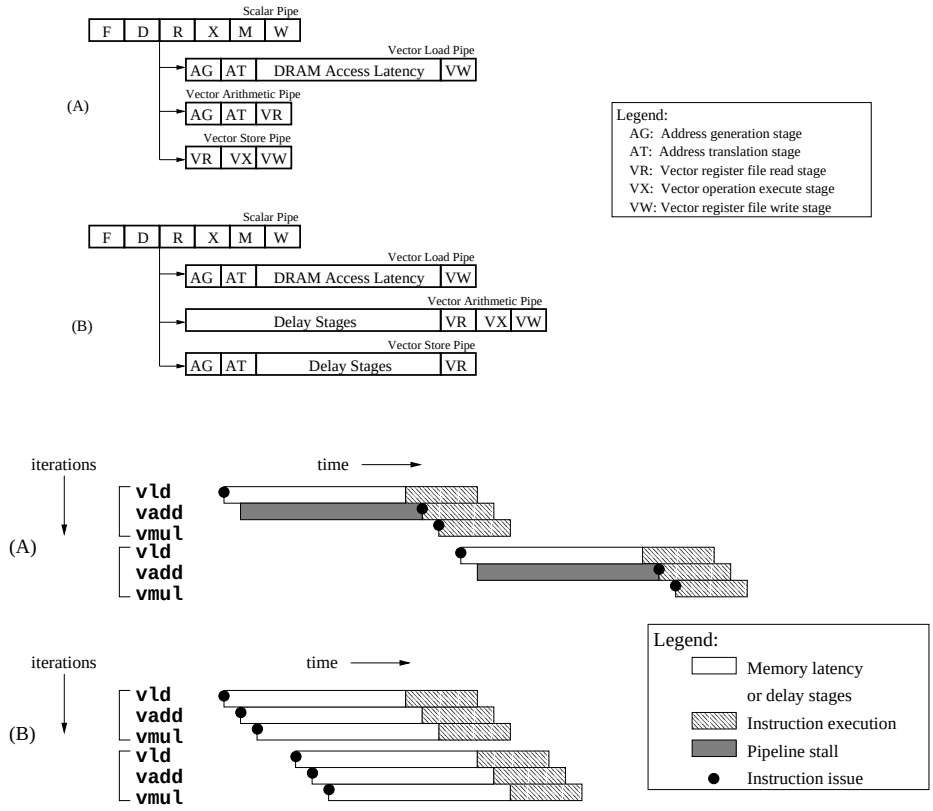


Figure 5.2: The simple and delayed pipeline models for a vector processor and the corresponding timings for executing two iterations of a simple vector loop. With the simple model (a), dependencies between memory and arithmetic instructions within a loop lead to long stall periods on every iteration. The delayed pipeline (b) introduces idle stages in order to match the pipeline length for arithmetic instructions to that for worst-case memory accesses. In other words, we delay the execution of arithmetic operations until the necessary data arrive from memory without stalling the pipeline.

in the main TLB.

The 8-KByte vector register file is at the heart of the coprocessor . It can store 32 64-bit, or 64 32-bit, or 128 16-bit elements per vector register. The register file should have seven read and three write ports to provide operands to the arithmetic and memory units. To reduce the area and energy consumption of the register file by approximately 30%, we used two SRAM banks, each with four read and three write ports. The first bank stores all the odd numbered 64-bit elements for each vector, with the second bank holding all even elements. When executing a vector instruction, the functional units alternate between the two banks on every cycle. If all units try to access the same bank on the same cycle, we need to stall one of them for one cycle in order to restore conflict free access to the register file.

5.2.3 Vector Pipeline

VIRAM-1 executes vector operations in a pipelined manner, without any additional startup overhead for new instructions. On every clock cycle, the control logic pushes a set of element operations down the pipeline of the proper functional unit. The number of element operations

processed concurrently may be 4, 8, or 16, depending on the virtual processor width used at the time.

The main challenge for the pipeline structure is to tolerate long latencies for load and store accesses to DRAM memory. A random access to embedded DRAM is pipelined but requires up to eight clock cycles. Without special support, the pipeline would have to stall for every arithmetic operation dependent to a memory access. Figure 5.2 presents the delayed pipeline model used in VIRAM-1 to tolerate DRAM latency. It introduces sufficient pipeline stages to handle the worst-case memory latency and pads the arithmetic pipelines so that operations execute after any previously issued loads have fetched the necessary data. Consequently, dependent load and arithmetic operations can issue to the pipeline back to back without any stall cycles. The pipeline length for all functional units in VIRAM-1 is 15 stages.

An alternative organization for tolerating memory latency is the decoupled pipeline [EV96]. It uses data and instruction queues to allow memory accesses to execute as soon as possible and eliminate unnecessary pipeline stalls. Although the decouple pipeline can tolerate longer latencies than the delayed one, we chose the latter for VIRAM-1. The delayed pipeline is simple to implement, has no power and area overhead for queues, and can handle the moderate latency of embedded DRAM. In Chapter 7, we introduce a vector microarchitecture that implements the decoupled pipeline without data queues.

5.2.4 Chaining Control

Chaining is the vector equivalent of pipeline forwarding [HP02]. It allows dependent vector instructions to overlap their execution by forwarding the result of element operations before the whole instruction completes. As long as dependencies between vector elements are preserved, dependencies between vector registers are relaxed. VIRAM-1 allows instructions executing in different functional units to chain for all three types of dependencies on vector registers: read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW). The vector register file implements the result forwarding without any additional busses. For a RAW case, the second instruction can read a vector element in the same cycle the previous instruction writes it to the register file.

The implementation of chaining is simple with the delayed pipeline, because it requires chaining decisions to be made only once per instruction. Before issuing the first set of element operations for a new instruction, the control logic checks the first few stages in the pipeline of all functional units for data dependencies with previously issued instructions. The check is distributed, with every pipeline stage checking for dependencies between the first element operation of the new instruction and the element operations it holds. Once the first set of element operations can proceed, it is safe to issue the remaining operations in following clock cycles without checking, because the pipeline is in-order and preserves any dependencies between the two instructions.

Chaining is not allowed for the four shuffling instruction, because of the complex logic required to determine dependencies to their element operations. On the other hand, it is easy to support chaining for the permutation instructions, for which the dependency checks are simple due to their regularity.

5.2.5 Memory System and IO

The memory system for VIRAM-1 consists of 13 MBytes of embedded DRAM. It serves as main memory, not a cache, for both vector and scalar accesses. There is no SRAM cache between the vector coprocessor and DRAM.

We organized the memory system as eight independent DRAM banks. Multiple banks allow overlapping of transfers for indexed and strided accesses, but introduce some area overhead for control logic. Eight banks was a reasonable compromise between performance and area efficiency. The bank design available to us is a monolithic array of 6,656 rows with eight 256-bit columns per

row. A column access to a previously open row can occur every one and a half cycles. A random access that requires opening a new row can take place every five processor cycles. The bank interface includes separate 256-bit buses for input and output data.

Alternatively, we could organize each DRAM bank as a collection of sub-banks sharing a single data and control interface. Multiple sub-banks allow overlapping of accesses to different rows within one bank in a pipelined manner, reducing the number of stalls due to bank conflicts. Unfortunately, the bank design available for VIRAM-1 did not include sub-banks, resulting to a performance penalty for applications with indexed and strided accesses [KJG+01]. We will discuss and evaluate sub-banks and other options for organizing an embedded DRAM memory system in Chapter 8.

A custom crossbar connects the memory system to the vector and scalar hardware. It can transfer 256 data bits per direction, load or store, to the eight DRAM banks and issue up to four memory addresses per cycle. We did not use a simpler bus structure because it would be difficult to support many concurrent transfers and the large number of configurations required for indexed and strided accesses. In addition, a bus is difficult to scale up to eight or more banks. On the other hand, because the memory and the processor are on the same die, it is easy to provide the wiring and switching resources necessary for the crossbar.

VIRAM-1 has a simple input-output interface that includes a system bus and a DMA engine. The system bus uses the SysAD protocol [Hei94] and provides connectivity to chipsets for 64-bit MIPS processors. The chipsets allow VIRAM-1 to connect to off-chip memory and a variety of peripheral devices such as audio and video interfaces, hard disks, and networking chips. The DMA includes two channels that can transfer data between the on-chip memory and external devices without occupying the processor. The DMA capability is important for many multimedia tasks because it allows fetching the next set of audio samples or video frames to the on-chip memory, while the application processes the current set [KMK01].

5.2.6 System Support

Virtual memory exceptions on VIRAM-1 are imprecise but restartable. On a vector memory fault, the delayed pipeline stalls and all stages maintain their state. The scalar core receives the exception and invokes the proper handler, which can service the exception through a series of instructions that manipulate the vector TLB. When the exception condition has been removed, the pipeline is released and vector execution continues from the point it was interrupted. If the handler decides to preempt the application currently running, it must also save the state of the first three stages of the delayed pipeline. When the application resumes, the handler must restore the pipeline state during the context switch. The original implementation of the Alpha architecture also used imprecise exceptions with saving and restoring of microarchitecture state for floating-point operations [Sit92, Dig96].

As discussed in Chapter 4, VIRAM-1 maintains dirty and valid bits for vector registers and tracks the maximum vector length used by an application in order to reduce the context switch time. It also allows marking the vector coprocessor unusable after a context switch in order to avoid saving and restoring vector state when switching to short handlers or applications without vector instructions. Moreover, the instruction queue that decouples the scalar core from the coprocessor allows vector hardware to proceed with the instructions it has received while the MIPS core executes an interrupt handler. Because vector instructions specify a large number of element operations, small IO handlers can run on the scalar core without underutilizing the vector coprocessor.

Special support is also available for maintaining memory consistency between the scalar and vector memory accesses. Because the two components have separate paths to memory, a scalar access can reach DRAM before a previously issued vector access to the same address. To assist compilers with preventing errors due to such cases, VIRAM-1 provides a set of synchronization instructions that can stall the whole processor until all currently issued accesses have completed. To

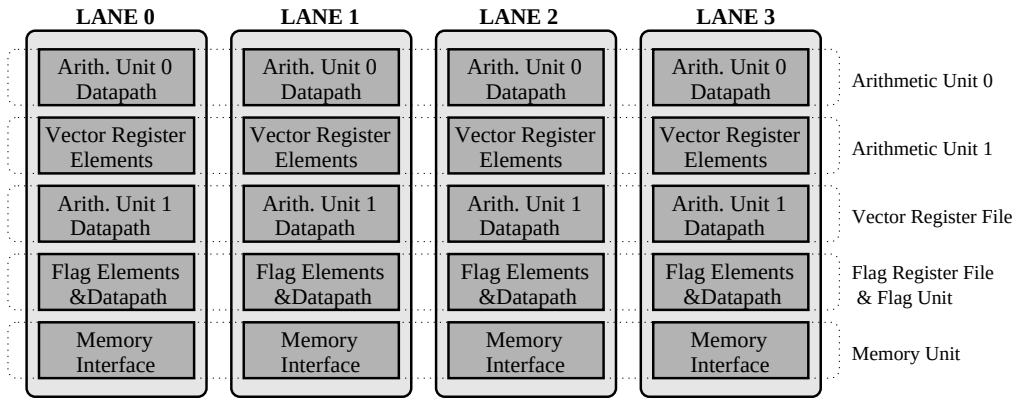


Figure 5.3: The vector datapath and register file resources of VIRAM-1 organized in four vector lanes. Functional units span across the design by contributing one 64-bit datapath to each lane. The elements of each vector register are assigned to lanes in an interleaved fashion. Two 64-bit, inter-lane busses are necessary to implement the permutation instructions for reductions and FFTs.

reduce the overhead, we support variations that wait only for a specific type of accesses to complete (vector or scalar, load or store). On vector stores, we also check to the data cache of the scalar core for data from the same address and invalidate them if present. To avoid unnecessary cache stalls due invalidation traffic, we maintain a duplicate set of tags for the invalidation checks.

5.3 The VIRAM-1 Implementation

VIRAM-1 includes a large amount of hardware resources, which typically means high design complexity and long interconnects within the chip. In this section, we present the design approach and methodology that organizes hardware in a modular fashion and leads to locality of interconnect.

5.3.1 Scalable Design Using Vector Lanes

Figure 5.3 presents the implementation of the vector coprocessor using four parallel lanes. Each lane contains a 64-bit datapath from each functional unit, a 64-bit interface to the memory crossbar, and a vertical partition of the two banks for the vector register file. All lanes are identical and receive the same control on every cycle. To execute a vector instruction, the datapaths in each lane operate on the elements stored in the local partition of the register file. Therefore, most vector instructions can execute without any data communication between lanes.

The use of lanes makes the vector coprocessor modular. We only need to design and verify a single 64-bit block and replicate it four times, which is simpler than implementing a separate 256-bit blocks for each functional unit and interconnecting them later. We can also scale the performance, area, and energy consumption of the coprocessor without significant redesign by allocating the proper number of lanes. A single design database can produce a number of coprocessor implementations, with a large or small number of lanes. All implementations are balanced in terms of hardware resources as each lane contributes both execution datapaths and storage for vector elements.

Lanes also allow us to trade-off die area for reduced power consumption [KJG+01, BCS93]. Increasing the number of lanes allows for a proportional decrease in the clock frequency and power supply without changing the peak computational throughput of the system. Because power consumption is an exponential function of power supply voltage but only a linear function of the switching

Technology	IBM SA-27E 0.18μ CMOS Process 6 metal layers (copper) deep trench DRAM cell (0.56μm^2)
Area	318.5 mm^2 (17.5mm x 18.2mm)
Transistors	120 million (7.5 logic, 112.5 DRAM)
Clock Frequency	200 MHz
Power Supply	1.2V logic, 1.8V DRAM, 3.3V IO
Power Consumption	2 Watt (average) (vector 1 Watt, scalar 0.5Watt, DRAM 0.5Watt)
Package	304-pin quad ceramic package
Peak Performance	Integer: 1.6/3.2/6.4 Gops/second (64-bit/32-bit/16-bit) Fixed-point: 2.4/4.8/9.6 Gops/second (64-bit/32-bit/16-bit) Floating-point: 1.6 Gflops/second (32-bit)

Table 5.1: The chip statistics for VIRAM-1. Peak performance for integer and fixed-point operations indicates the maximum operation throughput for the three supported virtual processor widths (64-bit, 32-bit, 16-bit). Fixed-point performance ratings assume two operations per multiply-add. One Gop and one Gflop are 10^9 integer and floating-point operations respectively.

capacitance of the circuits, the new processor with more lanes and lower clock frequency consumes less power than the original design. Hence, we can exploit the shrinking geometries in CMOS technology to introduce more lanes and reduce power consumption without hurting performance.

Furthermore, the ability to integrate additional lanes with future semiconductor processes does not create long interconnect wires that could limit performance. Excluding memory and permutation instructions, all other instructions require no inter-lane communication during their execution. Most wires are limited within each lane and are short, regardless of the size of the overall processor. Hence, the increasing delay of cross-chip wires is not a significant scaling problem for the vector coprocessor. For the long wires needed for memory transfers and permutations, pipelining is an appropriate solution because a vector processor can tolerate latency as long as high bandwidth is available.

5.3.2 Design Statistics

We designed VIRAM-1 in a 0.18μm bulk CMOS process with embedded DRAM. Table 5.1 summarizes its design statistics. The die occupies 17.5x18.2mm^2 and includes approximately 120 million transistors. It is designed to operate at 200 MHz, a conservative clock frequency for this technology, in order to simplify circuit design and decrease energy and power consumption. However, it can support up to 6.4 billion operations per second for 32-bit integer numbers by executing element operations in parallel on the four vector lanes. The average power consumption is 2 Watts, but the use of low power circuitry and a reduced power supply for the DRAM could drop it to less than 1 Watt in more aggressive implementations. Nevertheless, at 3.2 billion operations per second per Watt, VIRAM-1 is an order of magnitude better in power-performance than most high performance processors in 2002 [JYK+00, Wor01, Emb01].

Figure 5.4 presents the VIRAM-1 chip floorplan. It illustrates the overall design modularity as it consists of four lanes and eight memory banks. Apart from its benefits to design complexity and scalability, modularity is also good for improving yield. A modular vector processor can include an additional vector lane or DRAM bank for redundancy. Alternatively, a chip with permanent defects on some lane or memory bank can still function and be useful as a lower performance or lower capacity part. Table 5.2 presents the area breakdown for the chip. It indicates that the majority of

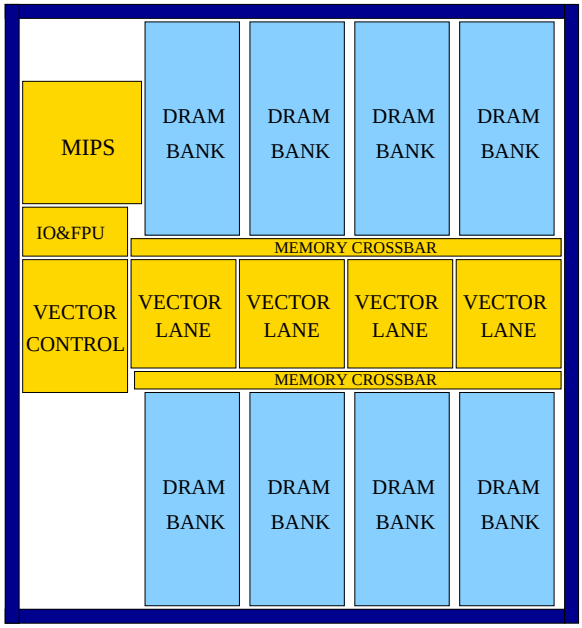


Figure 5.4: The floorplan of the VIRAM-1 processor. To fit in a square die, the DRAM banks are organized in two groups of four, with a copy of the crossbar to connect each group to the scalar and vector hardware. All blocks are drawn to scale. The total die area is $318.5mm^2$.

the area implements hardware components directly controlled by software: main memory, register arrays, execution datapaths. By issuing proper arithmetic and memory instructions, software can control their use and even turn them off to save energy, when their functionality is not necessary. In contrast, superscalar processors devote more than half of their die area for resources transparent to software such as caches and speculation logic [PJS96].

5.3.3 Design Methodology

Table 5.3 presents the design methodology statistics for VIRAM-1. To reduce development time and maximize the functionality supported, VIRAM-1 includes intellectual property from four sources. The use of simple and standard interfaces, such as the MIPS coprocessor interface, was a key factor to integrating them successfully. We also tried to avoid full custom circuit development as much as possible. We synthesized most of the logic from RTL description and used macro compilers for the SRAM arrays in the MIPS caches and the scalar register files. Only the vector register file banks and the memory crossbar were implemented using full-custom techniques, since their function could not be synthesized efficiently. The full custom logic accounts for 7.4% of the total die area (see Table 5.2).

We verified the functionality of the original RTL description and the synthesized circuits using a combination of directed and random tests. Directed tests allowed us to quickly correct most design errors, but random tests were necessary to remove the last few bugs and increase our overall confidence. The testing methodology and the infrastructure that allowed us to reuse test code on a number of simulators and design blocks are described in [Wil02]. Compared to superscalar processors, the verification of VIRAM-1 was much simpler due to its overall design modularity and its lack of complex logic structures for out-of-order or speculative execution.

In retrospect, it was the modular and simple microarchitecture of VIRAM-1 that made its

Component	Area (mm^2)	% Area	Design
Scalar Core	12.25	3.8%	Synthesized
Vector Control	11.40	3.6%	Synthesized
Vector Lane	(4x) 9.90	12.4%	
Arithmetic Unit 0	4.06		Synthesized
Arithmetic Unit 1	1.08		Synthesized
Memory Unit	1.18		Synthesized
Flag Unit & Registers	1.06		Synthesized
Vector Register File	2.31		Full Custom
Miscellaneous	0.21		Synthesized
Crossbar	(2x) 7.20	4.5%	Full Custom
FPU	2.10	0.7%	Synthesized
IO	1.40	0.4%	Synthesized
DRAM Bank	(8x) 16.90	42.4%	Macro block
Pad Ring	35.70	11.2%	
PLL, Decoupling Capacitors, Guard Ring	66.5	21.0%	
Total Area	318.5	100%	

Table 5.2: The area breakdown for VIRAM-1. The scalar core area includes the first level instruction and data caches. The vector control area includes the vector TLB and the replica tags for invalidations in the scalar core data cache. The second column represents the area occupied by each component as a percentage (%) of the total die area. The last column presents the design technique used for each logic or memory block: synthesized (standard cells), full custom, or automatically generated macro blocks.

development possible with the available man-power. It allowed us to quickly scale the design when the area budget changed. We could also easily experiment with a number of alternatives when we encountered a CAD tool bug or limitation. The small design team made it easier to communicate changes or issues and adapt to them fast.

Naturally, certain design decisions were not necessarily optimal. Eliminating all need for full custom circuits, for example, could have saved us a lot of time and the need to struggle with a mixed synthesized-custom design flow. We could assemble the register file partition in each vector lane from four 3-ported SRAM macro-blocks from IBM for approximately the same area. However, such a register file design would prohibit the implementation of the fixed-point multiply-add instruction. It would also have a negative impact on overall performance due to the higher probability of bank conflicts between vector instructions during their register file accesses for source and destination operands. We could also replace the full-custom crossbar with a simpler circular ring topology that is easier to implement using synthesized logic. Since the actual layout for both approaches is limited mostly by wire density, the area requirements should be nearly identical. However, the ring interconnect would add 4 clock cycles to the latency of every memory access. Due to the nature of the delayed pipeline, the higher memory latency could lead to significant performance loss for applications with reductions (`Autocor` and `Bital`), short vectors (`Cjpeg`, `Djpeg`, and `Viterbi`), or low ratio of arithmetic to memory operations (`Rgb2cmyk` and `Autocor`).

5.4 Performance Evaluation

In this section, we analyze the performance of VIRAM-1 for the multimedia benchmarks in the EEMBC suite. We also compare VIRAM-1 to a number of embedded processors that implement

Design Methodology	Synthesized: scalar core, vector control, vector datapaths, IO Full-custom: vector register file, memory crossbar Macro-blocks: DRAM banks, SRAMs
IP Sources	UC Berkeley (vector coprocessor, crossbar, IO) MIPS Technologies (MIPS Core) IBM (DRAM, SRAMs) MIT (original floating-point datapath design)
RTL Model	170K lines Verilog
Verification	566K lines directed tests (10M lines assembly) 4 months of random testing on 20 Linux workstations
Design Team	3 full-time graduate students 3 part-time graduate students and staff
Design Time	2.5 years

Table 5.3: The design methodology statistics for VIRAM-1.

alternative architectures such as CISC, RISC, VLIW, and DSP.

We measured the performance of VIRAM-1 using an execution-based simulator that provides near cycle-accurate modeling of the chip microarchitecture with a variable number of lanes. The simulator models all sources of stalls excluding DRAM refresh. However, refresh accesses to DRAM banks are periodic events with low frequency. In addition, the memory unit in the vector coprocessor has direct control of refresh and attempts to schedule refresh accesses to each bank during idle cycles. Hence, the practical impact of DRAM refresh on performance is negligible.

5.4.1 Performance of Compiled Code

Before we proceed with the performance evaluation, it is interesting to examine the quality of the code produced by the VIRAM compiler in terms of performance and compare it with code optimized in assembly. By taking a close look at the output of our research compiler, we can notice the following deficiencies:

1. Our compiler cannot always schedule the code within a basic block so that dependent instructions are separated by at least one independent instruction. This increases the probability that a register dependency will cause stall cycles in the in-order (statically scheduled) pipeline of VIRAM-1.

2. The code schedule does not attempt to alternate instructions to the four functional units in VIRAM-1. This leads to unbalanced workload for the functional units. Some of the functional units may be idling for long time periods despite the existence of independent instruction later in the program that they could execute.

3. Our compiler cannot move the code for calculating loop invariants out of the loop body (code motion). This causes unnecessary repetition of calculations.

4. Our compiler frequently generates spill code for kernels that use more than 8 vector registers, even if less than 32 registers are necessary. Spill code causes unnecessary memory traffic and increases the workload of the memory unit in VIRAM-1.

5. The MIPS scalar code quality is low both in terms of density and scheduling.

It is interesting to notice that none of the compiler shortcomings are related to vectorization, the most challenging part of generating code for a vector processor.

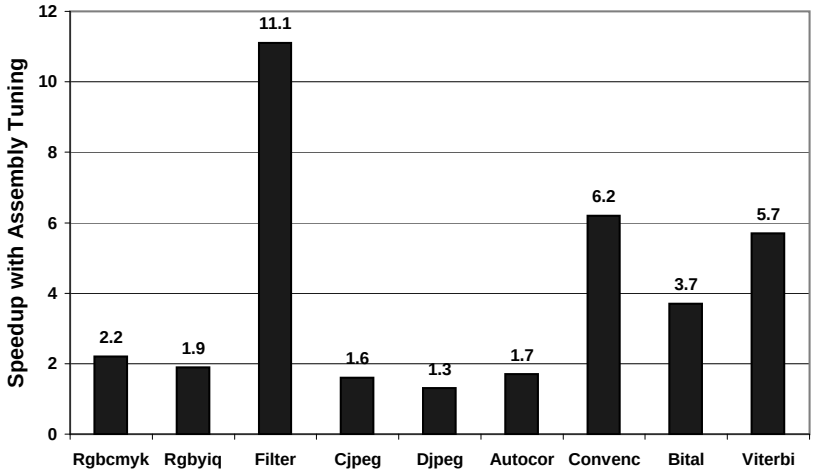


Figure 5.5: The performance speedup for VIRAM-1 after manually tuning the basic kernels of each benchmark in assembly. In each case, we present the ratio of performance achieved with the optimized code over that achieve with the original compiled code. A ratio larger than 1 implies higher performance with assembly code. We do not present the speedup for `Fft` because we can only vectorize this benchmark in assembly. For a comparison of code sizes for the two approaches, refer to Tables 4.6 (page 33) and 4.7 (page 34).

To evaluate the performance cost of the compiler shortcomings, we used assembly programming to remove the first four deficiencies from the basic kernels in each benchmark. Figure 5.5 presents the performance improvement for VIRAM-1 when running code tuned in assembly instead of the original compiler code. The speedup varies between 1.6 and 11, depending on the benchmark. The effect of assembly tuning is most obvious for applications with mostly unit stride accesses, such as `Filter` and `Convenc`. For benchmarks with many strided accesses such as `Rgb2cmyk` and `Rgb2yiq`, performance is limited by the address generation bandwidth of the memory unit in VIRAM-1, hence the effect of assembly tuning is less profound. The same holds for benchmarks with large percentage of scalar operations such as `Cjpeg` and `Djpeg`, where scalar optimizations are also necessary in order to achieve higher speedup with tuned code.

During assembly tuning, we applied code scheduling only within each basic block. We did not apply loop unrolling or software pipelining. These techniques require major code restructuring and would not introduce more than 10% performance improvement. As we demonstrate in the following sub-sections, VIRAM-1 can reach high levels of performance even without such optimizations. In addition, loop unrolling and software pipelining lead to significant increases in static code size (see Section 4.5).

Overall, the quality of the code produced by the VIRAM-1 compiler is satisfactory for a research project. The compiler can handle vectorization which is the most critical optimization for this architecture. However, the optimizations we applied during assembly tuning are straight-forward and are typically available in most industrial compilers. Hence, the VIRAM-1 results with code tuned in assembly are good indications of the performance potential for commercial implementations of the VIRAM-1 microarchitecture and its compiler.

5.4.2 Performance Comparison

To better understand the performance advantages of VIRAM-1, we compare it to six embedded processors that implement alternative architectures. Table 5.4 presents their basic charac-

Processor	Vendor	Architecture	Clock Freq.	Issue Width	Cache Size L1I/L1D/L2	Power
K6-III+	AMD	CISC (x86)	550MHz	2 (6)	32K/ 32K/256K	21.6W
MPC7455	Motorola	RISC (PowerPC)	1000MHz	4	32K/ 32K/256K	21.3W
VR5000	NEC	RISC (MIPS)	250MHz	2	32K/ 32K/—	12.1W
TM1300	Trimedia	VLIW+SIMD	166MHz	5	32K/ 16K/—	2.7W
C6203	TI	VLIW+DSP	300MHz	8	96K/512K/—	1.7W
21065L	ADI	DSP	60MHz	1	0.2K/—/ 68K	1.5W

Table 5.4: The characteristics of the six embedded processors used for performance comparisons with VIRAM-1. K6-III+ and MPC7455 use out-of-order execution. K6-III+ can decode 2 CISC instruction per cycle to sequences of simpler operations. It can execute up to 6 simpler operations per cycle. All cache sizes are in KBytes and refer to on-chip SRAMs. 21065L includes a 68KByte SRAM that serves as both second-level instruction cache and primary data cache. The power consumption numbers refer to typical usage and are derived from vendor data-sheets.

teristics. The selected group includes a variety of design points not only in terms of the underlying architecture (CISC, RISC, VLIW, and DSP), but also in terms of clock frequency, issue width, power consumption, and execution style (in-order, out-of-order). However, all processors rely on SRAM caches for fast memory accesses.

We retrieved the performance results for the six processors from the corresponding EEMBC reports [Lev00]. Results for the consumer benchmarks are not available for C6203 and 21065L. Results for the telecommunications benchmarks are not available for TM1300. For the two VLIW processors (TM1300 and C6203), we report separately the performance level achieved with direct compilation (*cc*) and after extensive optimization of the C code (*opt*). Similarly, for VIRAM-1 we report separately performance with compiled code (*cc*) and with code optimized in assembly (*as*).

Figure 5.6 presents the *ConsumerMark* and *TeleMark* composite scores for VIRAM-1 and the six embedded processors. Running compiled code, VIRAM-1 outperforms all processors excluding MPC7455, a 1GHz, 4-way superscalar, out-of-order design, and the two VLIW processors (TM1300 and C2603) when running optimized code. The use of code optimized in assembly allows VIRAM-1 to achieve scores at least 70% and 40% higher than any other processor for the consumer and telecommunications categories respectively.

Figures 5.7 and 5.8 present the individual benchmark scores. The performance advantage of VIRAM-1 is more noticeable for benchmarks with long vectors, such as `Rgb2cmyk`, `Rgb2yiq`, `Filter`, `Autocor`, `Convenc`, and `Fft`. Each vector instruction in these benchmarks can keep the parallel datapaths across the four lanes busy for multiple clock cycles, which leads to high performance. For `Rgb2cmyk` and `Rgb2yiq`, VIRAM-1 uses the four address generators in the memory unit to load or store four element per cycles for strided accesses. Despite the lack of SRAM caches for vector memory references, the high memory bandwidth of embedded DRAM and the delayed vector pipeline lead to high performance for memory accesses.

For `Bital`, the performance of VIRAM is limited by the reduction in each iteration of the outer-loop, which detects the termination condition for the benchmark. The code for reductions fails to utilize all vector lanes during its last stages. It also exposes the length of the delayed pipeline, because its result must be sent to the scalar core, where the termination condition is checked. Still, the performance of VIRAM-1 for `Bital` with compiled code is equal to that of K6-II+. With optimized code for `Bital`, VIRAM-1 can match the performance of MPC7455.

Despite the lack of long vectors, the performance of VIRAM-1 for `Viterbi` is comparable to the highest scores achieved by other processors. Instruction with short vectors can keep the four lanes busy for only one or two clock cycles. With instruction issue bandwidth of one instruction

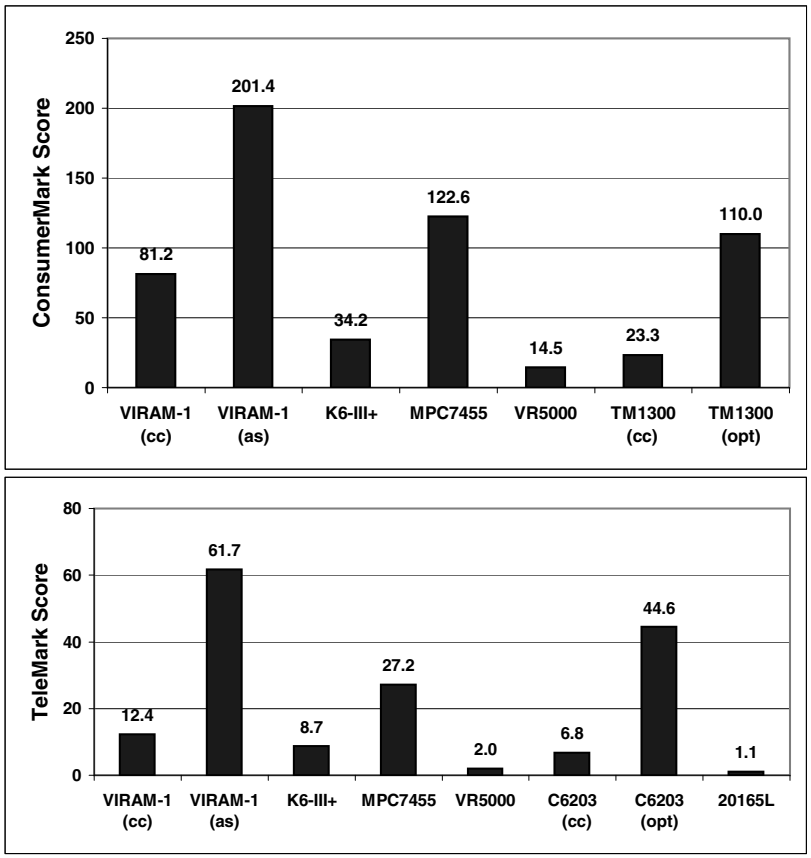


Figure 5.6: Comparison of the composite scores for the consumer and telecommunications categories in the EEMBC suite. Each composite score is proportional to the geometric mean of the performance (iterations per second) achieved by each processor for the benchmarks in the corresponding category. A higher score indicates higher performance.

per cycle, short vectors translate to low utilization of the functional units in the vector coprocessor. The other two benchmarks with short vectors are `Cjpeg` and `Djpeg`, for which the scalar code overhead dominates all other performance bottlenecks. The scalar code for Huffman coding in both benchmarks accounts for approximately 70% of execution time. The high overhead for scalar operations is because of the lower degree of vectorization for `Cjpeg` and `Djpeg` (65%) and the poor quality of scalar code produced by the VIRAM-1 compiler.

Figures 5.9 and 5.10 report the same benchmark scores normalized by the clock frequency of each processor. Normalization allows us isolate the contributions of architectural and microarchitectural features to performance from the benefits of advanced circuit design. It is worthwhile to notice the normalized performance of MPC7455 is significantly lower than that of VIRAM-1 running compiled code. Despite its four-way superscalar issue and out-of-order capabilities, MPC7455 relies mostly on high frequency operation (1GHz). However, high clock frequency also leads to high power consumption (21W). VIRAM-1 can achieve higher performance at low power consumption through parallel execution of element operations. In addition, VIRAM-1 is in-order and single-issue. Hence, assuming equal design efforts, it should be possible to achieve higher clock frequencies for VIRAM-1 than MPC7455.

The main architectural competition for VIRAM-1 is from the two VLIW designs, TM1300

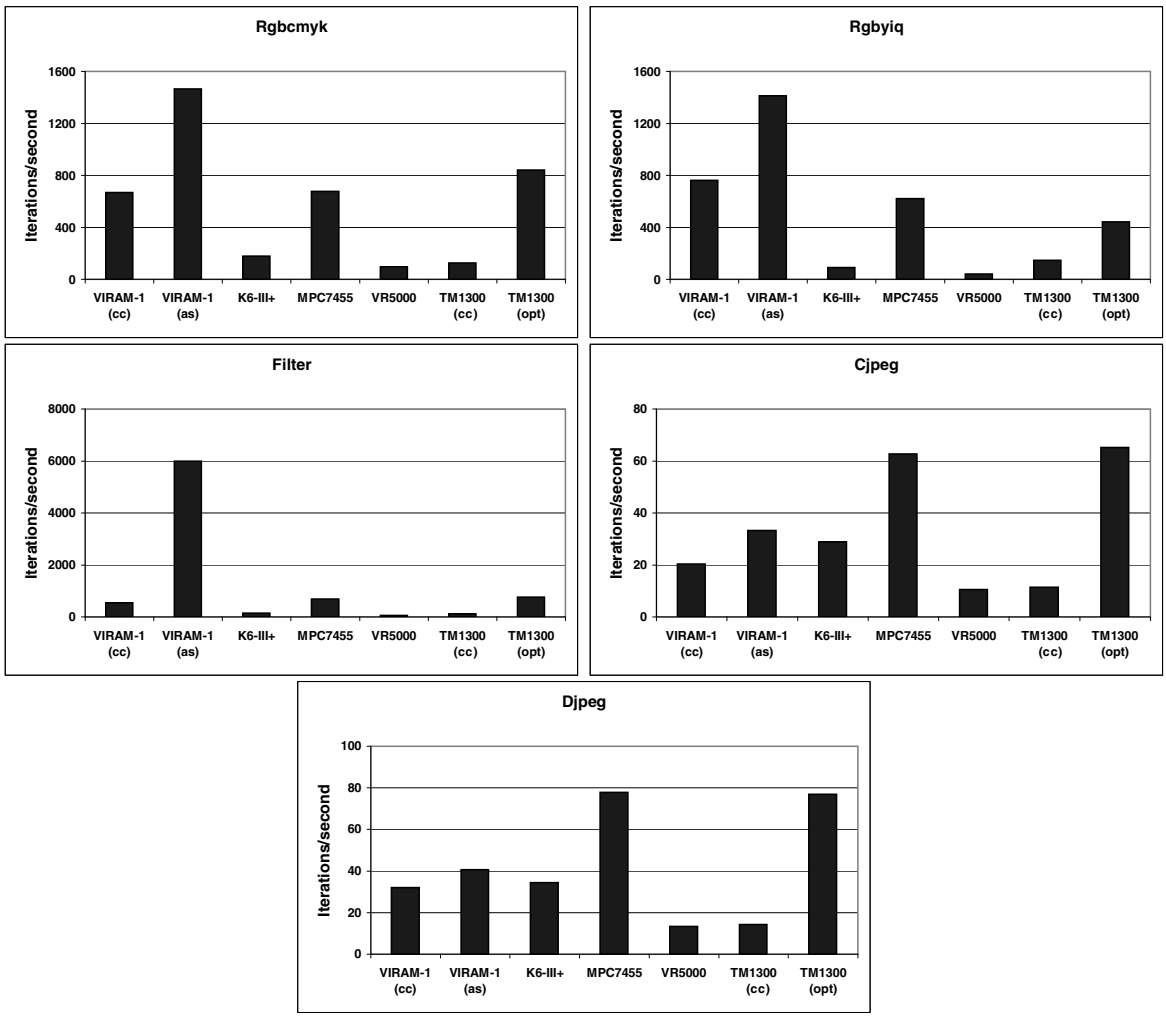


Figure 5.7: Detailed performance comparison between VIRAM-1 and four embedded processors for the consumer benchmarks. Scores are in iterations per second and a higher score indicates higher performance.

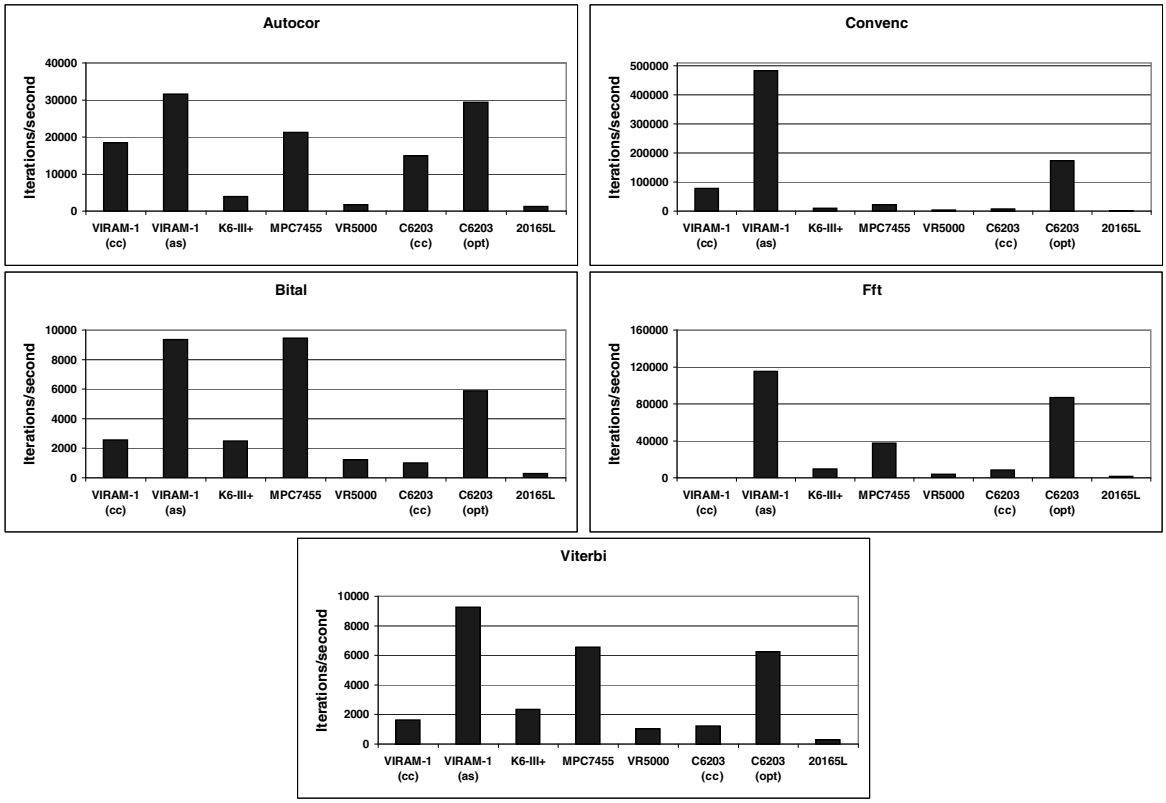


Figure 5.8: Detailed performance comparison between VIRAM-1 and five embedded processors for the telecommunications benchmarks. Scores are in iterations per second and a higher score indicates higher performance. The VIRAM-1 performance for `Fft` is only available for assembly code.

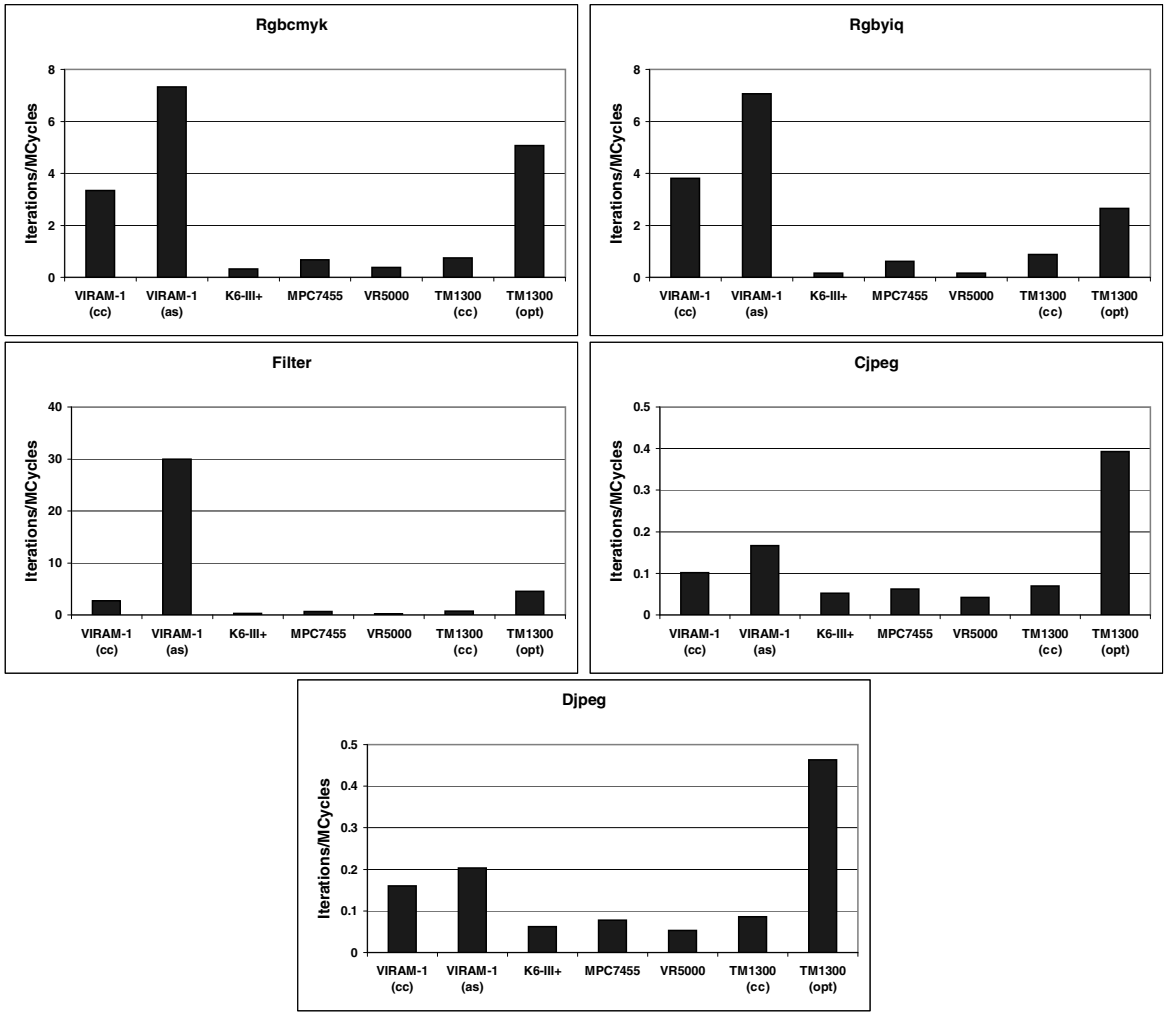


Figure 5.9: Normalized performance comparison between VIRAM-1 and four embedded processors for the consumer benchmarks. Scores are in iterations per million cycles and a higher score indicates higher performance. There scores were derived by normalizing the scores in Figure 5.7 by the clock frequency of each processor.

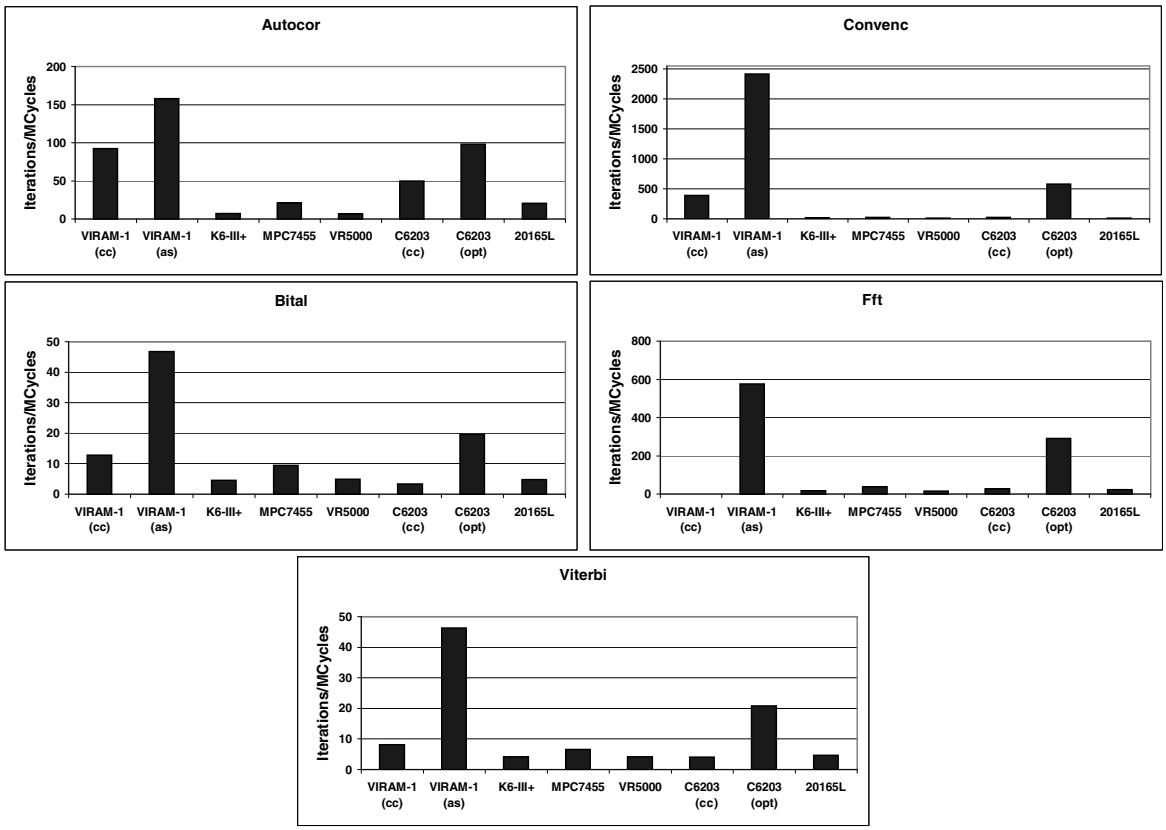


Figure 5.10: Normalized performance comparison between VIRAM-1 and five embedded processors for the telecommunications benchmarks. Scores are in iterations per million cycles and a higher score indicates higher performance. There scores were derived by normalizing the scores in Figure 5.8 by the clock frequency of each processor. The VIRAM-1 performance for `Fft` is only available for assembly code.

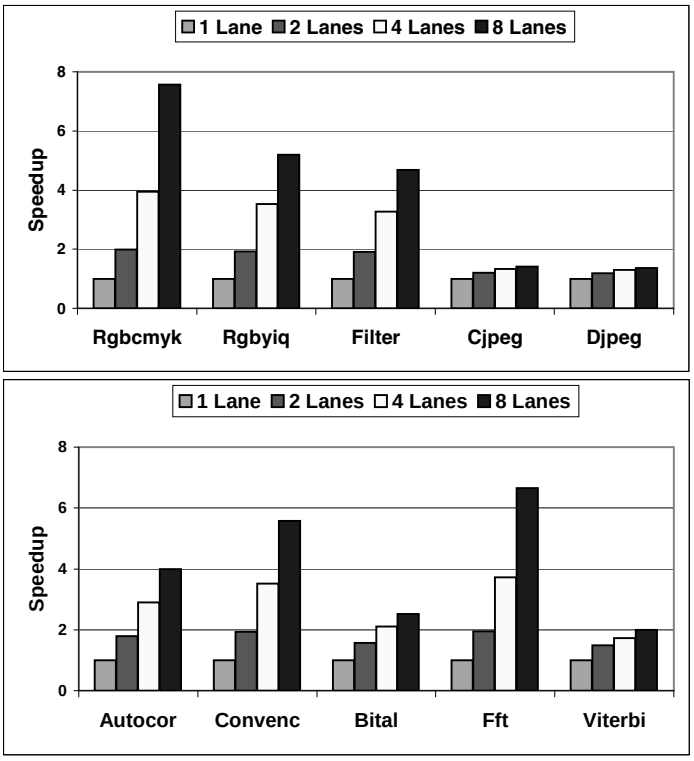


Figure 5.11: The speedup of multi-lane implementations of the VIRAM-1 microarchitecture over a processor with a single lane. For each configuration we set the number of address generators in the memory unit equal to the number of lanes.

and C6203. VIRAM-1 outperforms both processors by at least a factor of 2 when they all run compiled code. The same holds with optimized code for all three processors. Of course, restructuring of C code is easier than tuning in assembly. However, the optimization capabilities missing from the VIRAM-1 compiler are straight-forward and have been available for years in commercial compilers. On the other hand, efficient compilation for VLIW processors is a relatively new area for industrial compilers. Hence, it is reasonable to expect that a vector compiler will be able to match the performance of assembly code on VIRAM-1 long before a VLIW compiler can eliminate the need for restructuring of C code in order to achieve high performance with VLIW architectures.

5.4.3 Performance Scaling

One of the most interesting features of the VIRAM-1 microarchitecture is the ability to easily scale its performance, area, and energy consumption by allocating the proper number of vector lanes. Figure 5.11 presents the effect of the number of lanes on the performance for each consumer and telecommunications benchmark. To simplify the charts, we present performance as speedup over a VIRAM-1 configuration with a single lane. Ideally, the speedup with 2, 4, and 8 lanes is 2, 4, and 8 respectively.

Only two benchmarks, `Rgb2cmyk`and `Fft`, come close to the ideal speedup in all cases. The performance for other benchmarks with long vectors scales well for up to four lanes (`Rgb2yiq`, `Filter`, and `Convenc`). With 8 lanes, however, the number of cycles it takes to execute a vector instruction at maximum vector length is only 4 cycles. Hence, the overhead introduced by scalar

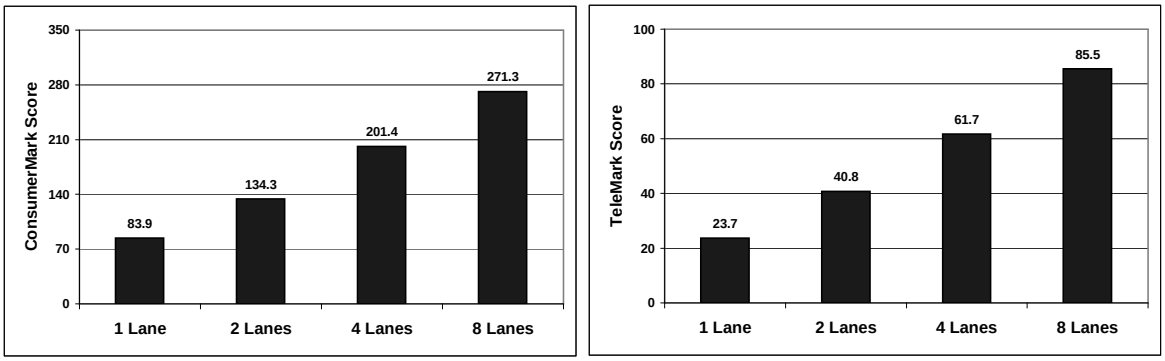


Figure 5.12: The composite scores for the consumer and telecommunications benchmarks for VIRAM-1 as a function of the number of lanes. The results represent performance achieved using VIRAM code tuned with assembly programming.

code, inefficiencies in scheduling, and stalls due to RAW hazards becomes much more significant than with four or less lanes. With benchmarks that include dot-products such as `Autocor` and `Bital`, the benefit of additional lanes is limited by the fact that the last few stages of a reduction cannot use datapaths across all lanes. Finally, the number of lanes has limited effect on benchmarks with very short vectors or high scalar overhead, such as `Cjpeg`, `Djpeg`, and `Viterbi`.

Figure 5.12 presents the composite scores of VIRAM-1 as a function of the number of lanes. For both benchmark categories, the use of a second lane leads to 75% performance improvement. The two subsequent doublings of the numbers of lane lead to 55% and 40% improvement respectively. It is also interesting to compare the scores in Figure 5.12 with the results in Figure 5.6 (page 52). Even with one or two lanes, VIRAM-1 running at 200 MHz can achieve performance scores comparable or better than complex out-of-order and VLIW designs running at higher clock frequencies.

5.5 Lessons from VIRAM-1

VIRAM-1 demonstrates the feasibility of using vector processing with embedded DRAM for multimedia applications in embedded systems. The microarchitecture is simple but possesses a set of key properties that lead to performance and energy efficiency:

- **Vector processing provides flexibility in architecting efficient processors.** We achieve high performance by relying on concurrent execution of element operations on parallel datapaths. The fact that high clock frequency is not necessary simplifies design and reduces energy consumption. Furthermore, we can trade off area for reduced power consumption. Vector processing provides this flexibility without complex control structures for speculation or out-of-order execution.

- **Vector lanes are an efficient method for exploiting parallelism in the form of long vectors.** Even though a single lane has simple structure and is easy to design, multiple operations can execute concurrently across parallel lanes. No additional instruction issue bandwidth and virtually no change in control logic are necessary for more lanes, assuming instructions specify a large number of element operations.

- **High memory bandwidth can lead to high performance.** VIRAM-1 uses the high bandwidth available from embedded DRAM to maintain a high throughput of element operations on its parallel datapaths. Both sequential and random access bandwidth are necessary to meet the requirements of the three addressing modes for vector loads and stores.

- **Both the processor and the memory system are modular and use a small number of replicated hardware blocks.** We can generate multiple processor implementations for a variety of performance, area, and energy requirements by mixing and matching the proper number of vector lanes and memory banks. Instead of a complete chip redesign, we only need to focus on the top-level interconnect in each case.

Nevertheless, prototyping VIRAM-1 also revealed certain inefficiencies that we did not initially anticipate. Most of the issues do not affect the effectiveness of the VIRAM-1 chip, but limit the potential of its microarchitecture with future processor implementations:

- **Vector lanes are not efficient with short vectors.** To keep the instruction issue requirements low, a vector instruction should keep a functional unit busy for more than four cycles. Vector instructions with only 8 to 16 element operations, which are frequent in image and video processing, cannot make effective use of a processor with four or more lanes.

- **The complexity of the vector register file is a bottleneck for exploiting non-vectorizable parallelism.** For applications with short vectors, we can organize additional hardware resources in the form of more functional units within each lane. Using chaining, each functional unit can execute another vector instruction in parallel, even in the presence of dependencies. However, each functional unit requires a set of read and write ports in the vector register file. The area, energy consumption, and latency of the register file are exponential functions of the number of access ports. Therefore, the vector register file becomes the major bottleneck to executing more vector instructions in parallel within each vector lane.

- **The delayed pipeline is not efficient with memory latency larger than that for VIRAM-1.** The low clock frequency of VIRAM-1 and the moderate latency of the embedded DRAM macro it uses, allows the delay pipeline to hide memory latency. However, a higher latency would translate to a longer pipeline with a larger area and power cost, and higher overhead for filling and draining between vector loops or functions. Hence, if we use the same microarchitecture with a slower memory system or with a higher clock frequency target for the processor, a significant performance drop will occur. In addition, introducing the additional pipeline stages requires non-trivial redesign.

- **Imprecise virtual memory exceptions complicate software development.** Restartable, imprecise exceptions with saving and restoring of microarchitecture state are sufficient for correct operation of virtual memory, but are cumbersome to use and often discourage software and operating system development. Supporting precise memory exceptions with the VIRAM-1 microarchitecture would require stalling all other functional units when a memory instruction executes and would reduce performance to unacceptable levels.

In the following chapters, we introduce an alternative vector microarchitecture for embedded media-processors. The new design builds on the experience of VIRAM-1 and addresses its shortcomings while maintaining its basic advantages.

Chapter 6

A New Vector Microarchitecture for Multimedia Execution

> "The significant problems we face cannot be solved at the same level of thinking we were at when we created them."
> *Albert Einstein*

Although VIRAM-1 provides a modular organization for a complete vector processor integrated with its memory system, it is inefficient for applications with short vectors and cannot scale to a larger number of functional units. This chapter presents *CODE* (**c**omposite **o**rganization for **d**ecoupled **e**xecution), a new microarchitecture for the VIRAM instruction set that overcomes the limitations of VIRAM-1. It provides two orthogonal methods for organizing vector hardware in order to exploit data-level parallelism in both long and short vectors. The CODE microarchitecture establishes the basis for a family of vector microprocessors tailored to the performance, power, and cost requirements of specific application domains.

Section 6.1 introduces the two basic elements of CODE and discusses their benefits. Section 6.2 provides a detailed description of the microarchitecture components for execution, memory access, data communication, and instruction issue. Section 6.3 presents a number of issue and control policies along with a set of instruction execution examples. Section 6.4 discusses the benefits from multi-lane implementations of the CODE microarchitecture. Sections 6.5 and 6.6 compare CODE to the VIRAM-1 microarchitecture and organizations for out-of-order execution respectively. Section 6.7 describes an experimental study for tuning the basic microarchitecture policies. Finally, section 6.8 reviews related work in academic and commercial microarchitectures.

6.1 Basic Design Approach

The CODE microarchitecture takes its name from the two basic ideas it combines: composite organization and decoupled execution. The two techniques replace the centralized register file and the delayed pipeline of VIRAM-1.

6.1.1 Composite Vector Organization

VIRAM-1 structures the vector coprocessor around a centralized vector register file that provides operands to all functional units and connects them to each other. In contrast, the composite approach organizes vector hardware as a collection of interconnected cores. Each core is a simple vector processor with a set of local vector registers and one functional unit. Each core can execute

only a subset of the instruction set. For example, one vector core may be able to execute all integer arithmetic instructions, while another core handles vector load and store operations.

The composite organization breaks up the centralized vector register file into a distributed structure. It uses renaming logic to map the vector registers defined in the VIRAM architecture to the distributed register file. The local vector register file within each core must only support one functional unit and has a fixed and small number of access ports. Therefore, the local vector register files are simple to implement without having to resort to complicated full-custom design techniques. Furthermore, associating a local register file with each core establishes a balance between the number of functional units and the number of vector registers. Registers provide short-term storage for the operands of functional units, hence the total number of registers in a processor must be roughly proportional to the number of functional units it includes [RDKM00]. In Section 6.6, we analyze the area, access latency, and energy consumption advantages of the distributed register file structure.

Another benefit of the composite organization is the ability to scale the vector coprocessor in a flexible manner by mixing the proper number and type of vector cores. If the typical workload for a specific implementation includes a large number of integer operations, we can allocate more vector cores for integer instruction execution. Similarly, if floating-point operations are not necessary, we can remove all cores for floating-point instructions. In contrast, with VIRAM-1 we can only increase performance by allocating extra lanes, which scales evenly integer and floating-point capabilities, regardless of the specific needs of applications.

The composite organization requires a separate network for communication and data exchange between vector cores, as there is no centralized register file to provide the functionality of an all-to-all crossbar for vector operands. If a full crossbar switch is necessary to accommodate the data transfers between cores, the area and energy benefits from eliminating the centralized register file will vanish. However, the inter-core communication requirements are far smaller so that a simpler network structure is sufficient. Certain types of cores rarely need to communicate, such as the cores for integer and the cores for floating-point instructions. When communication is necessary it is usually for a single operand, hence the network need not provide bandwidth for all instruction operands, as it is the case with the centralized register file. Finally, if multiple cores can execute an instruction, we can take locality of operands into account before assigning an instruction to a specific core, in order to minimize the need for inter-core communication.

6.1.2 Decoupled Execution

Decoupled execution [Smi84] plays a dual role in the CODE microarchitecture. First, it replaces the delayed pipeline as a more flexible technique for tolerating long memory latencies for vector accesses [EV96]. Second, it provides a mechanism for utilizing multiple vector cores with a single instruction stream.

The basic idea in decoupling is to separate an instruction sequence into multiple streams, one for each vector core. For example, memory instructions are placed in one stream to be issued to a load-store core, while integer instructions create a separate stream for an integer execution core. By providing an instruction queue in each core, the streams can slip with respect to each other. A load-store core may run further ahead and initiate memory transfers before other cores complete older instructions and long before following arithmetic operations need the fetched data. This load-store core is practically prefetching data and helps tolerate long memory latencies. To handle register dependencies between streams, decoupling requires a pair of queues for data transfers for every two cores [SWP86]. For example, to use the result of a load instruction with a vector add, the integer core reads data from a queue where the load core has placed them. An empty or full data queue causes the receiving or transmitting core respectively to stall. Similarly, a full instruction queue causes the issue logic to stall.

Decoupled execution allows the composite microarchitecture to execute multiple instructions in parallel on the vector cores. Long latency operations, such as memory accesses or divides,

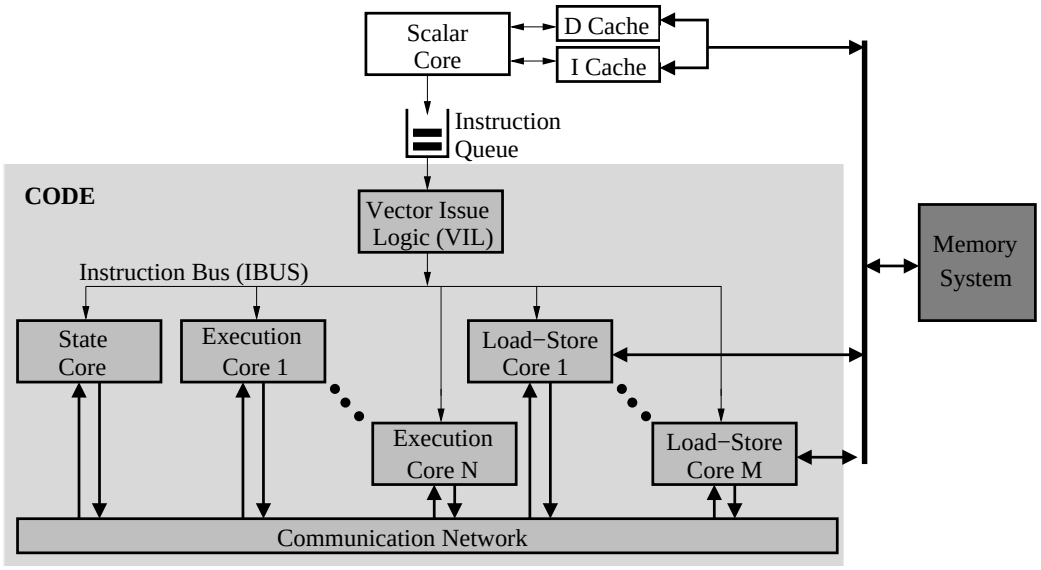


Figure 6.1: The block diagram of the CODE microarchitecture. The vector coprocessor consists of one state core, N execution cores, and M load-store cores, interconnected by a communication network. The vector issue logic (*VIL*) block issues instructions to the cores and communicates with the scalar core when necessary.

do not have to stall the whole system. Other cores can proceed with independent instructions from different streams. In addition, a core that runs ahead may be able to initiate any long latency operations early enough so that dependent instructions do not experience any stalls due to register hazards.

The disadvantage of decoupling is the area and power overhead introduced by data queues. A data queue in a decoupled vector processor must provide space for several vector registers, each with tens of elements [Esp97]. In addition, a system with multiple cores requires a large number of data queues. To avoid the overhead and the complexity of implementing two queues per couple of cores, the CODE microarchitecture uses the local vector registers in each core as a unified storage space for data communication. Each core writes the data it receives from any other core directly in a local vector register from where it can access them later. Local vector registers are already necessary for instruction operands and provide a large amount of storage capacity. Using them for the data queues as well leads to a more efficient use of storage space. We cannot have the situation where one data queue is empty while another one in the same core is full and causes stalls.

6.2 Microarchitecture Description

Figure 6.1 presents the block diagram for the CODE microarchitecture. The vector coprocessor consists of a collection of interconnected cores and a central issue logic block that dispatches vector instructions to them. The following subsections provide a detailed description of the structure and operation of each design component. An instruction queue decouples the scalar core and the vector coprocessor. The queue allows the scalar core to determine most control dependencies (conditional branches) ahead of time, without interrupting instruction execution in the vector coprocessor.

6.2.1 Vector Core Organization

Figure 6.2 presents the three classes of cores in the CODE microarchitecture and their internal structure. Execution cores (Figure 6.2.a) implement non-memory vector instructions. Memory operations execute in load-store cores (Figure 6.2.b). A processor may include multiple execution and load-store cores, some of which may be identical. A state core (Figure 6.2.c) provides additional storage for vector registers. There can be only one state core in the system.

Each execution core includes an instruction queue (*IQ*) for decoupling purposes, a local vector register file (*LVRF*), and one functional unit (*FU*). One input (*INIF*) and one output interface (*OUTIF*) connect to the communication network and allow data exchange with other cores. The LVRF must have sufficient read and write ports to support the functional unit(s), plus one read and one write port for the communication interfaces. Depending on the type of instructions supported by the functional unit, the number access ports for the LVRF may range from 3 read and 2 write ports to 4 read and 2 write ports. The number of LVRF ports is independent of the total number of cores in the system.

The capabilities of the functional unit, in other words the instructions they can execute, define the type of the execution core. We can simply implement one type per vector instruction group listed in Table 4.2 (page 28), with one core type for all floating-point instructions, one for vector processing operations, and so on. A single type can support both integer and fixed-point operations because they execute on similar datapaths. For integer and floating-point instructions, we can implement separate functional units for simple (add, subtract) and complex (multiply, divide) operations. Table 6.1 presents the common types of execution cores used in this thesis.

A load-store core is similar to an execution one, but, instead of a functional unit, it includes a load-store unit. Depending on the capabilities of the unit, the core may be able to execute all memory instructions or just a subset (loads or stores, unit stride or indexed and strided accesses). A load-store unit includes address generation hardware (*AG*) and an output address queue (*LSAQ*) for pending memory accesses.

The state core includes no functional units. Its only purpose is to introduce additional vector registers. There must be storage space for at least 32 vector registers in the whole processor, which is the number of registers defined in the VIRAM instruction set. Furthermore, an execution or load-store core may run out of local vector registers if it executes a large number of instructions that use disjoint sets of architectural registers. In this case, we must transfer a few vector registers to other cores in order to make space for the operands of new instructions. Extra registers in the state core can be particularly useful for this purpose.

6.2.2 Communication Network

The communication network provides the necessary interconnect for exchanging vector registers between cores. It handles transmit and receive requests from the input and output interfaces of the cores. Each matched pair of requests initiates a vector register transfer, assuming that sufficient network bandwidth is available at the time. In most practical implementations, the network will be able to transmit only a few vector elements in parallel and a register transfer will last several clock cycles. However, the transfer can take place ahead of time from the instruction that requires the vector register. In addition, the instruction execution and the register transfer can overlap in a chained fashion.

The communication network is free of deadlocks, regardless of the number of vector cores it connects, if all cores process the instructions and register transfers assigned to them in strict issue order. We can prove this network property by noticing that the two cores involved in the oldest pending transfer cannot make transmit or receive requests for any other transfers without violating issue order. Therefore, we can establish a transmit-receive request match for the oldest transfer, which allows the network to complete it. By induction, all following transfers will also complete and

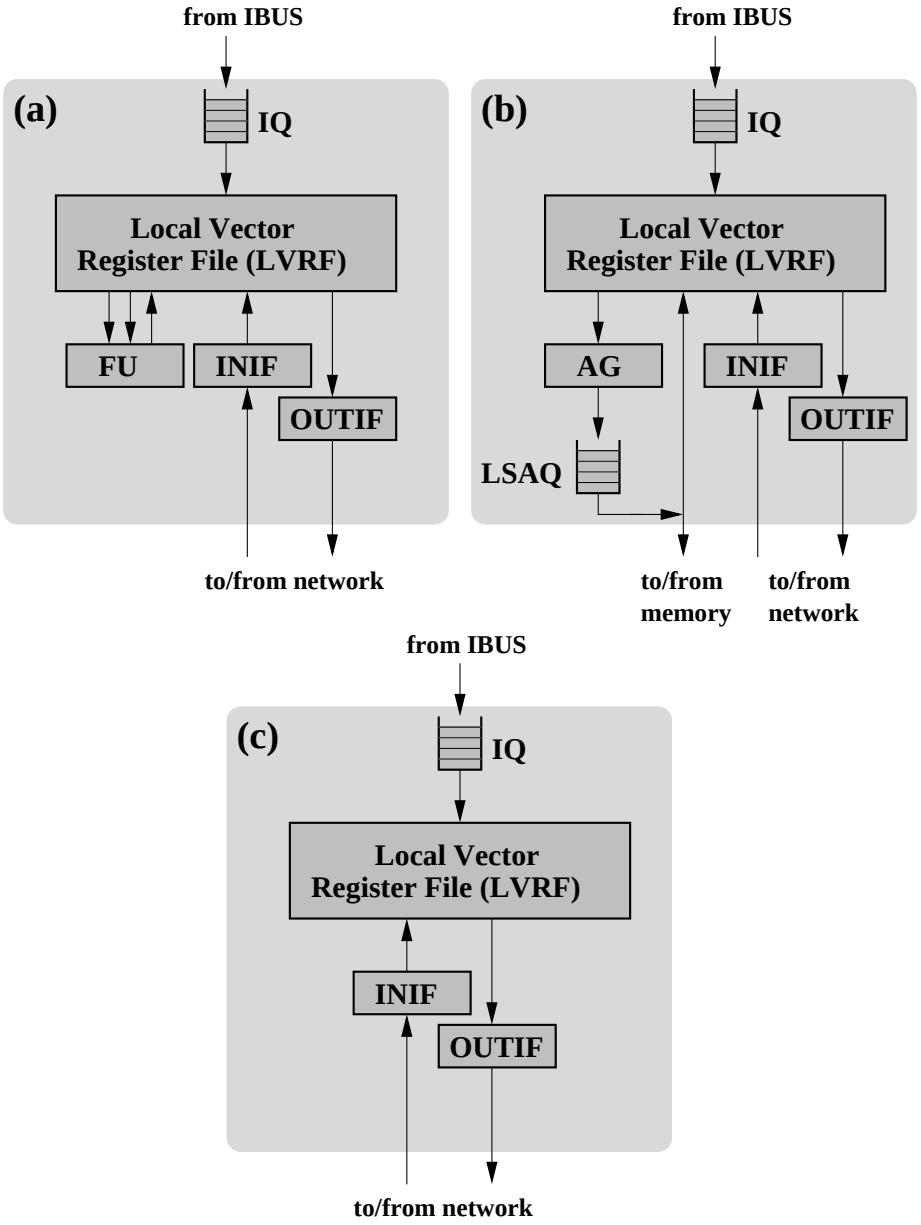


Figure 6.2: The internal organization of the three classes of vector cores: execution (a), load-store (b), and state core (c). All classes include an instruction queue (IQ), a local vector register file ($LVRF$), one input interface($INIF$) and one output interface ($OUTIF$). An execution core contains one functional unit (FU). A load-store core includes address generation hardware (AG) and an address queue for pending memory accesses ($LSAQ$). The input and output interfaces in each core connect to the communication network. They include no data storage other than a single cycle buffer for retiming.

Core Type	Core Class	Functional Unit Capabilities	VLRF Ports (read/write)
IntSimple	Execution	Simple integer operations	3/2
IntComplex	Execution	Complex integer operations	3/2
IntFull	Execution	All integer operations	4/2
ArithRest	Execution	Vector processing operations Permutations operations Flag processing operations	3/2
LDLoad	Load-Store	All load operations	2/2
LDStore	Load-Store	All store operations	3/1
LDFull	Load-Store	All load and store operations	3/2

Table 6.1: The common types of execution and load-store cores. The last column lists the number of LVRF access ports (read/write) required for each core configuration. It includes the access ports for the input and output interfaces. The *IntFull* type is the only one that can execute fixed-point multiply-add instructions. No floating-point cores are listed here because the EEMBC benchmarks include no floating-point operations. In practice, cores for floating-point execution are organized similar to integer cores.

deadlock can never occur.

On the other hand, livelocks are theoretically possible. If the number of matched accesses exceeds the available bandwidth, the network may indefinitely delay a transfer for an older instruction in favor of transfers for younger instructions. However, long livelocks cannot happen in practice. Data dependencies to the operands of the delayed instruction and structural hazards to the hardware resources it uses will eventually block the issuing of any new instructions and allow the delayed transfer to complete. We can even eliminate livelocks by using a narrow tag with every instruction that identifies issue order and allows the network to give priority to the oldest transfer with matched requests.

For a specific vector processor, we can choose from a wide variety of alternatives for the network implementation: bus, ring, crossbar, ad-hoc interconnect, and so on. Each alternative leads to different bandwidth, latency, power and energy consumption, and connectivity characteristics. The basic trade-off in selecting a network implementation is between performance, cost of hardware resources, and power consumption. A high bandwidth structure, such as a crossbar, can support multiple concurrent transfers and can minimize the number of stalls due to register dependencies across cores. However, higher bandwidth comes at the cost of increased power consumption for faster transfers and larger area penalty for wiring and switching resources. We will analyze the impact of the network characteristics on the performance of CODE in Chapter 9.

6.2.3 Vector Issue Logic

The vector issue logic (VIL) block issues vector instructions and register transfers to the cores that coordinate the program execution on the composite organization. It does not control the actual instruction execution within the cores or the steps for register transfers. Each core contains decoding and sequencing logic that executes the element operations and cooperates with the communication network for data exchanges.

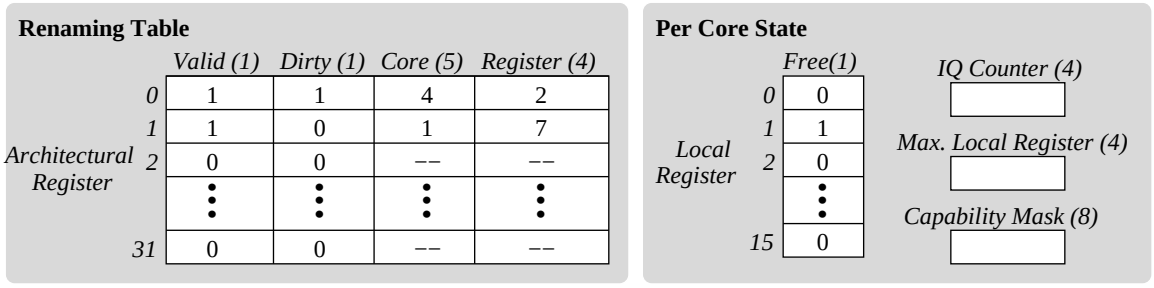


Figure 6.3: The control data structures in VIL. The valid and dirty bits in the renaming table entries specify if the corresponding architectural register is allocated and if it has been recently updated. A large system with up to 32 cores and up to 16 registers per core requires 11 bits per entry. The state for each core includes a free list with one bit per local vector register, a counter for tracking the instruction queue size, and two read-only registers. The read-only registers identify the number of local vector registers and the execution capabilities of the core.

Basic Operation

The VIL block performs two basic tasks. First, it selects the core that will execute each vector instruction. Second, it determines if any of the instruction operands are stored in cores other than the one selected for its execution, and identifies the proper inter-core transfers. To perform the latter task, the VIL needs to maintain the correspondence between each architectural vector register in the VIRAM instruction set and the local vector register in one of the cores that stores its data. The renaming hardware used for this purpose is similar to that in superscalar processors [AST67, Tom67, Gro90, Yea96], but considerably simpler because it processes a single vector instruction per cycle

The instruction bus (*IBUS*) broadcasts instructions from the VIL to the vector cores. The instruction format on the IBUS identifies the core selected for its execution. It also includes an annotated description of the physical location of all operands. For example, if architectural vector register `$vr12` is a source operand for an instruction to execute in core 2, a tuple of the form `<c1:r7,c2:r3>` will describe it in the annotated format. The first entry of the tuple (`c1:r7`) specifies that the content of architectural register `$vr12` is currently stored in the local vector register 7 of core 1. The second entry (`c2:r3`) notes that core 2 expects to read the content of `$vr12` from its local vector register 3 when it executes the instruction. Since the two entries are different, the tuple specifies a vector register transfer from core 1 to core 2. If the content of `$vr12` was already available in core 2, the two entries would be identical and the tuple would not trigger a transfer. The core to execute the instruction, as well as all the cores involved in the register transfers the instruction defines, read the instruction from the IBUS and write it in their instruction queue (IQ). The control logic in each core dispatches instructions from the IQ to a functional unit for execution and the proper input or output interface if any data exchanges are necessary.

Control Data Structures

Figure 6.3 describes the two data structures in the VIL, a renaming table and the per core control state. The renaming table maintains the current mapping between the vector registers in the instruction set and the actual registers available in the cores. Each entry corresponds to an architectural register. The core and local register fields specify the exact location of the current content of the architectural register. To process an instruction, the VIL must read the entries for all its operands. Once it makes the proper issue decisions, it must update the entries for any of the operands allocated for the first time or moved across cores.

For correct register allocation, the VIL must also maintain certain state for each core. A free list uses one bit per local vector register to identify those that do not store the data for some architectural register. An up/down counter tracks the size of the instruction queue (IQ). It increments when a new instruction is assigned to the core. It decrements when the core completes an instruction. The VIL also maintains in read-only registers the number of local vector registers and a bit mask that identifies the capabilities of the functional unit in each core. These registers are set at design time and allow the use of the same VIL logic with a variety of CODE configurations in terms of number and mix of vector cores.

The renaming table and free lists are similar to the data structures maintained in superscalar, out-of-order processors for architectural (virtual) to physical register mapping. The distinctive difference is that in CODE, the VIL processes just one instruction per cycle. A single vector instruction specifies a large number of element operations and can keep a core busy for several cycles. Hence, the complexity of VIL is independent of the number of cores in the system. On the other hand, a superscalar processor must issue multiple scalar instructions on every cycle to utilize a large number of functional units. Hence, it needs to perform multiple lookups and updates per cycle to the corresponding data structures [Soh90, SS95]. The complexity arises not only from the multiple access ports per data structure but also from the fact that the concurrent updates are dependent and the issue logic must maintain sequential semantics at all times. Consequently, the complexity of issue logic in a superscalar processor is an exponential function of the number of functional units it supports.

6.3 Issue Policies and Execution Control

This section discusses the options for issue policies and the implementation of chaining in CODE. It also provides an execution example that illustrates the function of the issue logic.

6.3.1 Issue Policies

The VIL block can use a number of alternative policies to select the core or allocate local registers for a vector instruction. We measure the effectiveness of each policy using two metrics: the number of register transfers between cores and, of course, performance. A large number of register transfers is undesirable since either they require an expensive communication network or lead to long stalls if sufficient network bandwidth is not available. However, transfers may be necessary in order to enable concurrent instruction execution in the vector cores. Ideally, we want to maximize performance while minimizing the number of transfers, but the two goals can often be incompatible.

The selection of a core to execute an instruction is trivial if a single core supports the necessary operation. If multiple candidates exist, we can use one of the following selection policies: random, load balancing, and minimum communication. To balance the load between cores with identical functionality, the VIL can use the size of their instruction queues as an indication of load. To minimize inter-core communication, the VIL must calculate the number of transfers necessary for each candidate assignment given the current location of vector operands.

After selecting the core for execution, the VIL must determine the local vector registers it will use for its sources and destination. For operands that are already available in the selected core, no decision is necessary. However, for input operands moved from another core, the VIL must allocate local vector registers to hold their values. The same holds for the destination register if its content is not stored in the selected core, even though no inter-core transfer is required. If the core has enough local registers free that do not store the value of any architectural register, the VIL simply selects one of them. Otherwise, it needs to make space by moving the contents of some local vector register(s) to another core. The potential policies for selecting local registers to replace are: random, least-recently-used (LRU), and most-recently-used (MRU). LRU assumes that applications

exhibit temporal locality and following instructions to execute in this core will likely access the same registers with previous instructions. In contrast, MRU assumes streaming applications, in which register values are often discarded after a single use. The choice between LRU and MRU is similar to that of choosing between caching [Smi82, Sez93] and streaming buffers [PK94, CB94] for applications running on cache-based microprocessors.

We analyze the impact of the different core selection and register replacement policies in Section 6.7.

6.3.2 Vector Chaining

The ability to chain the execution of vector instructions with the register transfers required for their operands is crucial for achieving high performance with CODE. Since each vector register contains a large number of elements, waiting for the whole register transfer to complete before any element operations can start would lead to frequent multi-cycle stalls. Chaining an instruction to a register transfer requires no additional inter-core signaling in CODE. The functional unit that executes the instruction must only check with the input interface in the same core. Every time a new set of elements for the input operand arrives, the input interface writes them to the LVRF. The functional unit can immediately read them and initiate the corresponding element operations. We support chaining between instruction execution and register transfers for all types of register dependencies: RAW, WAR, and WAW. WAW and WAR chaining allows a core to quickly reuse the storage of a local vector register as soon as its content starts being transferred elsewhere in the system, reducing the need for more local registers.

The chaining logic within each core needs to monitor the functional unit and the two interfaces for register dependencies. Since the core resources are constant regardless of the size of the overall processor, there are no scaling concerns for chaining logic. Chaining decisions on register dependencies are simple to make: the completion of an element operation in the functional unit or a transfer in the interfaces allows the dependent unit or interface to initiate the corresponding element action. Chaining control at a per element basis is the most flexible implementation possible, because it allows each element operation to start as soon as its data are available, regardless to stalls experienced by other element operations for the same instruction.

6.3.3 Execution Example

Figure 6.4 presents two execution cases for a vector add instruction assigned to core 1. In the first case (Figure 6.4.a), the renaming table indicates that both source operands, registers `$vr1` and `$vr2`, are already available in core 1. Therefore, the annotated instruction format indicates no inter-core register transfer for them. No transfer is necessary for the destination register `$vr0`, even though its content is initially available in core 2, because the add instruction will overwrite it. After the instruction issues, the renaming table indicates that local register 2 in core 1 is the new location for `$vr0`. The execution of the instruction involves only core 1, which performs its element operations within a certain number of cycles.

In the second execution case (Figure 6.4.b), the initial location of architectural register `$vr2` is core 2. Therefore, the annotated instruction format indicates a register transfer from core 2 to core 1 using the tuple `<c2:r0,c1:r3>`. Once the instruction issues, the renaming table holds the new mapping for `$vr0` and `$vr2`, local registers 2 and 3 in core 1 respectively. The actual execution involves both cores. Core 1 dispatches the instruction to both its functional unit and the input interface that generates an input transfer request. Core 2 generates the corresponding output transfer request by dispatching the instruction to its output interface. The communication network can start the transfer after matching the two requests. The arrival of `$vr2` elements in core 1 allows the execution of the corresponding element operations in a chained fashion. When the execution completes, the contents of all three operands are located in core 1.

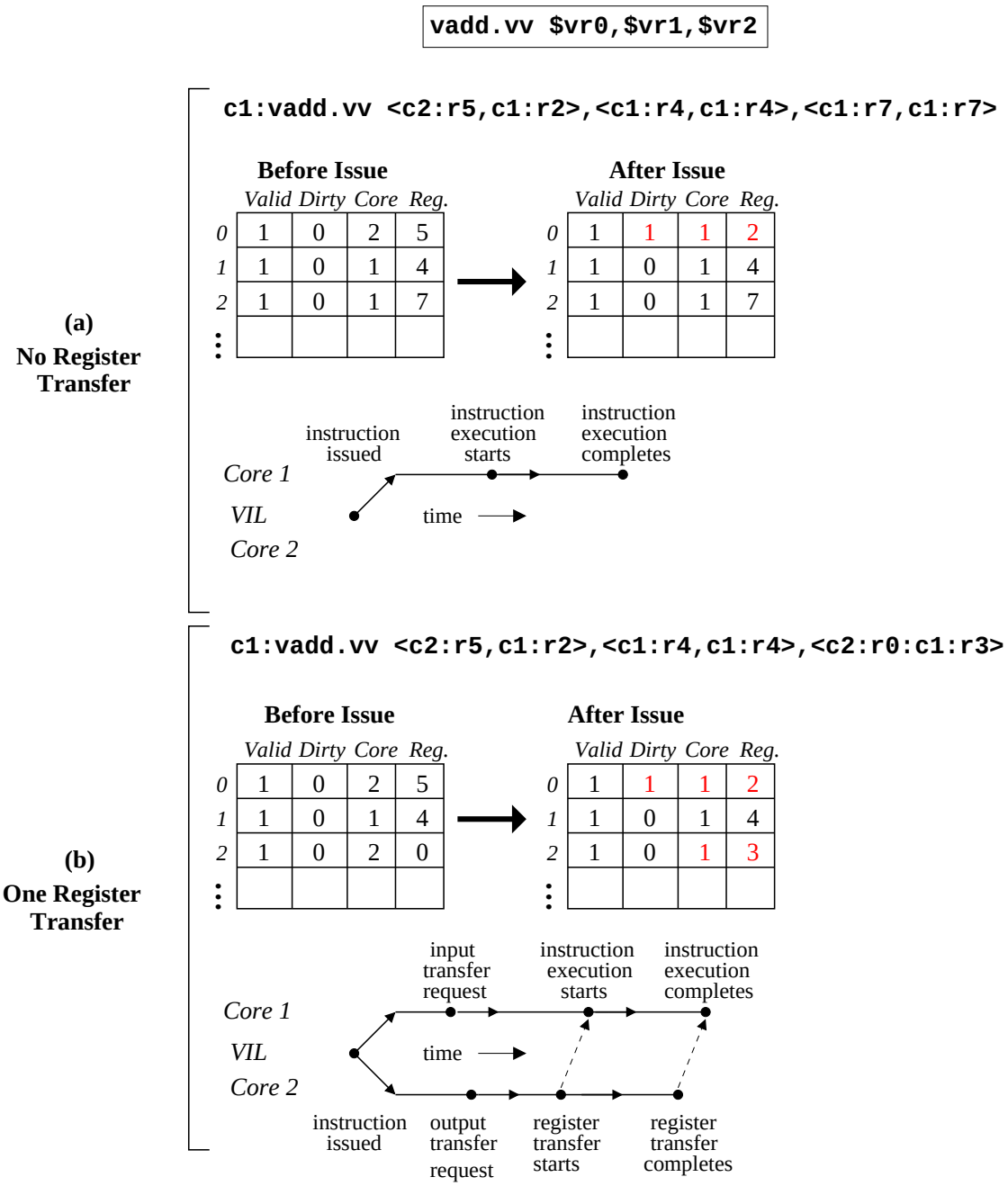


Figure 6.4: Two execution cases for a vector add instruction assigned to core 1. For each case, we present the annotated instruction format, the renaming table before and after the instruction issues, and a time-line of the actions required to execute the instruction. In case (a), we assume that the architectural registers `$vr1` and `$vr2` correspond initially to local registers 4 and 7 in core 1. In case (b), the contents of register `$vr2` are initially in local register 0 in core 2. The destination register `$vr0` is initially available in core 2, local register 5 in both cases. Case (a) requires no inter-core communication. Case (b) requires a register transfer from core 2 to core 1 for the contents of architectural register `$vr2`.

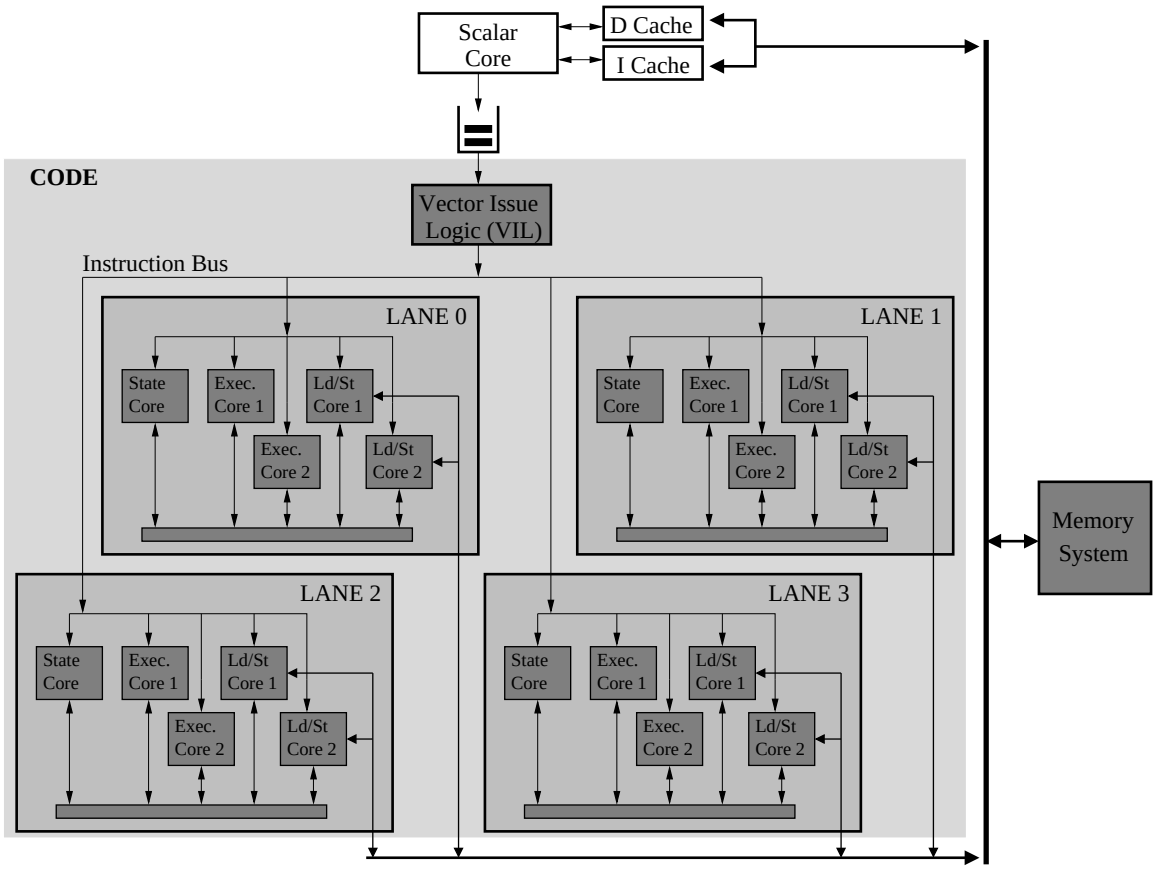


Figure 6.5: An implementation of the CODE microarchitecture with four vector lanes. The processor includes two execution and two load-store cores. The hardware resources of each core are distributed across the four lanes.

6.4 Multi-lane Implementation of Composite Organizations

Incidentally, the composite nature of CODE does not prohibit the use of vector lanes. Figure 6.5 presents an implementation of the microarchitecture with four vector lanes. Each lane contains part of each core, including a vertical partition of the LVRF, a 64-bit datapath from the functional unit, and a 64-bit slice of the interfaces to the communication network. We can replicate the core instruction queues in every lane or place them in a centralized location that connects to all lanes. The latter approach is preferable only for designs with few lanes or few cores per lane due to the wiring resources it requires.

Figure 6.5 shows the load-store core distributed across the four lanes. This approach simplifies the structure of the communication network. However, it may require the replication of resources such as the TLB or the output address queue. If the area overhead is significant, we can place these resources in a centralized location and allow all lanes to share them. A micro-TLB per load-store core can reduce the frequency of accesses to the main TLB and allow resource sharing without noticeable performance penalty.

Figure 6.6 shows how lanes and cores provide two orthogonal methods for scaling implementations of the CODE microarchitecture. Multiple lanes execute in parallel a large number of element operations for each instruction in progress. Multiple cores overlap the execution of multiple

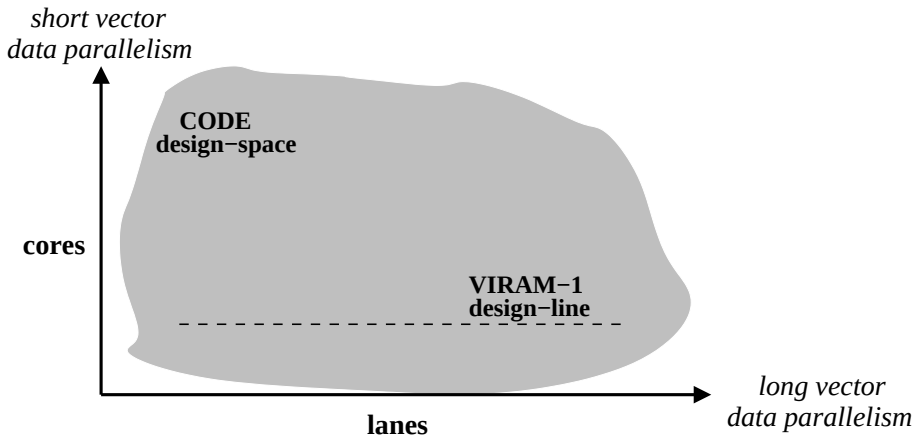


Figure 6.6: The two scaling dimensions of the CODE microarchitecture. Multiple lanes exploit parallelism in the form of long vectors, while multiple cores help with parallelism in the form of independent vector instructions. For each implementation, we can choose the point in the design space that best matches the characteristics of its applications. In contrast, the VIRAM-1 microarchitecture has a single scaling dimension, since we can only vary the number of lanes in the processor.

vector instructions. For applications with long vectors, we can organize hardware resources in many lanes and few cores. If long vectors are not available, we can exploit independent vector instructions by implementing many cores with only a few lanes. In other words, we can select the number of cores and lanes that matches the requirements of applications in the most efficient way given the available hardware resources. Chapter 9 explores the optimal balance between cores, lanes, and communication network bandwidth for the EEMBC benchmarks.

All processors based on CODE are binary compatible. A single design database can create multiple implementations in a two-step process. First, we assemble a lane using a set of narrow core designs from the database. Next, we construct the heart of the vector processor by allocating the proper number of lanes. In other words, we can amortize the development cost of circuits for the cores, the VIL, and the communication network over a large number of implementations with diverge characteristics. This is a major advantage for CODE in terms of design complexity, along with the fact that the circuits for its basic components are straight-forward to develop due to their relatively simple structure.

6.5 CODE vs. VIRAM-1

CODE differs from the VIRAM-1 microarchitecture in two ways. First, it breaks up the centralized register file and distributes the vector registers across all cores. Second, it uses decoupling, instead of the delayed pipeline, to tolerate long memory latencies. The following subsections discuss in details the implications of the two differences.

6.5.1 Centralized vs. Distributed Vector Register File

The distributed register file in CODE associates a small number of vector registers with the functional unit in each core. In Section 6.1, we discussed two obvious advantages of this approach over the centralized vector register file in VIRAM-1. The local register file in each core has a small number of access ports that is independent of the number of cores in the system. Therefore, we can implement it from compiled SRAMs available in design libraries for semiconductor processes,

without resorting to the tedious full-custom techniques. In addition, the total number of vector registers becomes proportional to the number of functional units in the system. Therefore, there are always sufficient registers to stage the operands for executing one instruction in each core, regardless of the total number of cores in the system.

Furthermore, the distributed register file introduces significant area, energy, and latency advantages as we increase the number of functional units in the processor. By properly adapting the analysis by Rixner in [RDKM00], we can derive the following equations for the total area for vector registers, the register file access latency, and the energy consumed for an element access:

$$\textbf{Area} = c \cdot r \cdot e \cdot d \cdot (w + p) \cdot (h + p)$$

$$\begin{aligned}\textbf{Latency} = &\frac{w+p}{\nu_0} \cdot \sqrt{\frac{r \cdot e \cdot d}{L}} + log_4[(C_{word} + C_w \cdot (w+p)) \cdot \sqrt{\frac{r \cdot e \cdot d}{L}}] \\ &+ \frac{h+p}{\nu_0} \cdot \sqrt{\frac{r \cdot e \cdot d}{L}} + log_4[(C_{bit} + C_w \cdot (h+p)) \cdot \sqrt{\frac{r \cdot e \cdot d}{L}}]\end{aligned}$$

$$\begin{aligned}\textbf{Energy} = &(C_{word} + C_w \cdot (w+p)) \cdot E_0 \cdot \sqrt{\frac{r \cdot e \cdot d}{L}} \\ &+ a \cdot (C_{bit} + C_w \cdot (h+p)) \cdot E_0 \cdot \sqrt{\frac{r \cdot e \cdot d}{L}}\end{aligned}$$

Parameters c, r, and p represent the number of cores in the system, the number of registers per core, and the number of access ports per register respectively[1]. For the centralized vector register file in VIRAM-1, there are 32 vector registers ($r = 32$) in one core ($c = 1$), and the number of access ports per register is typically $p = 3 \cdot FU$, where FU is the number of functional units. In CODE, each functional unit is in a separate core with a local vector register file, hence $c = FU$. The number of access ports per local register is typically $p = 5$. The number of registers per core (r) is a design parameter of the core.

Figure 6.7 provides a graphical representation of the three equations for VIRAM-1 and CODE. For the centralized register file of VIRAM-1, the total area for vector registers is a square function of the number of functional units (FU), even though the register file capacity is constant (vector 32 registers). The register access latency grows with the logarithm of FU because the lumped capacitance of the pass transistors for the access ports typically dominates the capacitance of the word-line and bit-line wire. The increased latency creates a serious limitation for the clock frequency of the whole processor. The energy for an element access is a linear function of FU, due to the capacitance introduced by the additional access ports for each functional unit. Consequently, implementing the VIRAM-1 microarchitecture with a large number of functional units becomes prohibitively expensive with all three metrics.

On the other hand, both the area and the capacity (number of registers) of the distributed register file in CODE are proportional to the number of functional units. The latency and the energy for an element access are independent from FU, since the local register file in each core has fixed size and a constant number of access ports. Therefore, the distributed organization for the vector register file is appropriate for implementations with a large number of functional units.

[1]The remaining design and technology parameters in the three equations are: e is the number of elements per vector register; d is the element width in bits; L is the number of lanes; w and h are the width and height of single-bit storage cell; C_{word} is the input capacitance of the word select transistor in a register cell; C_{bit} is the capacitance introduced by the storage cell to each bit-line; C_w is the wire capacitance per unit length; ν_0 is wire propagation velocity; E_0 is the energy required to charge a unit of capacitance.

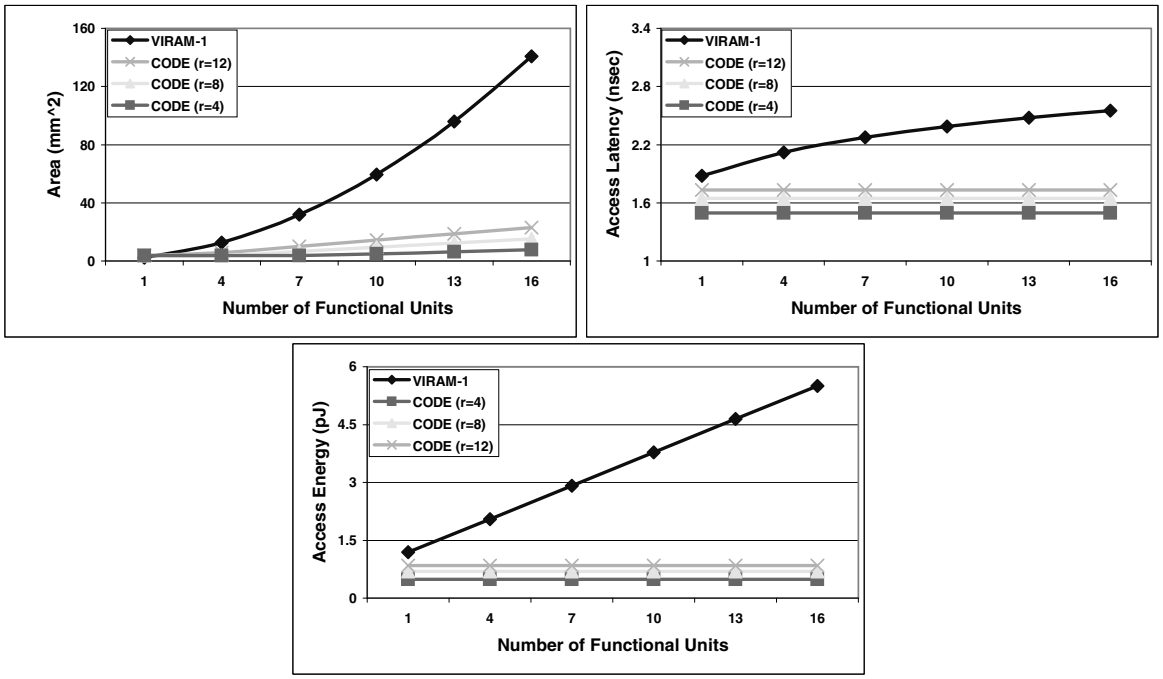


Figure 6.7: The register file area, register access latency, and energy for an element access as a function of the number of functional units (FU) in VIRAM-1 and CODE. For CODE, we present separate curves for the cases of r =4, 8, and 12 local vector registers per core. The values used for the technology parameters are derived from the IBM SA27E process and are representative of 0.18μm CMOS technology.

However, the distributed vector register file incurs an additional energy overhead for inter-core register transfers. Such transfers dissipate energy on the wires of the communication network. For any efficient network design, the length of wires is proportional to the square root of total area occupied by the cores. Similarly, the length of the wires that connect the boundary of the centralized vector register file in VIRAM-1 to the inputs of the functional units is proportional to the square root of the total area occupied by the functional units. In VIRAM-1, energy is consumed on the long wires for every register access, while in CODE energy is consumed on long wires only for register accesses that trigger inter-core transfers. Figure 6.8 presents the total energy consumed by a vector instruction for reading, writing, and transferring elements between the register file(s) and the functional unit for VIRAM-1 and CODE. The distributed register file leads to higher energy efficiency when the average number of inter-core register transfers per vector instruction (t) in CODE is low, or when the total number of functional units is large. Figure 6.9 presents the cross-over point in terms of number of functional units at which CODE exceeds VIRAM-1 in register file energy efficiency as a function of t. For $t \leq 1.2$, CODE leads to better energy efficiency for all implementations with three functional units or more. Note that the maximum value for t is 2.

6.5.2 Decoupling vs. Delayed Pipeline

Figure 6.10 presents the execution of a five-instruction segment on the VIRAM-1 delayed pipeline and the CODE decoupled organization. The segment includes two long latency instructions: an unpipelined vector divide (`vdiv`) and an indexed load (`vldx`) that causes multi-cycle stalls due to memory conflicts. No data dependencies exist between the five instructions. In the delayed

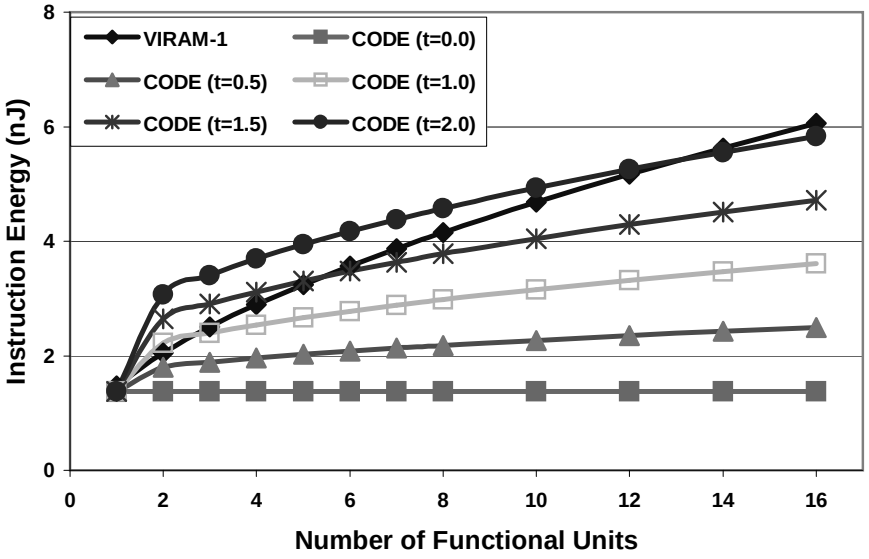


Figure 6.8: The total energy consumed by a vector instruction for reading, writing, and transferring elements between the register file(s) and a functional unit as a function of the number of functional units (FU). The instruction reads two vector registers as sources and writes one as the destination. Each vector register has 32 64-bit elements. For CODE, we present curves for five different values of the average number of inter-core register transfers per instruction (t). The curve for $t = 2.0$ represents the worst case in which all instruction operands require an inter-core transfer. Each core has $r = 8$ local vector registers. The estimates for the area of functional units and cores, as well as for the wire capacitance were derived from the VIRAM-1 prototype chip.

structure of VIRAM-1 (Figure 6.10.a), the pipelines of the functional and load-store units work in a lock-step in order to preserve dependencies. Thus, the stalls introduced by the two high latency operations also delay the processing of any instructions executing in parallel, despite the lack of register dependencies. Moreover, the vector multiplication (`vmul`) cannot issue until one of the two previous instructions has completed its execution in one of the two arithmetic units of VIRAM-1. Because of the in-order issue policy, the load instruction cannot issue earlier either, even though the load-store unit is idle.

Decoupling (Figure 6.10.b) removes both limitations and allows the code segment to execute faster. Unless there are register dependencies, the pipelines of different cores do not interact. Stalls in long latency operations do not affect instructions executing in other cores. Hence, the vector add instructions (`vadd`) can complete in the second execution core, despite the multi-cycle stalls for the divide (`vdiv`) in the first one. Furthermore, the instruction queue in each core can receive new instructions, while an older instruction occupies the functional unit. All instructions in the segment issue to cores in-order in back to back cycles. The multiplication (`vmul`) and subtract (`vsub`) instructions still have to wait for previous instructions to complete before their execution starts. However, the indexed load can start fetching elements immediately after issue and the data will become available faster for following instructions that use them for operands.

6.6 CODE vs. Out-of-Order Processors

At this point, it is also useful to emphasize the differences between the CODE microarchitecture and processor organizations for out-of-order (OOO) execution. Both approaches include a

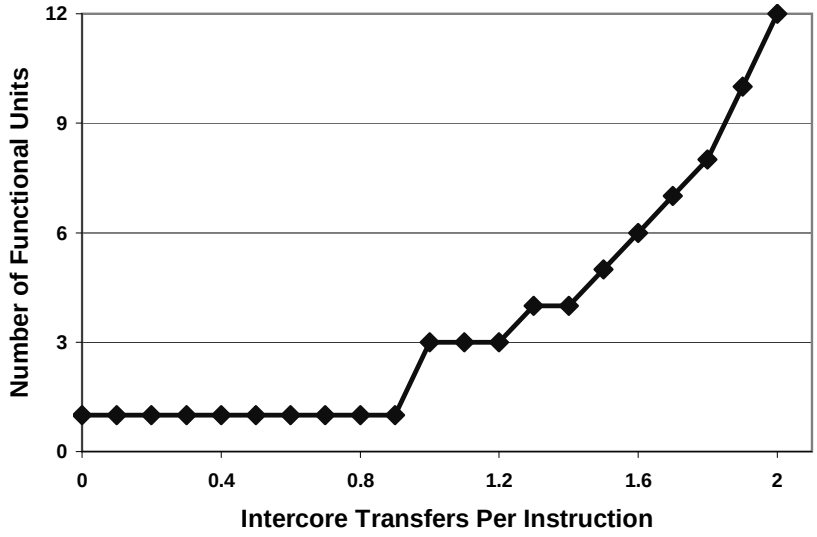


Figure 6.9: The minimum number of functional units (FU) for which the distributed register file of CODE leads to lower energy consumption for accessing register operands per instruction than the centralized register file organization in VIRAM-1. We calculate the number of functional units as a function of the frequency of inter-core register transfers per instruction (t) in CODE. For $t \leq 0.9$, CODE is always more energy efficient that VIRAM-1. For a processor with three functional units as in the case of the VIRAM-1 prototype chip, CODE dissipates less energy in the register file as long as $t \leq 1.2$,

large number of physical registers, which enable the elimination of name dependencies (WAW and WAR register hazards) through register renaming. However, there are significant differences in the control logic that orchestrates instruction scheduling and execution in each approach.

The CODE microarchitecture issues instructions to vector cores in strict program order. In addition, each vector core maintains program order for the instructions assigned to it by always executing them in the order they were received. However, instructions without data dependencies assigned to different cores may execute and complete out of order. Hence, we can characterize CODE as an architecture with in-order instruction issuing and scheduling, but potentially out-of-order instruction execution and completion. On the other hand, OOO organizations are more aggressive in terms of reordering. They issue instructions in order to reservation stations or an instruction window (buffer), from where instructions may be scheduled and executed in any order that does not violate data and control dependencies [Soh90].

In-order scheduling limits the ability of CODE to reorder instructions and eliminate all unnecessary dependencies. Two arithmetic instructions issued to the same core will execute sequentially, even if the operands of the second one become available earlier than the operands for the first instruction. An OOO processor with an idling functional unit would try to reorder the two instructions and execute the second one while the first one waits for its operands. However, we don't consider in-order scheduling a significant limitation for CODE. The most important benefit from reordering is the ability to prefetch data by executing load operations as soon as the addresses are available. By decoupling the load-store cores for those for arithmetic operations, CODE can look ahead in the instruction stream and initiate data prefetching for load instructions before preceding arithmetic operations actually execute. Therefore, the reordering capabilities of CODE are sufficient to tolerate long memory latencies. Reordering of arithmetic instructions with respect to each other has a smaller impact on performance. A compiler can achieve most of its benefits with

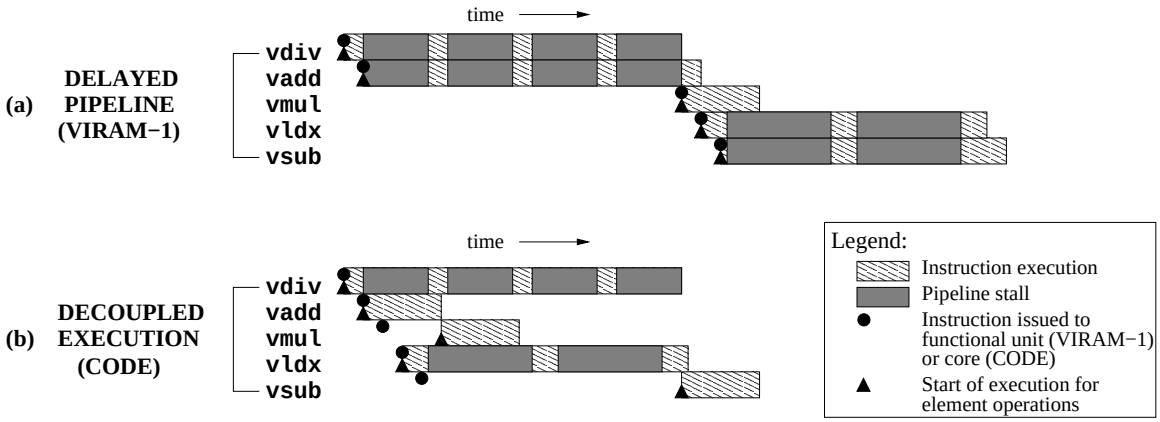


Figure 6.10: The execution of a five instructions on the (a) delayed and (b) decoupled pipelines. The code segment includes no register dependencies. The delayed pipeline assumes the VIRAM-1 configuration with two arithmetic and one load-store (memory) units. The decoupled pipeline assumes two execution cores for arithmetic instructions and one load-store core. For simplicity, we do not show the delay stages at the beginning of the execution of each instruction on the delayed pipeline.

static scheduling, since the latency of arithmetic instructions is short and predictable, unlike the latency of memory operations. In addition, OOO organizations use full reordering along with speculative execution to alleviate the cost of conditional branches [YP92]. In contrast, CODE uses vector instructions, which eliminate a large percentage of the predictable branches and the performance overhead associated with them.

The advantage of in-order scheduling in CODE is reduced design complexity. With relatively simple data structures, the VIL block can implement renaming and coordinate register transfers among cores at issue time, long before the instruction has its operands available and can execute. CODE does not require the instruction window of OOO organizations or the complicated, associative logic for dynamically scheduling instructions based on the availability of their operands. In addition, the in-order execution of instructions within each core guarantees that the communication network in CODE is free of deadlocks (see Section 6.2, page 62). If the cores could reorder the instructions assigned to them, just as the functional units in OOO processors using the Tomasulu algorithm [Tom67] can reorder the instructions assigned to their reservation stations, deadlocks would be possible in the communication network. The logic for detecting or preventing deadlocks can be quite complex, as it needs to monitor the progress of instruction execution across all cores.

6.7 Microarchitecture Tuning

In this section, we use the EEMBC benchmarks to tune some of the basic microarchitecture parameters for CODE, such as the issue policies and the number of local vector registers per core. We present a complete performance evaluation for CODE in Chapter 9, which also explores the trade-off between the number of cores and the number of lanes in the system.

The main figure of merit for this study is the average number of inter-core register transfers per vector instruction (`t`). Fewer transfers means less energy consumed on the wires of the communication network. A policy that results in a small average has the potential of high performance even with CODE configurations that have limited network bandwidth. However, the average number of transfers can also be low due to decreased concurrency of execution across the vector cores.

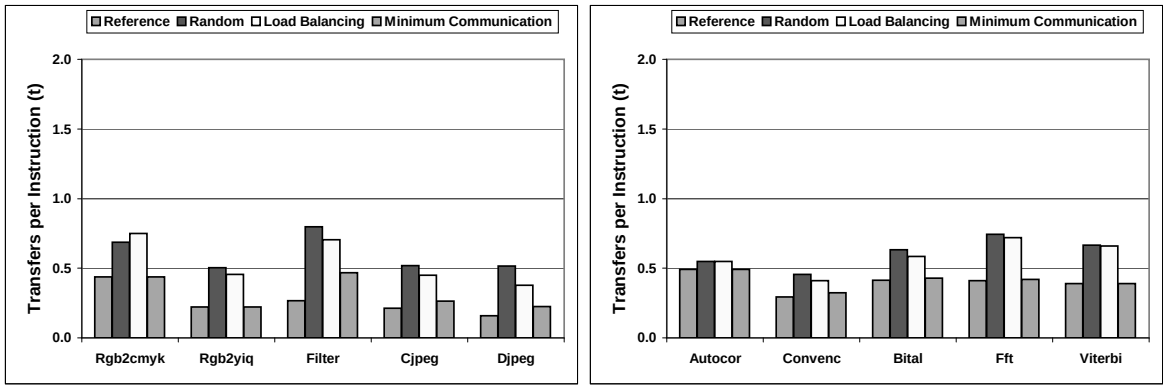


Figure 6.11: The average number of inter-core register transfers per vector instruction (`t`) for the three core selection policies. The first bar for each benchmark refers to the reference configuration with a single integer and load-store core. The other three bars represent a configuration with two integer and two load-store cores using the corresponding selection policy. Each core includes $r = 32$ local vector registers in both configurations. The maximum possible value for `t` is 2.0. A lower average is better.

Therefore, in certain cases we must also refer to raw performance in order to select the optimal value for a parameter.

We analyze the behavior of CODE using a parameterized trace-driven performance model. The model allows us to vary independently a large number of microarchitecture parameters including the number of cores and lanes, the latency of execution and communication events, the issue policies, and the characteristics of the memory system. The flexibility of the performance model enables the evaluation of a large number of realistic implementations of CODE. It also allows us to create configurations that isolate and emphasize the effect of specific design parameters and indicate the optimal values for them.

It is interesting to notice that the composite and decoupled nature of the CODE microarchitecture simplified significantly the development of the highly parameterized performance model. Changes in the parameters for a system component, such as a core or the VIL, do not affect the structure of the models for other components. They only affect the exact timing of input and output events for other models. With the delayed pipeline of VIRAM-1, on the other hand, concurrent execution occurs across functional units occurs in a tied lock-step. Almost any parameter variation requires global modeling changes in order to adjust the overall pipeline length or the location of the pipeline stage for a specific event.

6.7.1 Core Selection Policy

The core selection policy determines which core will execute a specific instruction in the case that multiple cores include the proper datapaths. The alternative policies, discussed in Section 6.3 (page 67) are random, load balancing, and minimum communication. To evaluate their impact, we set up a CODE configuration with two cores for integer arithmetic (*IntFull*) and two load-store cores (*LDFull*). To eliminate any register transfers or stalls that are not related to the core selection policy, we used $r = 32$ local vector registers per core and a constant latency memory system with infinite bandwidth.

For comparison, we also set up a reference configuration with one integer and one load-store core and $r = 32$ vector registers per core. This reference design exhibits the minimum number of register transfers for each benchmark. A register is transferred across cores only if its value produced

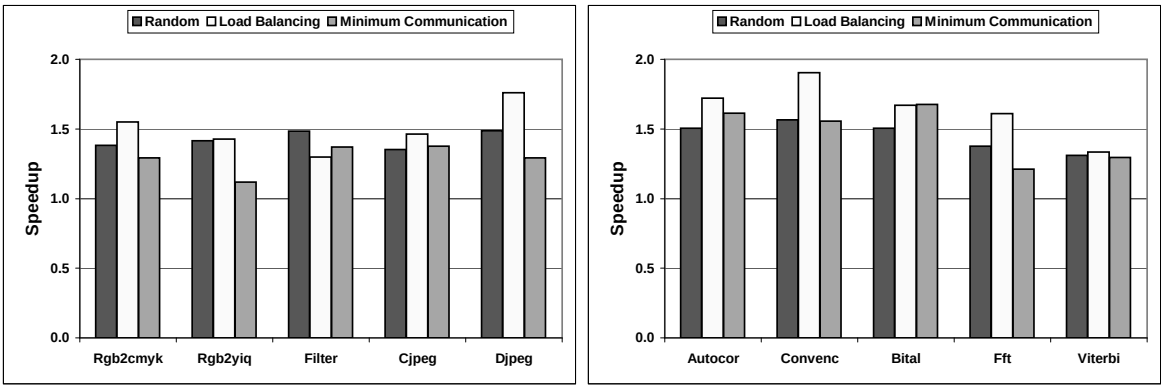


Figure 6.12: The speedup of the multi-core CODE configuration over the reference design for the three core selection policies. Both configurations use the same memory system, have a single lane, and have unlimited bandwidth available in the communication network. For `Cjpeg` and `Djpeg`, we only measure the execution time of the vectorizable functions. With twice as many cores in the multi-core system, the maximum possible speedup is 2. A higher speedup is better.

in an integer core is the operand of a load-store instruction and vice versa. There are no transfers due to core selection or register replacements with the reference configuration.

Figure 6.11 presents the average number of register transfers (`t`) for the three core selection policies. As expected, the minimum communication policy matches the number of transfers in the reference system for all benchmarks excluding `Filter`. The random and load balancing schemes, on the other hand, generate up to twice as many transfers, with the load balancing approach being slightly better.

From Figure 6.11 alone one could conclude that the core selection policy of choice is minimum communication. However, this policy has the tendency to execute most instructions in a single core and limits the degree of concurrency in the system. Figure 6.12 compares the performance potential of the three policies. It presents the speedup of the multi-core configuration over the reference design under the optimistic assumption of infinite bandwidth in the inter-core communication network. The load balancing scheme spreads the workload evenly across all cores and can deliver up to 40% higher performance than the minimum communication approach for applications like `Rgb2yiq`, `Djpeg`, `Convenc`, and `Fft`.

The conclusion from Figures 6.11 and 6.12 is that the core selection policy based on load balancing is most appropriate for CODE configurations for maximum performance. The minimum communication policy is practical for designs that target minimum energy consumption or have limited resources available in the communication network.

6.7.2 Register Replacement Policy

The register replacement policy (see Section 6.3, page 67) is important for CODE configurations with a small number of local vector registers per core. To measure the impact of the three candidate policies, random, LRU, and MRU, we simulated a CODE configuration with $r = 4$ registers per core. To mask out any register transfers due to core selection, we included a single integer and a single load-store core in the system. The minimal number of cores places maximum pressure on the local register file in each core and should reveal the full potential of each replacement policy.

Figure 6.13 presents the average number of register transfers (t) with each replacement policy for the EEMBC benchmarks. `Autocor` and `Convenc` are hardly affected by the choice of

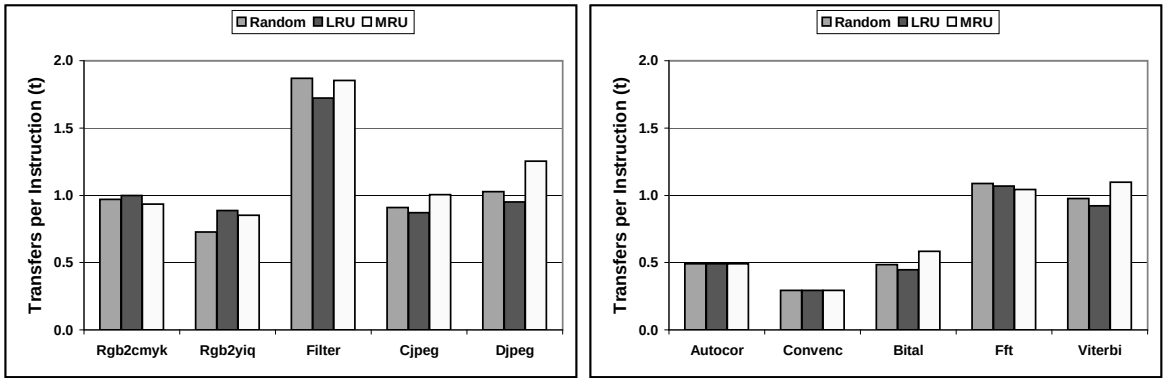


Figure 6.13: The average number of inter-core register transfers per vector instruction (`t`) for the three register replacement policies. All numbers refer to a CODE configuration with one integer and one load-store core. Each core includes only $r = 4$ local vector registers. The maximum possible value for `t` is 2.0. A lower average is better.

policy because they use less then six architectural registers for instruction operands. LRU results to slightly less transfers for `Cjpeg` , `Djpeg`, `Bital`, and `Viterbi`, where each vector value produced by arithmetic instructions is used as an input operand in many of the following instructions. On the other hand, MRU replacement is slightly better for `Rgb2cmyk` and `Fft`, where the result of each vector instruction is typically used once or twice as an input operand. Random replacement is usually in the middle, excluding `Rgb2yiq` for which it leads to the minimum number of transfers.

With just 4 register per core, the number of transfers for `Filter` is is above 1.7 regardless of the allocation policy. `Filter` uses 15 architectural registers for instruction operands. In addition, its main loop body is structured in way that both LRU and MRU result to frequent transfers. Random replacement can sometimes reduce the number of register transfers, but typically performs similarly to MRU, as reported in Figure 6.13. On the other hand, all other benchmarks require less than 1.0 transfer per instruction. This is true even for `Cjpeg` and `Djpeg` that use 25 vector registers in the code for forward and inverse DCT transforms.

Overall, the measured differences between the three replacement policies are small. It is unlikely that they can affect significantly the performance or energy consumption of a configuration, regardless of the bandwidth available in communication network. Therefore, it is preferable to use the random allocation policy for which we do not have to implement any hardware for approximating LRU or MRU selection.

6.7.3 Number of Local Vector Registers

Having set the register replacement policy, we can explore the impact of the number of local vector registers per core (`r`). Figure 6.14 presents the number of register transfers (t) as a function of `r`. The simulated CODE configuration includes only one integer and one load-store functional unit in order to put maximum pressure in the local register file in each core.

For most benchmarks, 6 to 8 local vector registers per core are sufficient to eliminate completely any inter-core transfers due to register replacement. In addition, all benchmarks excluding `Filter` require less than 0.5 transfers per vector instruction for $r = 8$ registers. `Filter` requires at least 12 registers in order to mask the ineffectiveness of the replacement policy with its register access pattern.

The necessary number of local vector registers per core can be further reduced with help from the compiler. The compiler can use the valid bits defined in the VIRAM instruction set for the

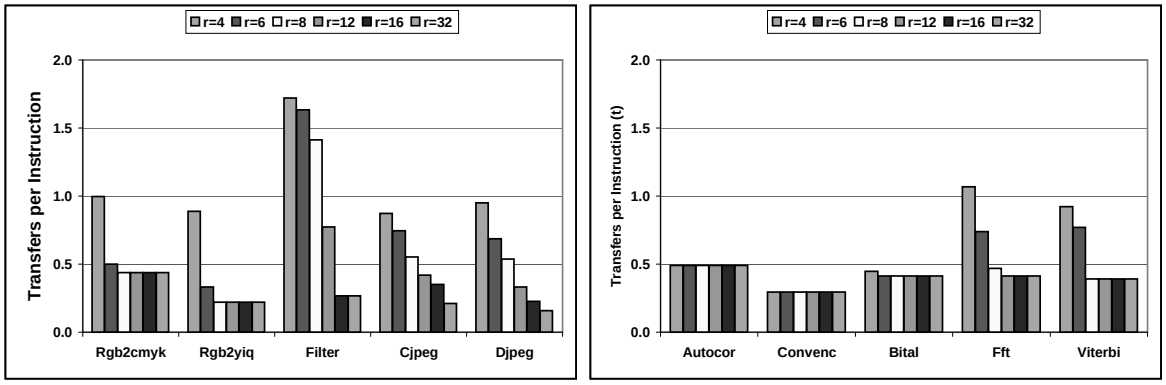


Figure 6.14: The average number of inter-core register transfers per vector instruction (`t`) as a function of the number of local vector registers per core (`r`). All numbers refer to a CODE configuration with one integer and one load-store core. The case of $r = 32$ local registers represents the best case with no inter-core transfers due to replacements. The maximum possible value for `t` is 2.0. A lower average is better.

32 architectural registers in order to notify the hardware when the contents of a register are no longer necessary and the corresponding local register can be reused without any inter-core transfers. For example, `Filter` would generate less than 1.0 transfers per vector instruction with just 8 registers if the compiler would mark as invalid any registers that don't carry useful data across loop iterations.

6.8 Related Work

Both composite organization and decoupled execution have been employed in various forms in a number of commercial and research architectures.

6.8.1 Composite Processors

Composing powerful processors by interconnecting simpler elements is an idea as old as the field of computer architecture. It is the basis of all parallel processors, regardless of the level of integration (multiple boxes, multiple boards, multiple chips, or single die) or the communication paradigm (shared memory or message passing) [CSG98]. Nevertheless, several recent uniprocessors have employed similar approaches for organizing hardware resources. The simpler cores within such designs are often described as "clusters," "cells," "slices," or "array elements."

The Alpha 21264 superscalar processor organizes integer resources in two clusters [Kes99]. Each cluster has a full copy of the architecture state. Clustering simplifies the register file design and keeps the forwarding latency low for instructions executing within the same cluster. In [FCJV97, Far97], Farkas proposes and analyzes a multi-cluster superscalar organization, where each cluster stores a static subset of the architecture state. The hardware performs the assignment of instructions to clusters based on the location of the registers it uses. Unlike with CODE, the multi-clustered superscalar design must issue one instruction per cycle to each available cluster, because each scalar instruction can keep a cluster busy for just one cycle. Consequently, the issue logic of the multi-clustered superscalar processor is highly complicated.

Several VLIW processors have also relied on composite techniques. The MAJC processor uses four slices, one for each operation in its 128-bit long instruction format [TCC+00]. A slice includes a functional unit and a set of registers, a subset of which is visible to the other slices (global

registers). Software, i.e. the compiler, is fully responsible for assigning instructions to slices and for scheduling inter-slice communication through global registers. In the Lx multi-cluster architecture, each cluster is a full four-way VLIW processor [FBFD00]. The clusters share instruction and data caches and communicate with explicit send-receive instructions inserted by software. The ManArray DSP architecture [Lev01] is a similar organization with multiple VLIW array elements per processor. Unlike these VLIW processors, CODE uses hardware to decompose the instruction sequence in multiple streams.

Finally, the Imagine stream processor [RDK+98] is a collection of ALU clusters. Each cluster has a local register file and executes VLIW instructions. Inter-cluster communication occurs by streaming data between cores or by accessing a separate, global register file. Software controls both instruction issue and communication in a micro-coded fashion.

6.8.2 Decoupled Processors

Despite its advantages with tolerating long memory and operation latencies, decoupled execution has fallen out of favor with modern microprocessors. However, several research and commercial architectures have used decoupling techniques in the past.

The IBM 360/91 [AST67] used two levels of queues to separate integer and floating-point instructions for parallel execution. Smith proposed and analyzed decoupling for memory accesses in [Smi84] and [SWP86]. This work lead to the Astronautics ZS-1 system [Smi89], which included one access and one execute processor that communicated through data queues. CODE is more similar to the PIPE [G+85] and MISC [TFP92] designs that used decoupling to run a single program on four processing elements that communicate through dedicated queues for each pair of elements. The WM architecture [Wul92] was a similar concept with the data queues visible to software through the instruction set. The ACRI processor [TM95, TRMM95] attempted to decouple the control portion of a program execution using a separate control processor in addition to the access and execution elements. CODE resembles this organization because it decouples the scalar core from the execution cores in the vector coprocessor. The scalar core resolves all branches and determines the control flow as early as possible.

Espasa studied a vector organization with decoupled memory accesses [EV96, Esp97]. Apart from tolerating higher memory latencies, decoupling reduced the overhead associated with register spilling due to the small vector register file. In [Asa98], Asanovic discussed several implementation issues for such a vector organization. CODE supports decoupling not only for memory accesses, but also for any two operations that execute in separate cores.

6.9 Summary

In this chapter, we introduced CODE, a new vector microarchitecture for multimedia execution. CODE eliminates the centralized vector register file, the main bottleneck for the VIRAM-1 organization. It consists of multiple interconnected cores, each with a small local vector register file and hardware resources for executing a subset of the vector instruction set. Decoupling techniques provide each core with a separate instruction stream, eliminate unnecessary dependencies, and hide long latencies. Along with a multi-lane implementation, CODE defines a microarchitecture with two orthogonal scaling dimensions that can exploit both vectorized and non-vectorized parallelism.

The next three chapters provide further information and evaluation of CODE. Chapter 7 describes a set of architectural and microarchitectural techniques that implement precise virtual memory exceptions for vector instructions. Chapter 8 discusses the memory system requirements of CODE, focusing mostly on on-chip, embedded DRAM organizations. Chapter 9 evaluates the performance potential of scale. It explores the tradeoff between cores and lanes and investigates the impact on performance of the bandwidth available in the inter-core communication network.

Chapter 7

Precise Virtual Memory Exceptions for a Vector Architecture

"A pessimist sees the difficulty in every opportunity;
an optimist sees the opportunity in every difficulty."
Winston Churchill

One of the biggest obstacles to the wide adoption of vector architectures in general-purpose computing systems is the difficulty of supporting virtual memory in vector hardware. Support for virtual memory is necessary for running a full-size operating system. The main challenge of virtual memory is implementing precise exceptions for translation errors. Vector processors either implement virtual memory with several restrictions that avoid exceptions or provide mechanisms for imprecise exception handling. However, imprecise exceptions complicate the development of operating system handlers and add to their execution overhead. This chapter explores a set of enhancements to the VIRAM instruction set and the CODE microarchitecture that implement precise virtual memory exceptions without introducing significant performance or area overhead.

Section 7.1 summarizes the general challenges in implementing precise exceptions and the additional hurdles with vector processors. Section 7.2 revisits the semantics of precise exceptions for virtual memory in the VIRAM architecture and provides an alternative definition with relaxed requirements on implementations. Section 7.3 describes the changes necessary in the CODE microarchitecture to implement precise virtual memory exceptions. Section 7.4 evaluates the impact of supporting precise exceptions on application performance and hardware resources. Finally, Section 7.5 reviews related work on precise exception support in general-purpose microprocessors.

The precise exceptions techniques presented in this chapter are general and can support both memory and arithmetic exceptions in vector instructions. However, we will focus mostly on the issue of precise exceptions for virtual memory violations such as TLB misses, refills, and write protection errors. Virtual memory exceptions require the invocation of an operating system handler before execution can be safely resumed. On the other hand, arithmetic exceptions in vector instructions do not always indicate significant execution errors and can be handled differently. Furthermore, arithmetic exceptions, such as overflow and underflow, are a secondary issue for embedded multimedia systems, because the overhead of running operating system handlers to fix them is prohibitive for real-time applications. Chapter 4 summarizes the support for arithmetic exceptions in the VIRAM instruction set that uses the flag registers and meets the requirements of software without the need for precise exceptions. In addition, we do not discuss exceptions due to sources other than

vector instructions. The scalar core can handle IO interrupts, scalar exceptions, and unknown or unimplemented instructions without significant impact on the vector coprocessor organization.

7.1 The Challenges of Precise Exceptions

When an instruction generates an exception, the processor must stop the execution of the current process and transfer control to an operating system handler. The handler uses the information provided by the hardware to identify the exception, fix it, and, if possible, resume the interrupted process. The status of the architecture state, in registers or main memory, at the time the handler starts is the characteristic that differentiates between precise and imprecise exceptions. An exception is precise if the architecture state corresponds with the sequential model of program execution, where one instruction completes before the next one begins [SP88]. All instructions preceding the one that causes the exception have completed their execution and written their results in the architecture state. The faulting instruction and any other that follow it have made no changes to the architecture state. If any of the above statements is not true, the exception is imprecise.

Precise exceptions are desirable because they simplify exception processing. The handler has easy access to the values of source operands for the faulting instruction. After handling the exception, the operating system can restart the interrupted process by resuming execution from the faulting instruction. With imprecise exceptions, on the other hand, the operating system must somehow compensate for the partial results of both preceding and following instructions during exception handling and process restart. In most cases, imprecise exceptions require saving and restoring intermediate results stored in hardware state not visible through the instruction set using operating system code that is specific to each processor implementation [MV96]. The high complexity of handlers for processors with imprecise exceptions usually discourages software and operating system development for such designs.

The difficulty of implementing precise exceptions rises from the departure of modern microprocessors from the sequential execution model. Pipelined implementations allow instructions to overlap their execution with a different instruction active in each pipeline stage. At the time a load instruction generates an address translation error, several instructions, both older and younger in program order, may be in progress. If the various functional units have a different number of pipeline stages, instructions may also complete out of order. The number of instructions executing concurrently is even larger with superscalar designs, where multiple instructions enter the pipeline in each cycle. Out-of-order issue and speculation techniques create further complications since they allow instructions to enter the pipeline out of program order and they often initiate instructions that should not execute at all.

In all cases, the processor must report exceptions in strict sequential order and, when an exception occurs, the processor must appear to commit instruction results to architecture state in order, even if this is not the case during normal execution. To maintain this impression, processors employ reordering techniques for instruction commit or mechanisms for reversing the effect of any instructions following the one that caused the exception [SP88].

Vector architectures introduce an additional level of difficulty to implementing precise exceptions. A vector instruction defines a large number of element operations. Each operation modifies the architecture state and may generate an exception condition for the whole instruction. Any practical implementation of vector hardware requires several clock cycles to execute tens of element operations. Therefore, a vector processor must handle instructions that involve a large amount of state and for which a long period is necessary to resolve their exception behavior. Moreover, even for a microarchitecture with in-order issue like CODE, chaining of vector instructions and parallel execution on multiple cores leads to out-of-order execution and completion. The *(i)*-th element operation for a vector instruction may execute well before the *(i+10)*-th operation for an instruction earlier in the program order. Consequently, supporting precise exceptions while processing instructions at

maximum speed can be more complicated for vector designs than for scalar processors.

7.2 Vector Architecture Support for Precise Virtual Memory Exceptions

The definition of precise exceptions requires that if any of the element transfers for a load or store instruction generates a translation fault, none of the element operations described by the instruction should modify the architecture state. This definition places hard requirements on the size of the TLB used for storing physical to virtual address mappings in a vector processor.

In order to guarantee forward progress, the TLB must have enough entries to map the maximum number of virtual memory pages accessed by one vector load or store instruction. Otherwise, if an instruction accesses a number of virtual pages larger than the number of TLB entries, a mapping fetched to translate an address for one element will evict the mapping for another element address, leading to an infinite sequence of virtual memory exceptions (livelock). With vector registers storing dozens of elements for narrow data-types and indexed loads or stores being able to access a separate memory page for each element, the minimum TLB size can be excessively large for most embedded implementations. Moreover, the TLB must typically be a little larger than the absolute minimum in order to reduce the time to select a TLB entry for a new address mapping that will not overwrite the mapping used for another element transfer in the same instruction.

To eliminate the need for a large vector TLB, we propose the following definition for precise exceptions for vector load and store instructions in the VIRAM architecture:

> "A vector load-store instruction that raises a virtual memory exception completes all element transfers up to and not including the first element that caused a translation error. All preceding instructions complete their execution and no following instruction makes any modifications to the architecture state."

The proposed definition maintains the same behavior with the original one for all instructions preceding or following the faulting one, but allows the faulting memory instruction to commit part of its result. For example, if the *10*-th and the *15*-th element operation in a vector load cause a TLB miss, the processor completes the memory transfers for the first 9 elements and modifies the corresponding element storage in the destination register. All other element transfers starting with the *10*-th are canceled and do not modify the architecture state in any way.

The modified definition enables vector processors to implement precise virtual memory exceptions and guarantee forward progress for vector load-store instructions even with a single TLB entry. When an instruction causes an exception, we can overwrite the TLB entries with mappings for the element transfers that have completed, without triggering translation errors for these elements when the process restarts. Each time a virtual memory exception for a vector instruction is processed, at least one more element operation will successfully complete before another exception occurs. Therefore, the TLB size is no longer a hard requirement but an implementation trade-off between hardware cost and performance. That is, a large, fully associative TLB occupies a significant amount of area but also reduces the frequency of TLB misses on vector memory accesses.

One control register is necessary to enable the operating system to resume the interrupted process by simply restarting the faulting instruction. On virtual memory exceptions, the processor updates this register with the number of the first element for which the address translation failed. When the interrupted process restarts, the same register indicates that the vector memory instruction must resume from the specified element number. All following vector instructions in the resumed process, will execute normally starting from element 0. If the operating system is about to resume a process that was not interrupted by a virtual memory exception, it must set the control register to zero.

7.3 Precise Virtual Memory Exceptions in CODE

The simplest way to implement precise virtual memory exceptions in CODE, and any other microarchitecture, is to revert to sequential execution on every vector memory instruction. We could stop issuing and processing following instructions until the load or store instruction completes address translation without errors [Asa98]. Even though this approach is simple and correct, it can reduce performance to unacceptable levels. It underutilizes arithmetic units during address translation and may lead to expensive pipeline draining on every load or store instruction. This section explores an alternative approach that implements precise exceptions in CODE without significantly affecting the sustained performance of the vector processor.

7.3.1 Hardware Support

The key to supporting precise virtual memory exceptions is to maintain in-order completion of vector instructions with respect to load and store operations. The following features are necessary to achieve this ability without prohibiting parallel instruction execution in the vector cores:

- We must be able to preserve the old values of any architecture state modified by instructions following a load or store instruction, until we know that the memory instruction will not raise any virtual memory exceptions.
- When a memory instruction raises an exception, we must be able to restore the old values in the architecture state affected by instructions following the one that generated the exception.
- We must be able to report virtual memory exceptions for vector instructions in strict program order.

We can use the local vector registers in the vector cores and the renaming capabilities of issue logic in CODE to meet the above requirements. Unallocated vector registers in the various cores can store the values of any operands modified by vector instructions without eliminating the old values, until we know that previously issued memory operations can complete without exceptions. If an exception occurs, we can restore the old values by reinstating the proper mappings in the renaming table of the issue logic. Therefore, the new functionality required to support precise virtual memory exceptions is a mechanism for revocable renaming and a technique for selecting when we must preserve the old values in the architecture state. We can implement both in the vector issue logic (VIL) without any changes in the organization of vector cores.

Figure 7.1 presents the two data structures added to the VIL to implement precise virtual memory exceptions. The *update queue* implements an in-order history buffer for changes to the renaming table [SP88]. Each entry describes one instruction issued to the vector cores, including its program counter (PC) and its annotated description. The *safe bit* indicates if the instruction will execute without causing exceptions, while the *fault bit* is set if the instruction has generated a virtual memory exception. The last field specifies the first element that caused an exception for a load-store instruction. The *guard list* is a bit mask with one bit per vector register in the instruction set. If the corresponding bit is set, an instruction that modifies the value or location of the vector register must preserve its old value. When the VIL processes a new vector memory instruction, it sets all bits in the guard list to indicate that future instructions must preserve the old values of any registers they modify until the memory instruction completes address translation.

The modified VIL processes vector instructions in the following way. It lookups up all instruction operands in the guard list. If the instruction changes the value or location of any operands with the corresponding guard bit set, the VIL allocates new local vector registers for them and does not deallocate the local vector registers with the old values or at the old location. Consequently, the VIL will preserve the old architecture state for these registers. Since following

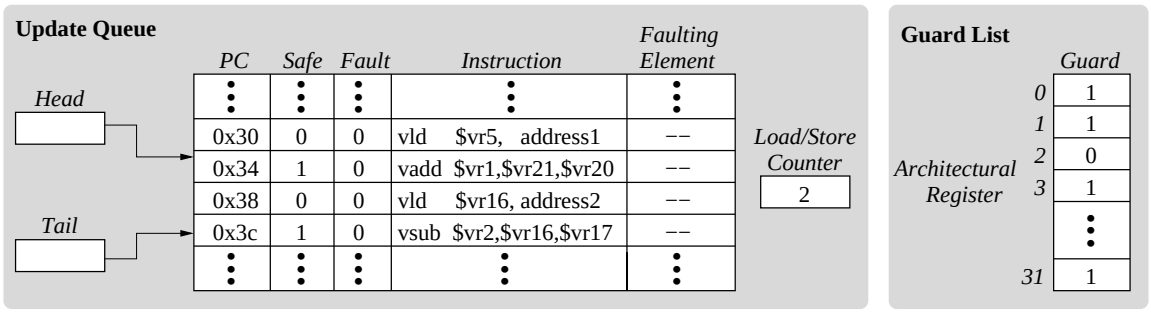


Figure 7.1: The data structures for implementing precise virtual memory exceptions in CODE. The update queue is a circular FIFO buffer with head and tail pointers maintained in registers. An up/down counter tracks the number of load or store instructions currently in the queue. The guard list indicates the architectural registers for which the processor must preserve the old values. On reset, the update queue is empty and all bits in the guard list are set to zero.

instructions should not preserve the new architecture state for these registers unless a new load or store instruction is issued, the VIL can clear the corresponding entries in the guard list. Upon issuing the instruction to the proper core(s), the VIL makes the necessary updates to the renaming table and pushes its description in the update queue. If the instruction is not a memory one, the safe bit is automatically set and the fault bit reset. For load-store instructions, the VIL clears both fields and waits for the selected core to set one of them when the instruction completes address translation or causes an exception.

On every clock cycle, the VIL examines the head of the update queue, which indicates the next instruction in program order for which exception behavior can be resolved. If its exception behavior is not available at the time, the VIL has to wait. Non-memory instructions cannot cause virtual memory exceptions. The VIL can remove them immediately from the queue and deallocate any local vector registers that hold old values for operands modified by these instructions. The same holds for vector load and store instructions that complete without exception. If the up/down counter indicates that are no more memory instructions in the update queue, the VIL can clear all entries in the guard list to indicate that old architecture state does not have to be preserved any more.

If the head of the queue describes a load-store instruction with translation errors, the VIL raises a virtual memory exception. It copies the program counter and the number of the first element transfer that generated an error to the corresponding control registers for exception processing. It also triggers a hardware finite state machine that uses the queue contents to undo all vector instructions following the one that caused the exception. The state machine processes one queue entry per cycle, starting from the tail, and uses the annotated instruction descriptions to undo the renaming mappings and new local vector register allocations. The state machine preserves the updates of the faulting instruction because the new definition for precise virtual memory exceptions allows for partial completion. At the end, the state machine flushes the queue and resets the bits in the guard list.

Because the update queue works as a history buffer for modifications to control data structures, the VIL uses the information it contains only when an exception occurs. The latest location of all architectural registers is always available in the renaming table. A two-ported register file used as a circular buffer is sufficient to implement the queue. On the other hand, if the update queue operated as a re-order buffer that allowed the modifications by an instruction to take place after exceptions are resolved in preceding instructions, the VIL would need to check the update queue in parallel with the renaming table for the location of instruction operands. Each queue entry would

need a set of comparators for these checks and priority encoders would be necessary to select the most recent entry with a valid mapping for a specific architectural register. A multi-ported, content-addressable array would be necessary to implement the queue in his case, which is complicated to design and is rarely available in design libraries for semiconductor processes. On the other hand, the re-order buffer requires no state machine for restoring architecture state. However, the state machine is negligible in terms of area when compared to the re-order buffer itself, and we can overlap the time required for its operation with the initialization of the operating system exception handler.

7.3.2 Implications to Performance

With the addition of the update queue and the guard list, the CODE microarchitecture can implement precise virtual memory exceptions while executing several instructions in parallel in the vector cores. However, the modified function of the VIL introduces new stall conditions that can lead to reductions in sustained performance.

When an instruction modifies the value or the location of a vector operand, the VIL must preserve the old values until any previously issued memory operations complete address translation. The processor cannot use the local vector register that temporarily stores preserved values for the instruction operands of any new instructions. Consequently, the system may run out of vector registers in one or more vector cores, causing the VIL to stall until older instructions retire from the update queue and deallocate the local registers that store old values for their operands. In addition, vector store instructions cannot transfer any elements to memory until all previous load-store instructions complete address translation, which generates additional pressure for local vector registers in the load-store cores. The VIL will also stall if the update queue is full, even if it could otherwise issue the instruction to a vector core.

We can reduce the performance loss due to the new stall conditions at the cost of higher area overhead in the vector cores and the VIL. Additional local vector registers in each core reduce the probability of running out of unallocated registers for new instructions. Similarly, we can increase the number of entries in the update queue. Alternatively, we can accelerate the address translation process for vector load-store instructions in order to discover their exception behavior faster and reduce the number of cycles each instruction spends in the update queue. The hardware resources necessary for this purpose are additional address generators, TLB and micro-TLB ports, and entries in the address queues in the load-store cores.

7.4 Evaluation of Performance Impact

This section provides a quantitative evaluation of the impact on performance of the new stall conditions introduced by the logic that implements precise virtual memory exceptions in CODE. In other words, we evaluate the performance penalty regardless of whether an exception occurs or not. We do not measure the performance of exception handlers or context switching because these issues are outside of the scope of this chapter. Measurements of handler performance would also require the availability of a full-scale operating system and the use of larger applications instead of the EEMBC benchmarks.

Table 7.1 presents the CODE configuration used for this study. It is similar to a single-lane VIRAM-1 system in terms of hardware complexity (area) and memory system. It includes an 8-entry update queue for precise virtual memory exceptions support. We vary the number of local vector registers (r) in the load-store and two integer cores, because we want to evaluate the number of local registers necessary to mask the performance loss due to stalls related with the logic for precise exceptions.

Figure 7.2 presents the percentage of performance lost with each benchmark when we enable the hardware support for precise virtual memory exceptions. With $r = 4$ local vector registers per

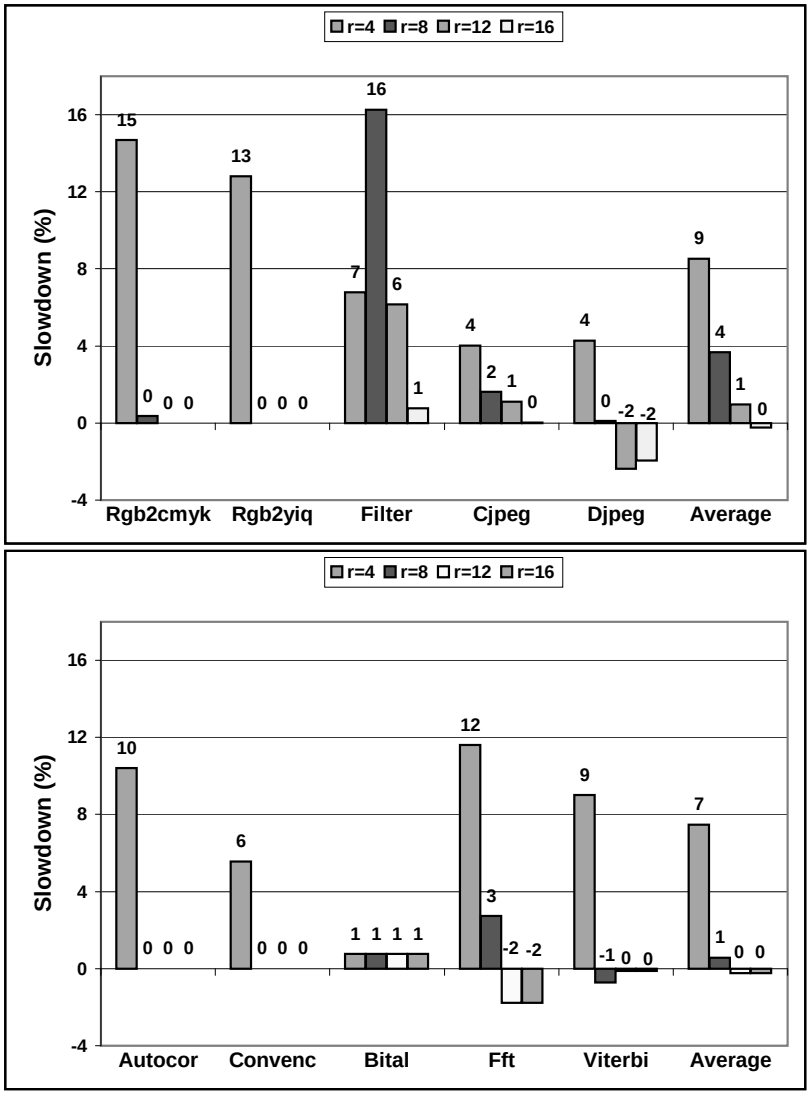


Figure 7.2: The performance loss (slowdown) due to hardware support for precise virtual memory exceptions in CODE. For each benchmark, we compare the execution time of a CODE configuration with a certain number of local vector registers per core (r) to that of the same configuration with the support for precise virtual memory exceptions enabled. The last group of columns presents the average slowdown for the five benchmarks. Positive slowdown means lower performance with precise exceptions support. Negative slowdown means higher performance with precise exceptions support. For `Cjpeg` and `Djpeg`, we only measure the vectorized functions of the benchmarks.

VIL	Load balancing core selection policy Random register replacement policy 8-entry update queue
Vector Cores	1 LDFull core (1 address generator) 1 IntFull core 1 IntSimple core 1 ArithRest core 1 State core
Lanes	1 (64-bit)
Memory System	8 DRAM banks 1 sub-bank per bank 64-bit crossbar with 2 cycle latency

Table 7.1: The CODE configuration used for evaluating the performance impact of precise virtual memory exception support. The configuration resembles the hardware complexity of a VIRAM-1 system with a single lane but without floating-point datapaths. The timing characteristics of the memory system are identical to that for VIRAM-1. The number of local vector registers in the one load-store core and the two integer cores varies in this study. The number of local vector registers for the ArithRest is set to $r = 4$. The number of local vector registers in the State core is $r = 8$ or as many as necessary to ensure that the whole system includes 32 vector registers across all cores. For details about the exact capabilities of the various cores, refer to Table 6.1 (page 65).

core, the performance loss varies from 4% to 16%. With $r = 8$ local vector registers, the number recommended for minimum inter-core communication (see Chapter 6.7, page 76), the performance loss drops below 1% for seven out of ten benchmarks. The small impact of precise exceptions support to performance is due to two reasons. First, we only maintain old architecture state after vector load and store instructions, which reduces significantly the number of local vector registers used for this purpose. Second, most of the times when the VIL must stall instruction issue due to insufficient space in the local register files or the update queue, the instruction queues in the vector cores have approximately 2, for $r = 4$, or 4, for $r > 4$, instructions in their instruction queue (IQ). Hence, the vector cores can proceed with instruction execution and the real impact of the VIL stalls to the overall performance is minimal. From Figure 7.2 one can also conclude that 8 entries in the update queue are sufficient. A smaller number of entries would not lead to noticeable area or energy savings.

`Filter` is the only benchmark that requires $r = 16$ local vector registers per core before the overhead of precise exceptions becomes less than 1%. This behavior is due to the many architectural registers it references in its code and the large number of inter-core register transfers it generates (see Chapter 6.7, page 76). The performance loss is actually larger as a percentage for $r = 8$ than for $r = 4$ local vector registers. The configuration without precise exceptions support for $r = 8$ can already eliminate a large portion of the inter-core transfers, which is not the case if some local registers are used to preserve old values. On the other hand, for $r = 4$ the number of inter-core transfers is quite large regardless of the support for virtual exceptions.

It is interesting to notice in Figure 7.2 that enabling precise exceptions support leads to higher performance (negative slowdown) for some cases with $r > 4$ for `Djpeg`, `Fft`, and `Viterbi`. The unexpected result is due to the interaction of the additional control logic with the policy for core selection. Stalls related to precise exceptions allow the VIL to postpone the core selection for integer instructions. At a later time, the load balancing decision can be more accurate as it reflects more the number of cycles it takes to execute the instructions assigned to each core as opposed to the number of instructions assigned to each core. In any case, the outcome of this interaction is practically insignificant.

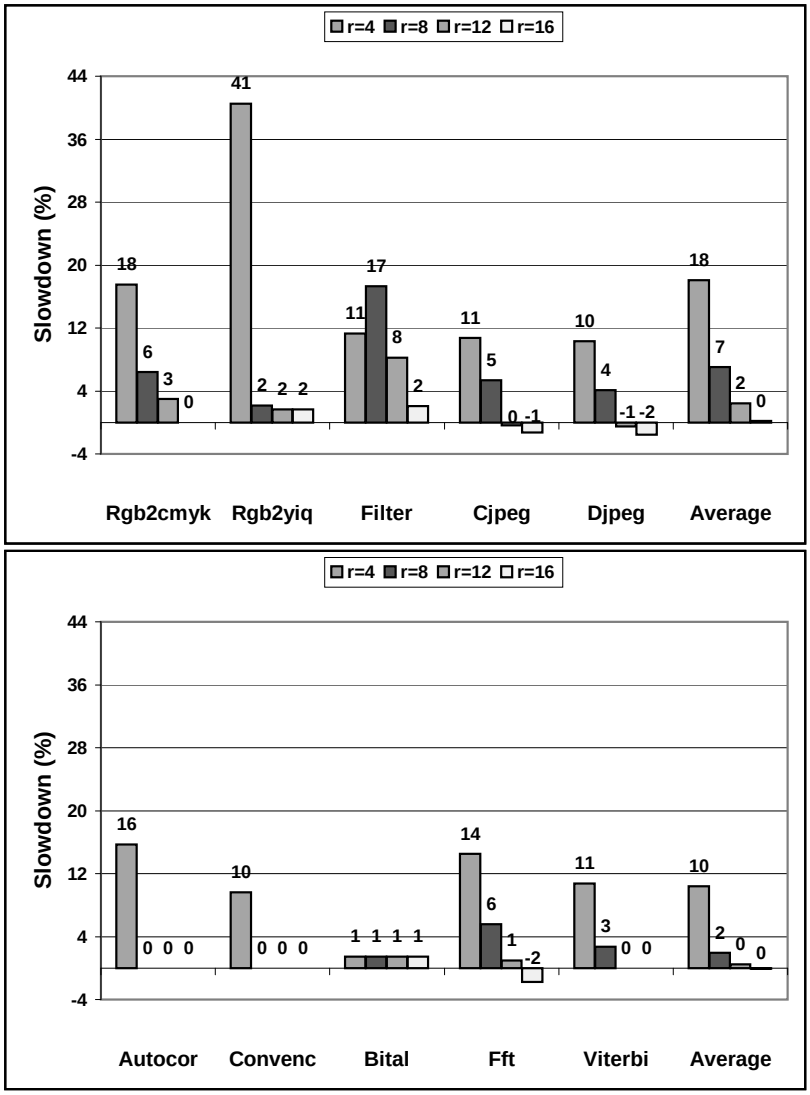


Figure 7.3: The performance loss (slowdown) due to hardware support for precise exceptions for both virtual memory and arithmetic faults in vector instructions. For each benchmark, the reference point is the performance of the simulated CODE configuration without any support for precise exceptions. The last group of columns presents the average slowdown for the five benchmarks. Positive slowdown means lower performance with precise exceptions support. Negative slowdown means higher performance with precise exceptions support. For `Cjpeg` and `Djpeg`, we only measure the vectorized functions of the benchmarks.

Finally, Figure 7.3 presents the performance impact from extending the proposed mechanism in order to support precise arithmetic exceptions for vector instructions as well. With almost every vector instruction being the likely source of an exception, the VIL must frequently preserve the old values of architectural registers, which places additional pressure on the local vector registers in each core. Consequently, the performance loss with $r = 4$ vector registers per core becomes significantly higher and reaches 40% for `Rgb2yiq`. With $r = 8$ vector register per core, however, the degradation of performance is negligible for the smaller benchmarks (`Autocor`, `Convenc`, and `Bital`) and less than 6% for the remaining applications, excluding `Filter`. With $r = 12$, the overhead is 2% or less except for `Filter`, which experiences an 8% performance loss. Hence, the proposed mechanism can also support precise arithmetic exceptions for vector instructions with only a small increase in performance or area overhead over the base case for precise virtual memory faults.

Nevertheless, it is important to keep in mind that enabling any support for arithmetic exceptions can lead to dramatic performance loss in real-time applications, due to the high overhead of running operating system handlers for arithmetic exception processing. Hence, precise arithmetic exceptions should be optional.

7.5 Related Work

The problem of precise exceptions came up with the first pipelined processors [Buc62]. The solution suggested in several early designs was to equalize the pipeline lengths for all functional units and allow instructions to update architecture state and report exceptions only in the final stage [Amd81, War82]. In [SP88], Smith proposed three basic methods for implementing precise exceptions in high performance processors: the re-order buffer, the history file, and the future file. All modern superscalar processors use re-order buffers to support precise exceptions, while simpler designs and VLIW architectures usually employ the pipeline equalization technique [HP02, PH98]. The update queue in CODE is practically a history file for updates to the renaming table.

Despite the implications to software development, some older commercial processors implemented imprecise exceptions under some circumstances [AST67, Tho70, HT72, Con81]. A common mechanism for handling imprecise exceptions is to allow software to save and restore machine-dependent pipeline state [Con81, HJBG82, Dig96, KGM+00]. In [MV96], Modgill and Vassiliadis review a number of software and hardware techniques for defining and implementing imprecise exceptions in ways that are manageable for software.

Several vector computers do not support virtual memory due to the difficulty of implementing precise exceptions [Rus78, Tho70]. This approach is acceptable in the supercomputing domain, where a single application runs for long periods and maximum performance is more important than ease of software development. Some vector processors support virtual memory [Jon89, UIT94], but in most cases in a restricted mode that eliminates virtual memory exceptions for vector instructions. For example, the processor may need to keep locked in main memory all the pages accessed by vector instructions [HL96]. In [Asa98], Asanovic proposed a decoupled vector pipeline that supports precise memory exceptions by stalling all instructions until preceding memory operations complete address translation, but did not evaluate the impact of this technique on performance. Espasa explored a vector architecture with out-of-order issue capabilities that uses its re-order buffer to support precise exceptions for all error sources including virtual memory [Esp97, EVS97]. This processor preserves the old values of any vector registers modified by arithmetic or memory instructions until their execution completes without faults. Espasa demonstrated that doubling of the number of physical vector registers available in this design was necessary in order to limit the performance overhead of precise exceptions support to 5%.

7.6 Summary

In this chapter, we presented at set of complimentary architectural and microarchitectural techniques for efficient implementation of precise virtual memory exceptions in vector microprocessors. The architectural enhancement involves modifying the definition of precise exceptions to allow partial completion of the faulting instruction. The microarchitectural technique involves the use of local vector registers in the vector cores in CODE to maintain the old values for operands of vector instructions until we know that any preceding loads and stores will complete without exceptions. We can implement this technique with a simple extension to the VIL block without affecting the structure or operation of the vector cores.

We demonstrated that the performance impact of precise exceptions support is less than 15% for CODE implementations with 4 local vector registers per core. With 8 local vector registers or more, there is no noticeable performance loss due to the support for precise virtual memory exceptions.

Chapter 8

Embedded Memory System Architecture

"The advantage of a bad memory is that someone enjoys several times the same good things for the first time."
Friedrich Nietzsche

As the computing power of microprocessor chips improves at exponential rates, the lagging performance of memory systems becomes a serious bottleneck [HP02, SN96]. As a result, modern microprocessors devote an increasing percentage of their die area to techniques that improve memory latency and bandwidth, such as multi-level caches, prefetching mechanisms, and stream buffers [PK94, CB94]. However, caches are not the obvious remedy for the memory performance problems in a vector processor for embedded multimedia applications. Video and sound processing kernels do not always exhibit high degrees of temporal locality in their data references. In addition, the cost of large, multi-level caches is often too high for embedded applications. This chapter explores the design of high performance memory systems for vector microprocessors based on embedded DRAM technology. Even though embedded DRAM is not the only alternative for this purpose, it has the potential to provide high memory bandwidth in a cost-efficient and highly integrated manner.

Section 8.1 explains the memory performance issues in vector microprocessors. Section 8.2 introduces embedded DRAM technology, along with its major benefits and challenges. Section 8.3 presents the design options available to architects of embedded DRAM memory systems for vector microprocessors and discusses how they affect efficiency in terms of latency, bandwidth, energy consumption, and cost. Section 8.4 continues with a quantitative evaluation of embedded DRAM memory systems for the CODE vector microarchitecture. Finally, Section 8.5 reviews related work in memory systems for vector processors.

8.1 Memory System Background

Figure 8.1 presents the processor-memory performance gap, the fundamental reason for which main memory is a serious performance bottleneck for modern computer systems [HP02]. Microprocessor architectures and manufacturing processes have been optimized for maximum performance. On the other hand, the organizations and manufacturing processes for DRAM, the basic technology for main memory, have been optimized for minimum cost and maximum capacity [Prz94, Pri96]. The consequence is a gap between processor performance and memory latency that grows each year at an exponential rate. A main memory access that took one processor cycle in 1980 requires several hundreds of cycles in 2002. Regardless of the degree of decoupling or out-of-order

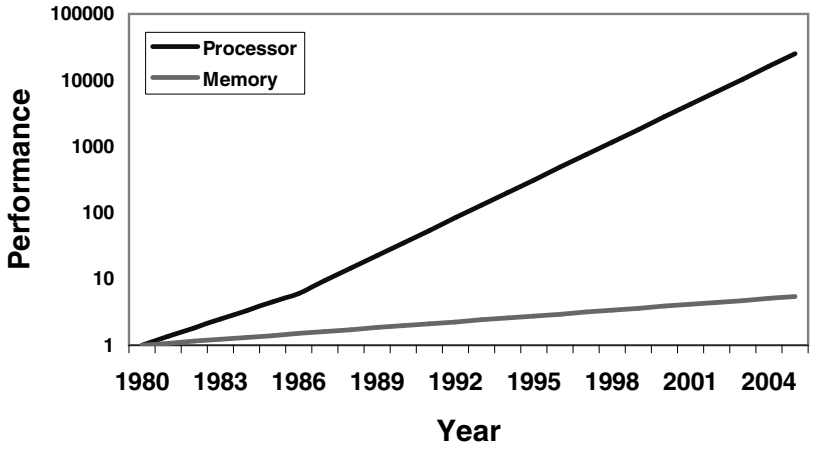


Figure 8.1: The processor-memory performance gap [HP02]. Using the 1980 state-of-the art as the baseline, the processor performance has been improving by 35% per year until 1986 and by 55% per year thereafter. On the other hand, the access latency of DRAM main memory has been improving by merely 7% per year. Note that the performance axis uses a logarithmic scale to capture the size of the gap between processor and memory.

execution available in the processor, it is impossible to tolerate this kind of memory latency without a devastating reduction in performance due to data dependencies and the associated stalls.

The mainstream solution to the problem of memory latency has been hierarchies of caches [Smi82]. Caches exploit the temporal and spatial locality in the address stream of applications in order to maintain frequently accessed data in small SRAM structures close to the processor that allow for quick references. Virtually every processor chip in 2002 includes at least one level of instruction and data caches. In most cases, caches occupy at least 50% of the processor die. For high-end processors that integrate multiple levels of high speed, high capacity SRAM arrays, caches occupy as much as 80% of the die area [Gre00].

The memory performance problem is slightly different for vector processors. They can exploit the well-known memory access pattern of vector instructions to (pre)fetch multiple vector elements per memory access, which allows them to tolerate relatively higher memory latencies than scalar processors. However, the consequence is that high memory bandwidth becomes an important requirement. Caches can easily support low latency in the presence of access locality, but cannot provide high bandwidth for strided and indexed accesses without using expensive, multi-ported organizations. Several studies have demonstrated that either caches provide limited performance gains for vector processors or extremely expensive cache organizations are necessary for consistent performance improvements [KSF+94, FP91, HS93, GS92]. Consequently, the memory system for vector supercomputers typically consists of hundreds of SRAM chips, organized in a multi-bank memory system [ABHS89, Cra93]. Even though this approach eliminates the problem of DRAM latency, it is prohibitively expensive for vector processors for embedded systems in terms of manufacturing cost, system size, and power consumption.

The streaming nature of multimedia data is another reason caches are not appropriate for an embedded media-processor. Real-time applications typically process each video frame or audio sample just once before discarding it for the next set of inputs [DD97, CDJ+97]. Hence, temporal locality, a basic premise for the success of caching, is significantly limited. The effect of streaming accesses has motivated the inclusion of special hardware in several commercial processors that allows for bypassing the caches and eliminating the power consumption and lookup latency they introduce for streaming data. A vector processor further reduces the usefulness of caches with multimedia

programs, because vector registers capture a large degree of the spatial locality available. However, we should note that some classes of input data to multimedia programs are amenable to caching. Examples include the dictionary data in speech recognition applications and the texture information in 3-D graphics pipelines.

In summary, to match the requirements of a vector microprocessor for embedded multimedia applications a memory system must have the following characteristics:

- **High memory bandwidth** for parallel element transfers for sequential, strided, and indexed access patterns.
- **Moderate access latency** for non-vectorizable access patterns or references to short vectors.
- **Low energy consumption**, especially for sequential streams.
- **Low implementation cost** to make the overall system appropriate for consumer products.

Furthermore, we want the memory system to exhibit similar scalability characteristics with the vector processor, for which we can easily scale the performance, energy consumption, and area by allocating the proper number of vector cores or lanes. Ease of scaling allows designers to match the memory system to the specific requirements of the vector processor at the minimum possible cost or complexity.

8.2 Embedded DRAM Technology

Embedded DRAM technology attempts to merge the manufacturing processes for high speed logic circuitry and high capacity DRAM memory. With such a merged process, we can exploit the increasing densities of CMOS chips to integrate on a single die a microprocessor and a memory system large enough to be part of the main memory, not just a cache.

The key benefits of logic-DRAM integration over the traditional approach closely match the requirements for memory systems for embedded vector processors [PAC+97]:

- **Higher bandwidth**: The single-chip integration allows us to eliminate the narrow, off-chip bus between the processor and external memory. Instead, we can use a switched interconnect with thousands of high speed wires that exposes to the processor a large percentage of the bandwidth available at the sense amplifiers of the DRAM arrays.
- **Lower latency**: Servicing data references from the on-chip DRAM without accessing the off-chip memory effectively halves the access latency. The integration also enables the customization of the on-chip DRAM organization. The use of multiple independent memory modules built from small arrays of DRAM cells reduces the access latency within each module and allows overlapping of independent accesses across modules.
- **Lower energy consumption**: The elimination of the off-chip memory bus with its high capacitance board traces also reduces the energy consumption for memory transfers [FPC+97]. Further reductions are possible by activating only the corresponding module in the on-chip memory for each access. In addition, we can exploit the high bandwidth, low latency properties of embedded DRAM to eliminate SRAM caches and save the energy consumed for cache lookups.
- **Reduced system size and cost**: Even with 0.18μm CMOS technology, a chip of reasonable size for commercial use ($\leq 200mm^2$) can integrate a processor core of significant complexity with over 10MBytes of embedded DRAM [SK99]. For several consumer products, this on-chip memory capacity is sufficient and no external memory chips, boards, or controllers are

Basic Technology Parameters	
Feature size	0.18μm bulk CMOS process with dual gate oxide
Interconnect	6 copper layers
Power supply	1.8V (3.3V, 2.5V, or 1.5 V for IO)
Gate delay	33psec (NAND-2, nominal conditions)
Power dissipation	0.02 μW/MHz/gate
Embedded DRAM Parameters	
Capacitor type	Burried-trench capacitor
DRAM cell size	0.56μm^2
Refresh rate	0.4μsec
DRAM modules	1 Mbit to 16 Mbit with 1 Mbit minimum increment 2 Kbits per row in each 1 Mbit sub-array
Module interface	256 bits input, 256 bits output separate address & control lines
Module testing	Integrated BIST engine and test interface
Random access latency	20nsec
Page mode access latency	6.6nsec
Maximum bandwidth	4.8GBytes/sec/module

Table 8.1: The basic parameters of the IBM SA27E CMOS process for embedded DRAM technology. In 2002, SA27E represents a mature embedded DRAM process for mainstream commercial development. SA27E was the implementation technology for the VIRAM-1 prototype chip.

necessary. Although DRAM chips are cheaper per megabyte than embedded DRAM, the elimination of external chips and boards leads to significant savings in system size, design complexity, and development cost. In addition, with each technology shrink, the capacity of on-chip memory will increase and will become sufficient for a larger number of embedded applications.

We can develop an embedded DRAM process starting from either a DRAM or a logic manufacturing process. The use of a DRAM process allows for the highest density for DRAM cells but creates significant challenges to implementing high-speed logic. To minimize manufacturing costs and power consumption, a DRAM process has transistor devices with long channels, thick insulation layers, and high threshold voltages, which all hurt transistor speed. In addition, the dynamic storage node in DRAM cells is typically a "stack capacitor" structure [Pri96], which rises above the transistors and complicates the implementation of multiple levels of wiring. On the other hand, a logic process combined with a "buried-trench capacitor" structure provides a better starting point [SK99]. The capacitor extends bellow all other devices and creates no problem with implementing fast transistors and multiple wiring levels using the same manufacturing steps with a pure logic process.

Table 8.1 summarizes the features of the IBM SA27E CMOS process as an example of a mature embedded DRAM technology based on a logic process. It supports the same level of logic performance as a pure logic process for the same feature size. Designers assemble on-chip memory systems in SA27E by allocating pre-designed DRAM modules. The mix of modules in terms of the number and their size is a design-time option. For example, the VIRAM-1 chip includes eight 13-Mbit modules for a total memory capacity of 13 MBytes. This design methodology is extremely flexible and allows designers to customize the memory system to the requirements of the processor core on the die. The maximum bandwidth from each DRAM module is 4.8GBytes/sec, which is equal to the peak bandwidth of three independent Rambus channels running at 800MHz [Cri97].

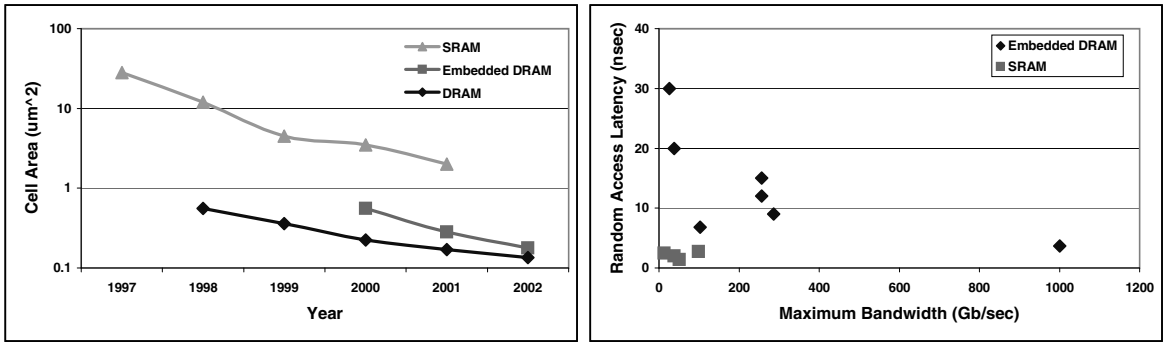


Figure 8.2: The evolution of the cell area, random access latency, and maximum bandwidth for embedded DRAM technology. The chart on the left compares the embedded DRAM cell size with that for on-chip SRAM and conventional DRAM. All data points refer to memory arrays available from IBM. The cell area axis uses a logarithmic scale. The chart on the right compares the latency and bandwidth of DRAM modules to that of SRAM modules used in the cache system of high-end processors. The data sources are papers presented in the International Solid-State Circuits Conference in 1998 [IEE98], 1999 [IEE99], and 2000 [IEE00]. Some of the slower DRAM macros use 0.25μm or 0.35μm CMOS technology. All SRAM macros use 0.18μm technology.

The typical criticism for embedded DRAM technology is its efficiency in terms of cell area and manufacturing cost when compared to off-chip DRAM and on-chip SRAM, respectively. Figure 8.2 presents the evolution of cell area for the SRAM, DRAM, and embedded DRAM technologies available by IBM. Embedded DRAM cells are approximately two times bigger than cells in stand-alone DRAM chips because they are optimized for low access latency and robust operation in the same die with high-speed logic circuitry. However, embedded DRAM is still 6 to 10 times denser than SRAM. Embedded DRAM memory systems can be large enough for use as main memory, while on-chip SRAM can only provide sufficient capacity for caches. Figure 8.2 also shows that the initial embedded DRAM modules available between 1998 and 2000 provided higher bandwidth than SRAM modules for caches for high-end processors, despite the higher random access latency. Second generation embedded DRAM modules have managed to reduce the latency gap with SRAM to a factor of two, while offering 50 times higher bandwidth [TD+00].

The manufacturing of storage capacitors adds four extra lithography masks to an embedded DRAM process like SA27E. Along with the time necessary for DRAM testing and repair, this translates to approximately 30% higher cost per wafer. However, the overall cost for a system using an embedded DRAM chip can be lower than that for a conventional system. The high capacity of embedded DRAM enables designers to eliminate partially or completely the external memory system along with the associate costs for the development or purchase of its components (boards, DIMMS, and chipsets). In other cases, designers may be able to reduce the chip area by replacing the on-chip SRAM memory with embedded DRAM that has the same capacity but occupies one sixth of the area. In addition, the high bandwidth of embedded DRAM modules eliminates the need for expensive off-chip memory interfaces such as Rambus. Therefore, embedded DRAM technology can provide high bandwidth, integrated memory systems with low manufacturing cost for the overall system.

8.3 Memory System Design Space for Embedded DRAM

Embedded DRAM technology allows for the integration of the main memory system on the same die with the vector processor. Consequently, the chip designers have full control over the

organization and implementation of the memory system and the processor to memory interconnect. Ideally, we want to construct an organization that meets the bandwidth and latency requirements of the processor at the minimum energy and area overhead. This section introduces the most important design parameters of the on-chip memory system and discusses how they affect its performance (bandwidth and latency), energy consumption, and cost. Section 8.4 provides a quantitative analysis for some of these options within the framework of the CODE microarchitecture.

Even though the discussion focuses on meeting the requirements of a vector processor, we can extend the conclusions to any other processor architecture that relies on high bandwidth memory systems, such as single-chip multiprocessors or multithreaded systems.

8.3.1 Memory Banks and Sub-banks

To simplify the design of memory systems and hide the complexity of assembling arrays of DRAM cells, semiconductor vendors either provide a selection of predesigned DRAM modules or support software that generates DRAM module configurations within certain constraints. Hence, we can construct the memory system as a collection of independent memory banks, where each bank is DRAM module with separate interface and control logic.

There are several advantages to a multi-bank memory system. Memory transfers that map to different banks can execute concurrently, which allows the memory system to fetch in parallel multiple elements for indexed and strided operations or overlap vector accesses with requests from the scalar core and the IO system. Since each access involves circuitry in just one memory bank, the access latency and energy consumption is lower in a multi-bank system than in a system organized as a single bank with centralized control logic. Finally, multiple banks introduce a caching effect within DRAM. Each access transfers a row of bits from the array of DRAM cells to the sense-amplifiers of the bank. This data row is called open because the sense-amplifiers can directly serve any subsequent accesses to the data it contains and, therefore, reduce the effective access latency [Prz94]. A multi-bank memory system can have one open row per bank, which increases the probability of low latency for access streams with spatial or temporal locality.

Nevertheless, a vector processor can quickly saturate a multi-bank memory system, especially for applications with indexed and strided accesses. For the VIRAM-1 chip, for example, the vector processor can issue 4 addresses for vector elements per cycle to the 8 banks. Due to the high latency of DRAM for random references, each access occupies a bank for 5 processor cycles. No other access can use the same bank during the 5 cycles, unless it references data from the open row. Hence, a truly random access stream for an indexed load can saturate the memory system in just 2 cycles, and the processor must stall for 3 clock cycles for every 8 element accesses it issues. The frequency of stalls is even higher if many element addresses map to a small subset of banks.

We can increase the degree of concurrency in the memory system by using small DRAM modules in order to implement a large number of memory banks. However, each memory bank introduces a significant area overhead due to the interface, control, and testing circuitry it includes. In SA-27E, the fixed overhead of a bank is equal to the area for 2 Mbits of DRAM. Hence, for a fixed area budget, the effective capacity of the memory system decreases with the number of banks. In addition, each extra bank introduces additional wiring and switching overhead in the processor to memory interconnect.

A cost-effective way to achieve the performance benefits of a large number of banks is to increase the degree of concurrency within each bank. To keep the access latency low, each DRAM module consists of a collection of small memory arrays connected to a common data bus [YHO97]. With the addition of pipeline registers for addresses and control, we can turn each array into a sub-bank that can execute accesses to its local data in a semi-independent manner. Figure 8.3 presents the block diagram of an embedded DRAM bank with multiple sub-banks. We can only issue an access to one sub-bank per clock cycle, but we can overlap multiple accesses that map to different sub-banks in a pipelined fashion. Consequently, sub-banks improve random bandwidth by

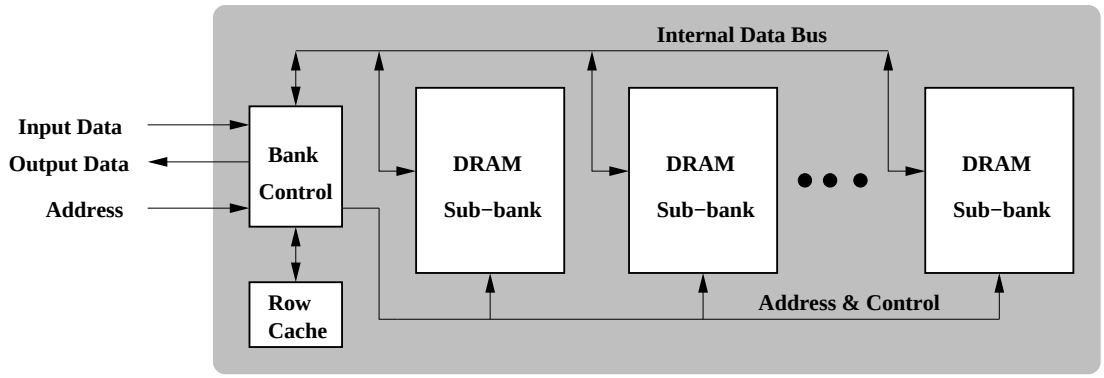


Figure 8.3: The block diagram of an embedded DRAM bank with multiple sub-banks. Each sub-bank includes pipeline registers for address and control that allow overlapped execution of accesses that map to different sub-banks. The row cache is an optional SRAM-based array that caches the most frequently accessed rows (see Section 8.3.3, page 100).

decreasing the frequency of bank conflicts and increasing the number of accesses that can execute concurrently. In addition, each sub-bank has a separate open row, which reduces effective access latency due to caching. The area overhead for introducing 4 to 8 sub-banks in a DRAM module is typically less then 3%.

The appropriate number of banks and sub-banks for a specific vector processor depends on the processor organization and the frequency of indexed and strided operations. A vector processor produces a single address per cycle per load-store unit for unit-stride accesses, but typically has multiple address generators for parallel element transfers for strided and indexed operations. To reduce the frequency of bank conflicts, the number of banks should be approximately two times larger than the average number of addresses issued per cycle by the vector coprocessor, scalar core, and the IO system. For example, VIRAM-1 can issue 4 addresses for vector accesses and 1 address for scalar or IO references per clock cycle. Hence, at least 8 DRAM banks are necessary in the VIRAM-1 memory system to minimize bank conflicts for applications with frequent strided and indexed accesses.

The area overhead of sub-banks is low and there is no reason to have less than 4 sub-banks per bank. In any case, the total number of sub-banks across all the banks in the memory system should be approximately two times larger than the product of the random access latency and the average number of addresses issued per cycle. This number of sub-bank reduces the probability that a strided or indexed stream will access a second row in some sub-bank, while it is busy servicing an previous access to another row. For example, VIRAM-1 can issue up to 5 addresses per cycle (4 vector, 1 scalar or IO) and the random access latency is 5 processor cycles. Hence, a total of 25 sub-banks are necessary, which translates to approximately 4 sub-banks in each of the 8 DRAM banks.

We should stress again that the above recommendations for selecting the optimal number of banks and sub-bank assume frequent use a strided and indexed accesses. If the applications use mostly unit stride accesses, one should consider using a smaller number of DRAM banks in order to reduce the area overhead and cost of the memory system. We should also note that the selection of predesigned modules or the capabilities of the module generator provided by the process vendor may significantly limit the number of memory configurations that are practical.

8.3.2 Basic Bank Configuration

Regardless of the number of sub-banks, there are two design parameters for the DRAM bank that affect its performance and energy consumption: the width of rows and the width of columns.

A DRAM sub-bank is a rectangular array of memory cells with capacity of 32 Kbits to 1 Mbit, depending on the size of the sub-bank. In the case of the IBM SA-27E technology, the 1 Mbit array consists of 512 rows with 2048 memory cells per row. On every memory reference, we transfer a whole row of data bits to the sense amplifiers of the array, even if the reference accesses only a few bytes from the row. Subsequent references to data in the same row can execute faster because they can skip the access to the DRAM array and they can read data directly from the sense amplifiers. Hence, wide rows in DRAM arrays effectively implement prefetching for sequential accesses. Only the few references that stretch to a new row will incur the additional latency for accessing the DRAM array. On the other hand, if the access stream has little spatial locality, accesses to wide rows are wasteful in terms of energy. For every memory reference, we transfer a large number of bits from the array to its sense amplifiers, but only use a small fraction of them in the application. In addition, wide rows and narrow data accesses complicate the implementation of error correction codes (ECC).

The maximum number of bits from a sub-bank row that we can transfer to the interface of the DRAM bank and subsequently to the processor is called the column width. In practice, the column width is set by the width of the internal data bus that connects the sub-banks within a bank (see Figure 8.3). In the case of the SA27E technology by IBM, the column width is 256 bits, one eighth of a row. Wide columns implement prefetching for unit stride references as well, since they allow the processor to access a large number of sequential elements in a single cycle. For address streams with little spatial locality, however, very wide columns waste energy by transferring unnecessary data. In addition, they complicate the design of the memory system because they require a wide internal data bus within each bank, a wide bank interface, and a large number of wires in the processor to memory interconnect.

A designer should select the row and column widths to optimize the latency and energy consumption for unit stride references, the undoubtedly most common access pattern in vector processors (see Table 4.4, page 31). To match the bandwidth of a load-store unit to the throughput of the arithmetic units, the column width should be equal to the data width in the functional units times the number of lanes in the vector processor. For example, VIRAM-1 includes 4 lanes with one 64-bit datapath for each arithmetic unit per lane, hence the column width should be at least 256 bits. This column width allows the load-store unit to produce with a single bank access as many sequential elements as the functional units. The row size should be at least as long as the average vector length times the element size in memory. For example, applications that run on VIRAM-1 frequently load a full vector (128 elements for 16-bit VPW) with 16-bit data from memory. Hence, setting the row width to 2048 bits allows unit stride load and store instructions to experience the latency and energy overhead of a row access only once on the average.

We should note here that there may be electrical reasons that limit the practical values for the two parameters or that the process vendor may fix them both in the modules for a certain embedded DRAM technology. The latter is the case for the SA-27E technology that we used for the implementation of VIRAM-1. However, we engineered the vector coprocessor so that the optimal row and column widths match the settings in the DRAM modules of SA-27E.

8.3.3 Caching in the DRAM Memory System

Several studies have concluded that the performance benefit from the use of traditional cache hierarchies with vector processors is not proportional to the cost of implementing large SRAM arrays [KSF+94, FP91, GS92]. However, we can improve the performance and energy efficiency of the embedded DRAM memory system by including some amount of caching in each DRAM bank.

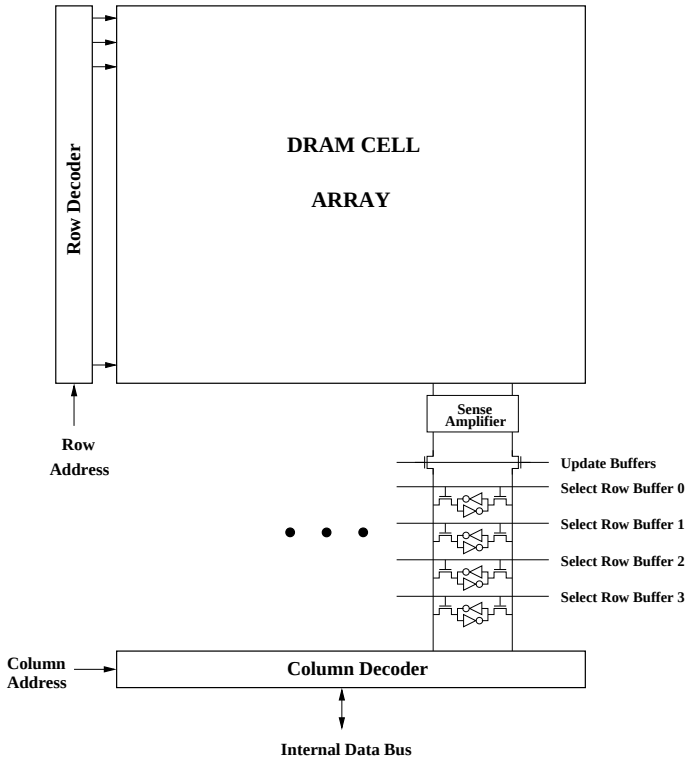


Figure 8.4: The block diagram of a DRAM sub-bank with four row buffers. The buffers are arrays of static memory cells that connect to the output of the sense amplifiers of the DRAM array. Each buffer can store a full DRAM row. A set of select lines controls the pass-transistors that connect the row buffers to the column decoder. The column decoder selects the subset of the bits in a row that will be transferred to the external interface of the DRAM bank. The pass-transistors between the sense amplifiers and the row buffers provide isolation and allow accesses to the buffers while another row is read, written, or refreshed in the DRAM array using the sense amplifiers.

The basic idea is to introduce some SRAM storage array or buffers that cache a large number of frequently accessed DRAM rows and allow read and write references without incurring the latency and energy overhead of retrieving a row from a large array of DRAM cells.

There are two alternative approaches to adding caches to DRAM banks. Figure 8.3 (page 99) presents the first alternative, in which we introduce a row cache close to the bank interface. Each line in the cache can store a DRAM row. Figure 8.4 presents the second approach, in which we introduce a number of row buffers at the sense amplifiers in each sub-bank. The row cache requires no significant changes to the layout of DRAM sub-banks. For accesses that hit in the cache, the row cache also eliminates the latency and energy consumption of the internal data bus that connects sub-banks. However, the internal data bus must be wide enough to facilitate the transfer of a whole row from the sub-banks to the row cache within a few clock cycles (one to four). The row buffers require modifications to the basic layout of the DRAM sub-banks and do not eliminate the latency and energy overhead of the internal data bus for accesses to cached data. However, row buffers constitute a distributed cache across the DRAM sub-banks, which we can access with minimum energy overhead. In contrast, the energy required to access a row cache with a large number of lines may be equal or even larger than the energy saved by eliminating the DRAM array access and the transfer on the internal data bus.

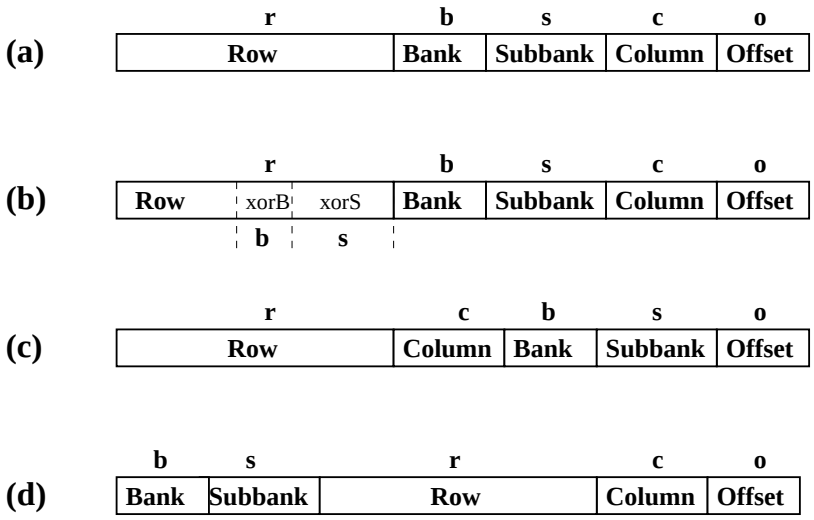


Figure 8.5: Four simple address interleaving schemes for an embedded DRAM memory system. The figure assumes that the system has 2^b banks, 2^s sub-banks per bank, 2^r rows per sub-bank, 2^c columns per bank, and 2^o bytes per column. Each scheme presents the location of the bank, sub-bank, row, column, and offset fields within a memory address. The most significant bit of the address is the leftmost one. The VIRAM-1 prototype chip implements all four schemes with s=0. One can construct several variations or combinations of the four schemes to address specific performance or design issues in the memory system.

Both forms of caching require some control logic and a set of tags that determine which rows should be cached and when an incoming access hits in the cache. The trade-offs for replacement (LRU, MRU, random) and allocation policies (write allocate, write no-allocate) in cached DRAM are the same as with traditional caches [HP02]. We can fully implement cache tags and control within the DRAM bank with no modification to the rest of the memory system. Alternatively, we can expose the existence of row caches or buffers to the processor and allow the load-store units to determine the optimal caching policies based on the access characteristics of the applications. Commercially available DRAM chips with integrated row caches like the NEC virtual-channel DRAM implement the latter approach [MS+99].

8.3.4 Address Interleaving

The address interleaving scheme indicates the mapping between the address for an access and a specific storage location in the memory system. In other words, it specifies the correspondence between address bits and a set of numbers that identify the bank, the sub-bank, the row, the column, and the column offset for the storage location. The choice of interleaving scheme has little impact on the complexity of the circuitry required to implement it. Nevertheless, it has a significant effect on performance and energy consumption as it determines the frequency of bank or sub-bank conflicts and the probability of accessing data from open rows for a given address stream.

Figure 8.5 presents four simple interleaving schemes for an embedded DRAM memory system. With scheme (a), a sequential address stream accesses a full row of data in each sub-bank in every bank before visiting any sub-bank for the second time. We can retrieve multiple consecutive vector elements with a single column access and use all data in an open row. Hence, scheme (a) leads to maximum performance and energy efficiency for sequential accesses. However, scheme (a) is inefficient for strided streams with very large strides because two addresses for consecutive elements

will probably map to different rows in the same sub-bank. Therefore, the two accesses cannot execute in parallel or overlap. Large strides can be frequent in applications that operate on two-dimensional arrays or image processing programs that use outer-loop vectorization. Scheme (b) alleviates the inefficiency for strided accesses by performing an exclusive or operation between the bank and sub-bank fields and fields `xorB` and `xorS` from the row number respectively, in order to determine the exact bank and sub-bank to access. Consequently, two addresses that differ only in the low order bits of the row field will map to different banks or sub-banks and can execute concurrently.

Interleaving scheme (c) in Figure 8.5 is appropriate for address streams with small strides or indexed accesses to a small data structure. The addresses for two consecutive elements will map either to the same column or to different banks and sub-banks. Small strides appear in applications that handle color images, where the red, green, and blue components of each pixel require separate handling.

Finally, scheme (d) accesses all bits within a sub-bank before moving to the next one for a sequential access stream. In terms of performance, it is beneficial to applications with a single unit-stride access stream or to programs with huge strides. The latter case occurs rarely in multimedia applications. However, scheme (d) allows the use of a chip with non-repairable faults in its memory system. We can disable a whole sub-bank or bank by mapping out a large consecutive region of the physical address space, which is easy to handle at the operating system level.

The implementation of one interleaving scheme does not prohibit the use of any others. The VIRAM-1 chip implements all the schemes in Figure 8.5. Since the 24 least significant bits of the address are sufficient to address the 13 MBytes of on-chip memory, the two most significant bits from the 32-bit physical address select the interleaving scheme. Schemes (a), (b), and (c) define the bank and sub-bank number in the least significant portion of the address which does not change during virtual to physical address translation. Hence, an application can use all three schemes by selecting one scheme for each memory page. Pages with data accessed mostly with unit-stride instructions can use scheme (a), while pages with frequent strided accesses can use scheme (b) or (c) depending on the stride. Scheme (d) identifies the bank and sub-bank numbers in the most significant bits, which change during address translation. To guarantee correct operation in the presence of memory aliases, an application that runs with virtual memory enabled must use scheme (d) for all data and code accesses or not at all.

8.3.5 Memory to Processor Interconnect

The memory to processor interconnect is an important component of the memory system. The fact that the memory banks can execute in parallel or overlap a large number of sequential or random accesses is of little importance if the interconnect does not have matching capabilities in terms of bandwidth.

The memory interconnect transfers three types of information: addresses from the processor to the memory banks, store data going in the same direction, and load data from the memory banks to the processor or the IO system. Since the cost of wires reduces dramatically once the memory system becomes part of the same die with the processor, it is preferable to use a separate interconnect structure for each type. Separate paths for addresses, load data, and store data increase the degree of concurrency of transfers, require simpler control, and have lower energy consumption due to the elimination of multiplexing of independent sources of information on the same set of wires.

The structure of the memory interconnect depends on the exact organization of the vector processor and the memory system. To support multiple independent accesses per cycle from the vector processor, a form of crossbar for both data and addresses is necessary. The degree of parallelism in the crossbar depends only on the number of addresses that the processor can issue per cycle and not on the number of banks or sub-banks in the memory system. An over-clocked bus or a token-ring structure can provide cheaper alternatives to the crossbar. However, the over-clocked

bus introduces some complexity to handle the different clocks and the token-ring has higher latency than the crossbar, unless we over-clock it as well.

Overall, the integration of the processor and the memory system on the same die and the ability of a vector processor to tolerate moderate latencies allow the use of several alternative structures for the memory interconnect, including hybrid, ad-hoc, and pipelined schemes. For example, the VIRAM-1 chip separates the eight memory banks in two groups of four and provides a crossbar structure for each group. In any case, it is wise to use wide paths for data transfers, since this is a cheap way to increase sequential bandwidth in an on-chip network. We can improve random bandwidth by allowing multiple narrower transfers to use the same wide path if necessary. In addition, the use of an interconnect structure with uniform latencies and conflict characteristics to all memory banks simplifies significantly the design of its control and conflict resolution logic.

8.4 Memory System Evaluation

In this section, we use the EEMBC multimedia benchmarks to explore the interaction of the CODE vector microarchitecture and memory systems based on embedded DRAM technology. We focus on the two most important issues: the effect of memory latency and the optimal number of DRAM banks and sub-banks. The other design options discussed in Section 8.3 have little impact on the behavior of the EEMBC benchmarks. However, they can be significant for other applications that make frequent use of large strides or use larger data sets.

8.4.1 Effect of Memory Latency

Several factors can lead to increased memory latency for vector element accesses in CODE: the use of DRAM banks with slow timing parameters, bank and sub-bank conflicts, the use of a processor to memory interconnect structure with high latency, or clocking the vector processor significantly faster than the memory system. To explore the effectiveness of the microarchitecture with tolerating high memory latency, we evaluate a CODE configuration with a memory system that offers infinite bandwidth at fixed latency. This memory system never stalls an element access and always returns the load data after a specific number of cycles. We vary the memory latency from 1 to 128 processor cycles in order to capture the behavior of both on-chip memory systems (latency ≤ 32) and off-chip memory systems (latency > 32). For example, the latency of the on-chip memory system in VIRAM-1 is 8 processor cycles. It includes the address propagation, the memory access, the data crossbar, and operations for sign extension and alignment. However, bank conflicts during indexed or strided operations can increase the effective latency by up to three times.

Figures 8.6 and 8.7 present the effect of memory latency on the execution time of the EEMBC benchmarks. For 8 out of 10 benchmarks, increasing memory latency from 1 to 32 cycles leads to less than 15% increase in execution time. CODE is able to initiate most vector load instructions early enough so that only a small portion of the memory latency is exposed to dependent arithmetic instructions. This is an additional advantage to the inherent ability of vector instructions to tolerate latency by amortizing the cost of initiating a memory access over a large number of element transfers.

`Cjpeg` and `Viterbi` experience a higher increase in execution time (40%) for memory latency of 32 cycles. Both benchmarks have a low ratio of arithmetic to memory operations, which limits the ability of CODE to issue load instructions early in order to prefetch their data. In addition, their code frequently requires that all input data are loaded at the very beginning of each loop iteration. Furthermore, `Cjpeg` and `Viterbi` operate mostly on short vectors with 13 to 18 elements.

Overall, Figures 8.6 and 8.7 show that execution time increases slowly with memory latency. Even with latency of 128 cycles, which occurs when a multi-gigahertz vector processor uses an off-

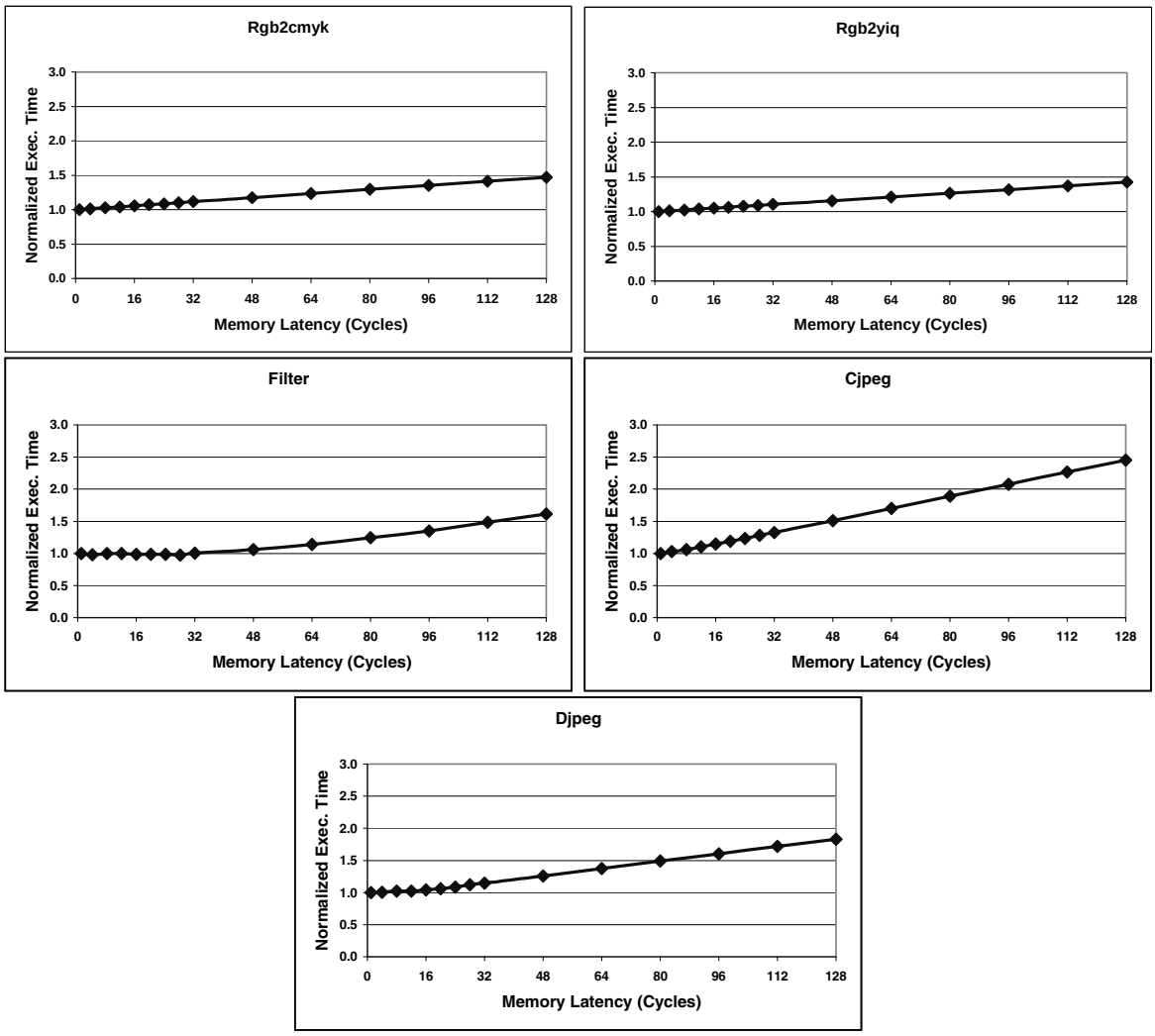


Figure 8.6: The effect of memory latency on the execution time of the consumer benchmarks on CODE. Memory latency is measured in processor cycles. It includes the delay of the DRAM bank access and the latency of the processor to memory interconnect. The execution time is normalized to that with memory latency of 1 cycle. Larger execution time implies lower performance. The simulated CODE configuration has similar hardware resources to a single lane VIRAM-1 chip (see Table 7.1, page 89) and includes $r = 8$ vector registers per execution or load-store core.

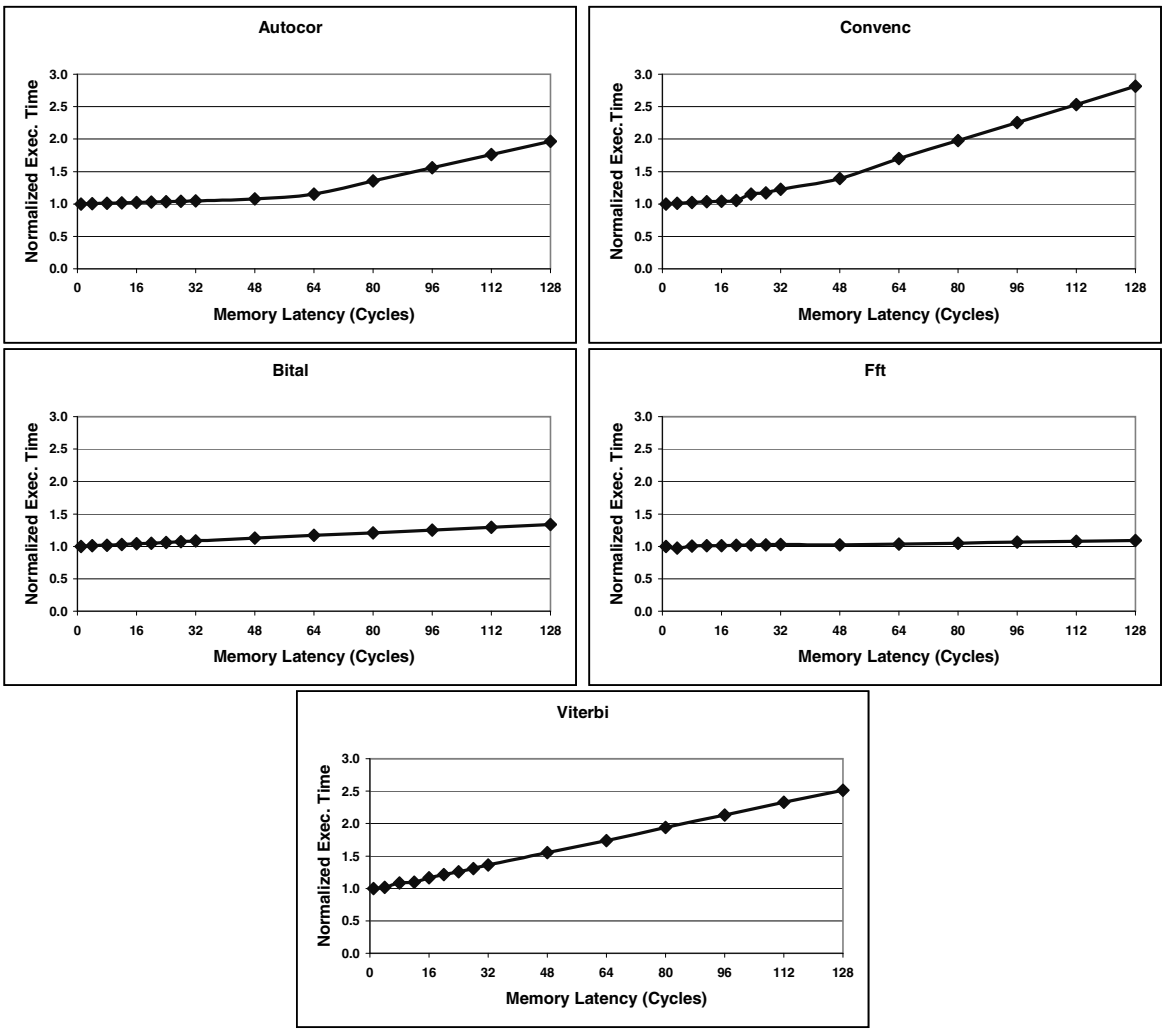


Figure 8.7: The effect of memory latency on the execution time of the telecommunications benchmarks on CODE. Memory latency is measured in processor cycles. It includes the delay of the DRAM bank access and the latency of the processor to memory interconnect. The execution time is normalized to that with memory latency of 1 cycle. Larger execution time implies lower performance. The simulated CODE configuration has similar hardware resources to a single lane VIRAM-1 chip (see Table 7.1, page 89) and includes $r = 8$ vector registers per execution or load-store core.

chip DRAM memory system, the execution time is only 1.5 to 2.8 times higher than that with single cycle memory latency. Despite the lack of SRAM caches, decoupling allows CODE to hide most the memory latency from both on-chip and off-chip DRAM systems, assuming sufficient bandwidth is available.

We can draw several interesting conclusions from Figures 8.6 and 8.7 about the design options in the memory system of a CODE microprocessor:

- The processor to memory interconnect in CODE must provide high bandwidth but not necessarily low latency. Hence, a single cycle crossbar switch, like the one in VIRAM-1, is not the only implementation option. Simple, pipelined structures, such as a circular ring, are viable alternatives as long as they provide sufficient bandwidth for load and store data. For example, replacing a single cycle crossbar with a ring with 4 to 8 cycles of propagation latency will slowdown most benchmarks by less than 5%.

- We can clock the vector processor significantly faster than the embedded DRAM banks. Even though this increases the effective latency of element accesses in terms of processor cycles, decoupling can hide most of this latency and allow the performance improvement to closely match the clock frequency of the processor. However, one should keep in mind that an increase in the operating frequency of the vector processor leads to a proportional increase in its power consumption.

- It is not prohibitive in terms of performance to use CODE with an off-chip, high bandwidth memory system. Replacing an on-chip memory system with an 8-cycle latency with external memory with 32 to 64 cycles latency (128 to 320 nsec for a 200 MHz CODE chip) leads to a performance loss between 25% and 50%. Of course, supporting high memory bandwidth with an external memory can be more expensive and less energy efficient. It typically requires the use of advanced memory technology such as DDR or Rambus, as well as hundreds of pins on the processor chip.

8.4.2 Number of DRAM Banks and Sub-banks

Figures 8.8 and 8.9 present the average memory latency for an element access as a function of the number of banks and sub-banks per bank in the on-chip memory system of a CODE implementation. Ideally, we always perform column accesses to open rows, which lead to the minimum memory latency of 1 processor cycle at minimum energy consumption. However, bank or sub-bank conflicts can introduce memory stalls and access serialization that increase latency and reduce effective bandwidth. They also lead to higher energy consumption as they reduce the number of element transfers that require only a column access to an open DRAM row.

Figures 8.8 and 8.9 show that for applications with unit stride accesses (`Filter`, `Autocor`, `Convenc`, and `Bital`) 2 DRAM banks or 4 sub-banks in one bank are sufficient to eliminate most conflicts and reduce memory latency to the minimum possible. For benchmarks with strides of three to four bytes (`Rgb2cmyk` and `Rgb2yiq`) both 2 DRAM banks and 4 sub-banks per bank are necessary for the same purpose. The exact value of the minimum memory latency for each benchmark depends on the number of distinct DRAM rows it accesses and the amount of data used from each row.

For benchmarks with larger strides (`Cjpeg`, `Djpeg`, and `Viterbi`) or indexed accesses (`Fft`), memory latency keeps decreasing as we introduce more DRAM banks or sub-banks per bank. It is interesting to notice that we get approximately the same improvement by doubling the number of banks or the number of sub-banks per bank in each case. Since the area overhead of sub-banks is smaller than that of banks, we should always start by introducing additional sub-banks when trying to improve the performance of the memory system.

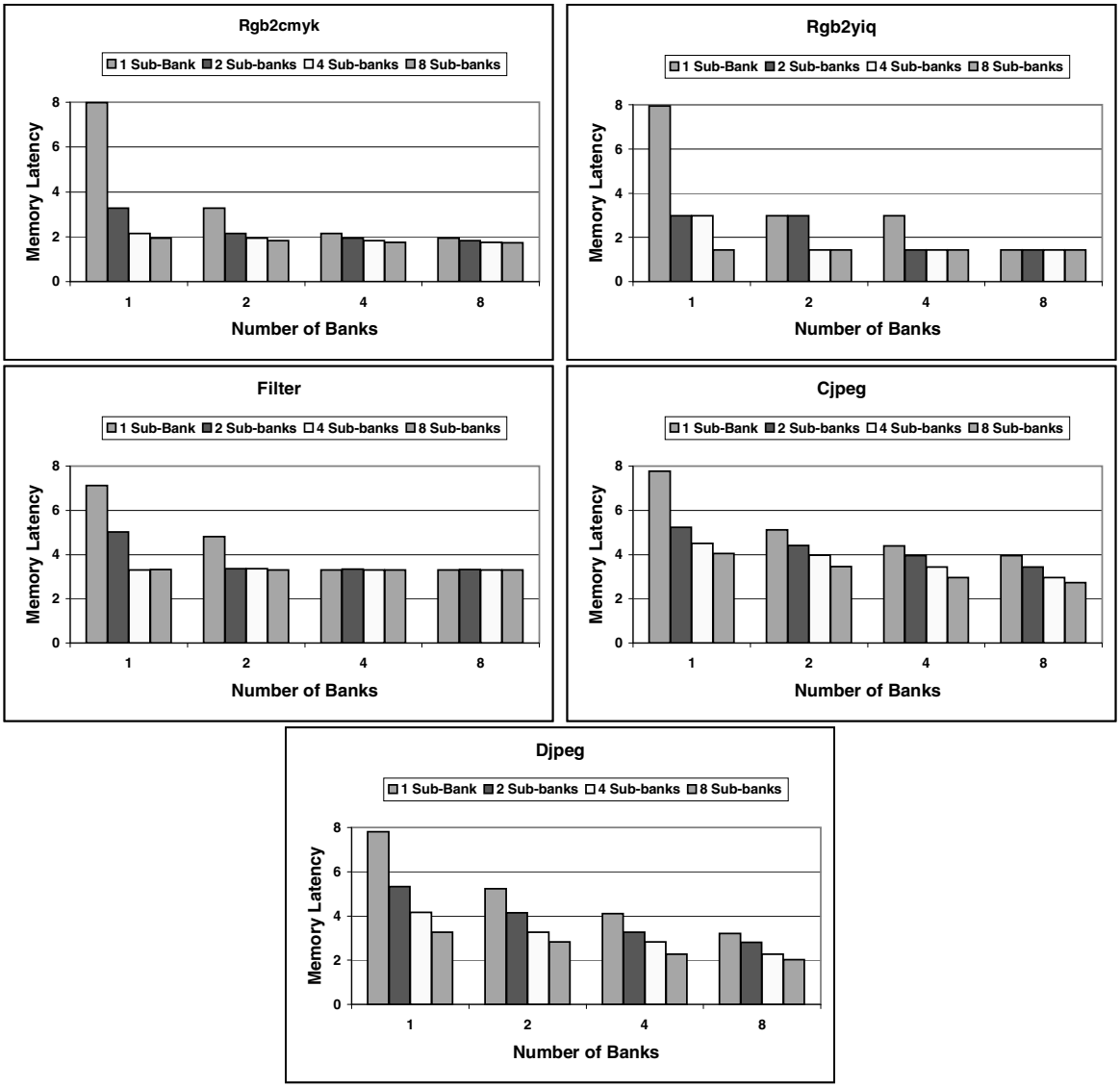


Figure 8.8: The average memory latency for an element access in the consumer benchmarks as a function of the number of DRAM banks and sub-banks per bank. The latency is in processor cycles and does not include the processor to memory interconnect. We assume that the timing characteristics of DRAM are similar to those in the IBM SA27E technology used in the VIRAM-1 chip: 5 processor cycles for a random access and 1 cycle for a column access to an open row. The simulated CODE configuration is described in Table 7.1 (page 89).

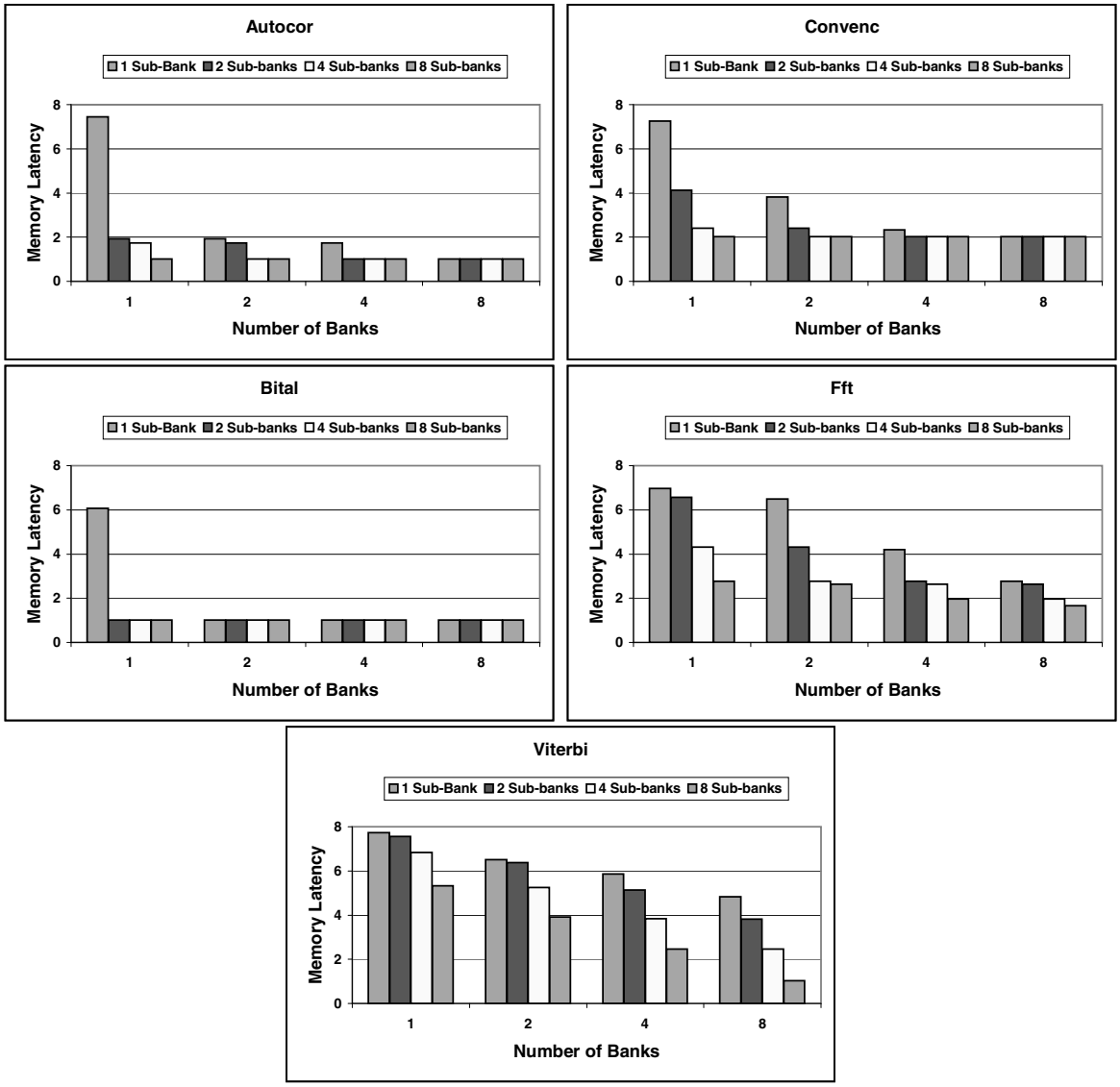


Figure 8.9: The average memory latency for an element access in the telecommunications benchmarks as a function of the number of DRAM banks and sub-banks per bank. The latency is in processor cycles and does not include the processor to memory interconnect. We assume that the timing characteristics of DRAM are similar to those in the IBM SA27E technology used in the VIRAM-1 chip: 5 processor cycles for a random access and 1 cycle for a column access to an open row. The simulated CODE configuration is described in Table 7.1 (page 89).

8.5 Related Work

The existence of strong industrial standards and practices for the interface, form-factor, and overall operation of off-chip DRAM has been a significant obstacle to major innovation in main memory systems [PAC+97]. Consequently, the vast majority of research in the last decade has focused on cache hierarchies and mechanisms for prefetching. However, embedded DRAM technology gives processor designers full control of the main memory system organization and enables the use of techniques for improving its bandwidth and latency characteristics.

Vector supercomputers have traditionally relied on multi-bank main memory systems with SRAM chips to achieve high bandwidth at moderate latency. For example, the Cray C90 Y-MP uses a total of 20,000 SRAM chips in its main memory system of 1024 banks [Cra93]. Consequently, the main memory dominates the cost, system size, and power consumption for such supercomputers. Several studies have examined the potential of various cache organizations for vector processors [KSF+94, FP91, GS92]. In some cases, caches allow the reduction by two of the number of banks in the system or the replacement of SRAM chips with DRAM without significant performance reduction. More recently, Espasa studied a victim cache organization that reduces the memory traffic due to register spilling for a vector architecture with few vector registers [Esp97]. Asanovic proposed two "virtual processor" caches for histogram and rake access patterns in vector processors but provided no data on their overall efficiency [Asa98].

DRAM chips that integrate some amount of SRAM caching have been commercially available for nearly a decade [H+90, Jon92, MS+99]. In [HS93], Hsu and Smith evaluated the potential of cached DRAM chips in the main memory system of uniprocessor and multiprocessor vector systems. They concluded that the use of cached DRAM increases the effective bandwidth of the memory system by factors of about two to four when compared to systems using traditional DRAM chips. Several researchers have evaluated the performance benefits of cached DRAM chips in the main memory system or the third-level cache of scalar processors [ZZZ01]. They concluded that cached DRAM can improve main memory performance by up to 30% and that it can replace SRAM in the lowest levels of the cache hierarchy.

Interleaving schemes have also attracted a lot of theoretical and experimental research work as a cost-effective method to reduce bank conflicts and improve the sustained bandwidth for vector processors. The most important classes of interleaving schemes are linear data-skewing [Law75], XOR-based [FJL85], permutation-based [Soh93], and pseudo-random [Rau91]. The interleaving schemes discussed in this chapter belong to the first two classes that have the lowest cost since their implementation requires no arithmetic operations or table lookups.

Several commercial chips for disk drive controllers, graphics accelerators, and networking systems have successfully used embedded DRAM technology. However, the Mitsubishi M32R/D is the only commercial microprocessor that uses embedded DRAM as main memory [Shi98]. Nevertheless, several academic projects have used embedded DRAM technology in prototype chips (Diva [HKK+99], CRAM [ESSC99]) or simulation studies (FlexRam [KHY+99]) for embedded or supercomputing applications.

8.6 Summary

In this chapter, we explored the architecture of the memory system for vector multimedia processors. We focused mostly on embedded DRAM technology which can match the requirements of a vector processor by providing high memory bandwidth, at modest latency, and low energy consumption. We discussed the various design options for memory systems based on embedded DRAM technology and explored the trade-offs between performance and energy consumption.

We demonstrated that the CODE microarchitecture can efficiently hide large memory latencies in the presence of high memory bandwidth. CODE can accommodate an increase in latency

from 8 cycles to 32 with a performance impact of less than 30%. We also established the importance of banks and sub-banks for improving the performance and energy consumption of the embedded DRAM memory systems. A multi-bank memory system with 8 sub-banks per bank can decrease the effective memory latency by up to 7 times for both unit stride and strided benchmarks.

Chapter 9

Performance and Scalability Analysis

"We all agree that your theory is crazy,
but is it crazy enough?"
Niels Bohr

In previous chapters, we introduced the CODE microarchitecture for vector processing (Chapter 6), explored its ability to support precise exceptions (Chapter 7), and studied how it interacts with memory systems based on embedded DRAM technology (Chapter 8). In this chapter, we proceed with a detailed analysis of the performance and scaling potential of CODE.

Section 9.1 presents a quantitative comparison between the CODE to VIRAM-1 microarchitectures. Section 9.2 analyzes the impact of the inter-core communication network on the performance of CODE. Finally, Section 9.3 evaluates the ability of CODE to exploit additional hardware resources in the form of vector cores and vector lanes.

9.1 CODE vs. VIRAM-1

To facilitate a fair comparison between CODE and VIRAM-1, we set up a CODE configuration that occupies approximately the same area with VIRAM-1 for the same number of lanes. Table 9.1 presents the characteristics of the CODE configuration. Even though the distributed vector register file in CODE allows for a higher clock rate than with VIRAM-1, we assume that the two microarchitectures operate at the same clock frequency. We also assume the same memory system for both designs, namely the one used in the VIRAM-1 prototype chip.

Figures 9.1 and 9.2 compare the performance of CODE and VIRAM-1 for the ten EEMBC benchmarks. For all applications excluding `Filter` and `Bital`, CODE exhibits a performance advantage over VIRAM-1. Since the two microarchitectures include the same number of execution datapaths, the performance advantage of CODE is due to decoupling and its effectiveness with hiding the latency of memory accesses. The benefit of CODE is more obvious for benchmarks like `Rgb2cmyk`, for which the delayed pipeline of VIRAM-1 suffers from frequent stalls from strided memory accesses. CODE, on the other hand, is able to issue load instructions for `Rgb2cmyk` early enough and hides most stalls in the memory system from the execution cores. Nevertheless, the performance gain of CODE is smaller with applications like `Autocor`, for which the delayed pipeline of VIRAM-1 is effective in hiding the moderate latency of its embedded DRAM memory system for unit stride accesses.

VIL	Load balancing core selection policy Random register replacement policy
Vector Cores	1 LDFull core ($r = 8$ local vector registers) 1 IntFull core ($r = 8$ local vector registers) 1 IntSimple core ($r = 8$ local vector registers) 1 ArithRest core ($r = 4$ local vector registers) 1 State core ($r = 8$ local vector registers)
Lanes	1, 2, 4, or 8 lanes with 64-bit datapaths
Communication Network	2 64-bit buses per lane
Memory System	13 MBytes in 8 DRAM banks 1 sub-bank per bank crossbar interconnect with 2 cycles latency

Table 9.1: The CODE configuration for comparison with VIRAM-1. For the same number of lanes, this configuration and a VIRAM-1 implementation without floating-point datapaths occupy approximately the same area. We assume that the two microarchitectures operate at the same clock frequency. We also assume that both designs use the VIRAM-1 memory system based on the IBM SA27E embedded DRAM technology. For details about the exact capabilities of the various cores, refer to Table 6.1 (page 65).

As expected, VIRAM-1 outperforms CODE for the `Filter` benchmark. The register access pattern of `Filter` leads to a large number of inter-core register transfers in CODE, which reduces its performance and energy efficiency. `Bital` performs unit stride accesses and series of short reductions. The instruction sequence for a reduction is strictly sequential and cannot benefit from decoupling. It also requires inter-core register transfers between the execution cores for arithmetic operations and the core for permutations, that can generate stalls on the communication network.

It is also interesting to notice in Figures 9.1 and 9.2 how the performance of CODE scales with the number of lanes. When scaling to 2 or 4 lanes, CODE behaves similarly to VIRAM-1. The performance improvement is almost proportional to the number for lanes for applications with long vectors, such as `Rgb2cmyk` and `Rgb2yiq`. The benefit from additional lanes is significantly smaller for applications with short vectors such as `Viterbi` and `Djpeg`. When scaling to 8 lanes, however, arithmetic instructions occupy the execution cores for a small number of cycles, hence there is less time to hide the impact of memory latency or inter-core transfers through decoupling. As a result, CODE performs slightly worse than VIRAM-1 for `Cjpeg` in the case of 8 lanes.

Figure 9.3 presents the composite EEMBC scores for the two microarchitectures when operating at 200 MHz. CODE outperforms VIRAM-1 in both benchmark categories, regardless of the number of lanes. For the consumer benchmarks, the benefit from CODE is higher (20% to 35%) because these benchmarks access large data sets and frequently use strided accesses. The telecommunications benchmarks, on the other hand, operate on small data sets, use unit stride accesses almost exclusively, and include short reductions. Hence, the benefit from decoupling in CODE is limited to approximately 12%.

We should note here that the comparison results are slightly skewed in favor of VIRAM-1. With both microarchitectures, we use executables that have been optimally tuned in assembly for the VIRAM-1 pipeline. CODE relies less on static scheduling due to its ability to reorder instructions that execute in separate cores. However, better scheduling of instructions that execute in the same vector core could lead to slightly better results. One should also keep in mind that, apart from the performance benefits, CODE provides significant advantages over VIRAM-1 in terms of energy efficiency and design complexity (see Chapter 6.5, page 71). In addition, CODE can reach similar

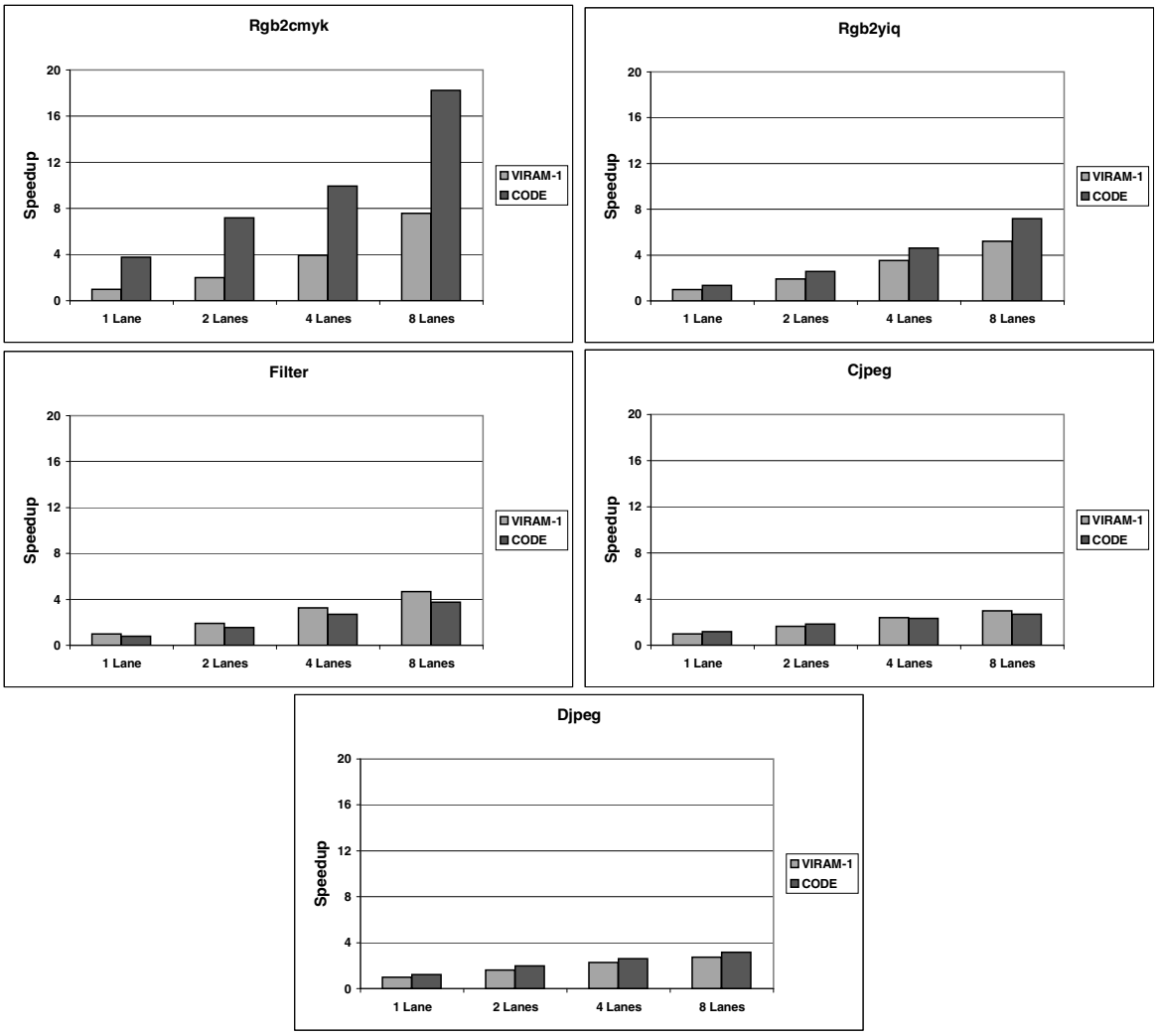


Figure 9.1: The performance of CODE and VIRAM-1 for the consumer benchmarks as a function of the number of lanes. We report performance as speedup over the VIRAM-1 implementation with 1 vector lane. For `Cjpeg` and `Djpeg`, we measure the performance only for the vectorized portion of each benchmark.

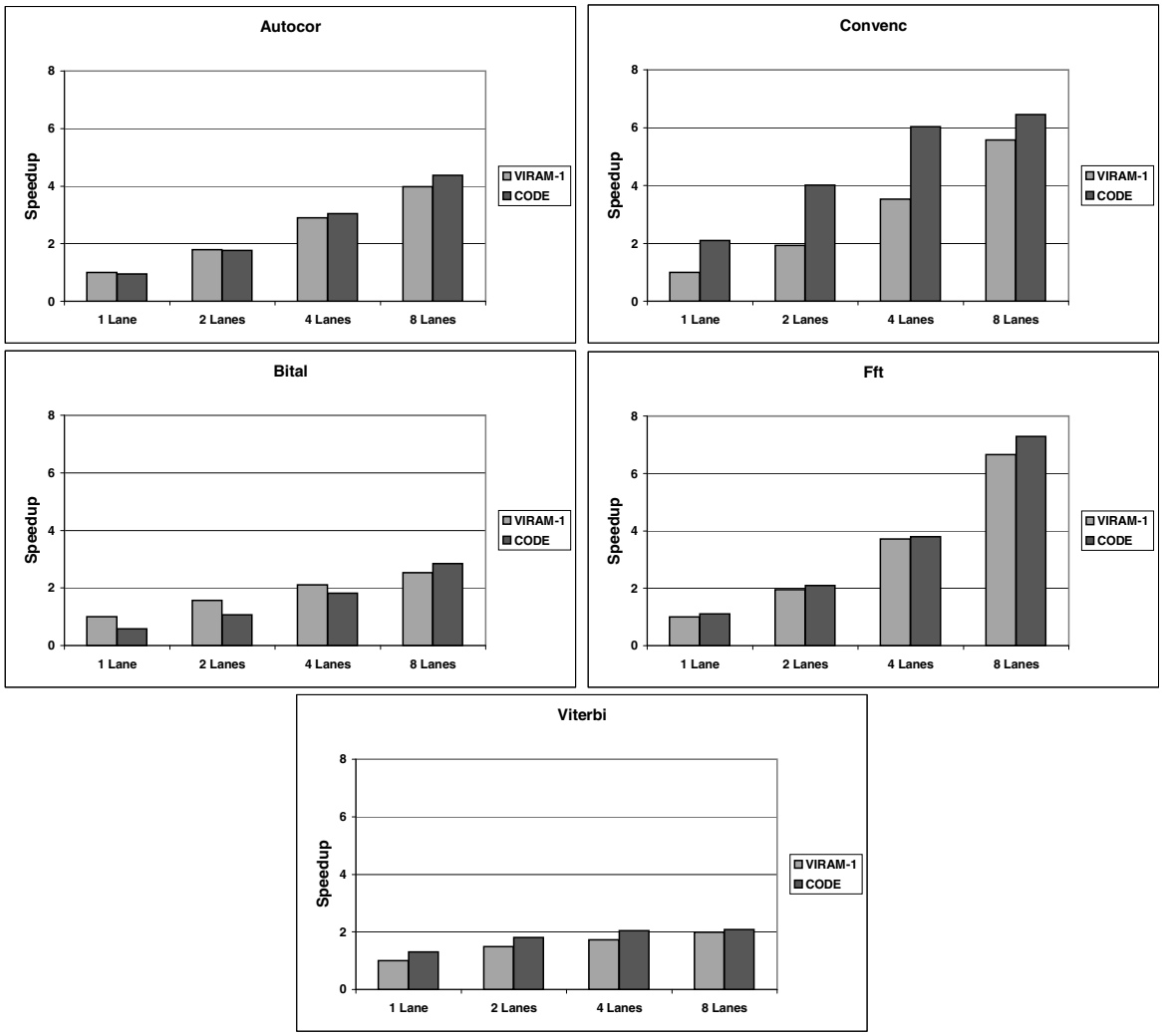


Figure 9.2: The performance of CODE and VIRAM-1 for the telecommunications benchmarks as a function of the number of lanes. We report performance as speedup over the VIRAM-1 implementation with 1 vector lane.

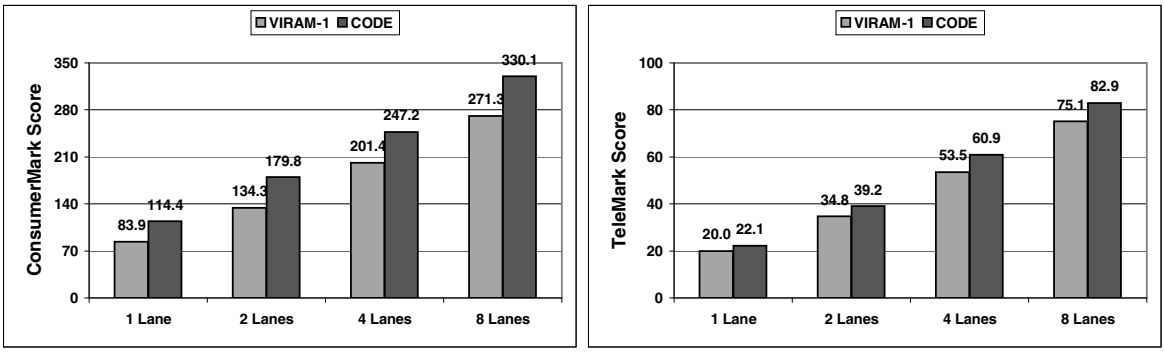


Figure 9.3: The composite scores for the consumer and communications benchmarks for VIRAM-1 and CODE as a function of the number of lanes. We assume that both microarchitectures operate at the clock frequency of 200 MHz. A higher benchmark score reflects higher performance.

performance levels even in the presence of high memory latency (see Chapter 8.4, page 104). On the other hand, the effectiveness of the delayed pipeline of VIRAM-1 drops quickly when the memory latency is higher than ten processor cycles.

9.2 The Impact of Communication Network Bandwidth

The communication network in CODE facilitates vector register transfers between the various cores. Unlike with VIRAM-1, where the centralized vector register file integrates a single-cycle crossbar switch between the inputs and outputs of all functional units, the communication network in CODE can use a simpler and easier to implement interconnect structure. This simplification is possible because the vector cores in CODE need to exchange only a small number of vector operands for each instruction (see Chapter 6.7, page 76).

Regardless of the implementation details, the parameter that establishes the cost and complexity of the communication network is the bandwidth it supports: the number of vector register transfers that can occur in parallel and the number of bits per cycle at which each transfer makes progress. The desired bandwidth determines the number of wires and transistors necessary to implement the network and the complexity of the input and output interfaces in the vector cores.

Figures 9.4 and 9.5 present the performance of the CODE configuration described in Table 9.1 as a function of the bandwidth available on the communication network (BW). We assume that each inter-core transfer advances at the rate of 64 bits per cycle per lane, in the same way that each vector instruction uses a 64-bit datapath per cycle per lane during its execution. Therefore, we scale the available bandwidth at increments of 64 bits per cycle per lane. The notation $BW = n$ specifies that the communication network has sufficient bandwidth to support n concurrent transfers per cycle. We also measure performance for the case of infinite network bandwidth, where an unlimited number of transfers may take place in parallel. For the simulated CODE configuration, the input and output interfaces of the vector cores can support up to 5 concurrent transfers.

With bandwidth $BW = 1$, performance is 20% to 50% lower than with the case of infinite bandwidth. The average number of inter-core transfers for vector registers per instruction varies between 0.3 and 1.0 for nine out of ten EEMBC benchmarks. Hence, $BW = 1$ is not sufficient to support the concurrent execution of up to four vector instructions in the cores available in this CODE configuration. On the other hand, with bandwidth $BW = 2$, performance is within 10% of the case of infinite bandwidth for all benchmarks, regardless of the number of lanes. Even though, there are still stalls when more than 2 instructions attempt to transfer data on the communication

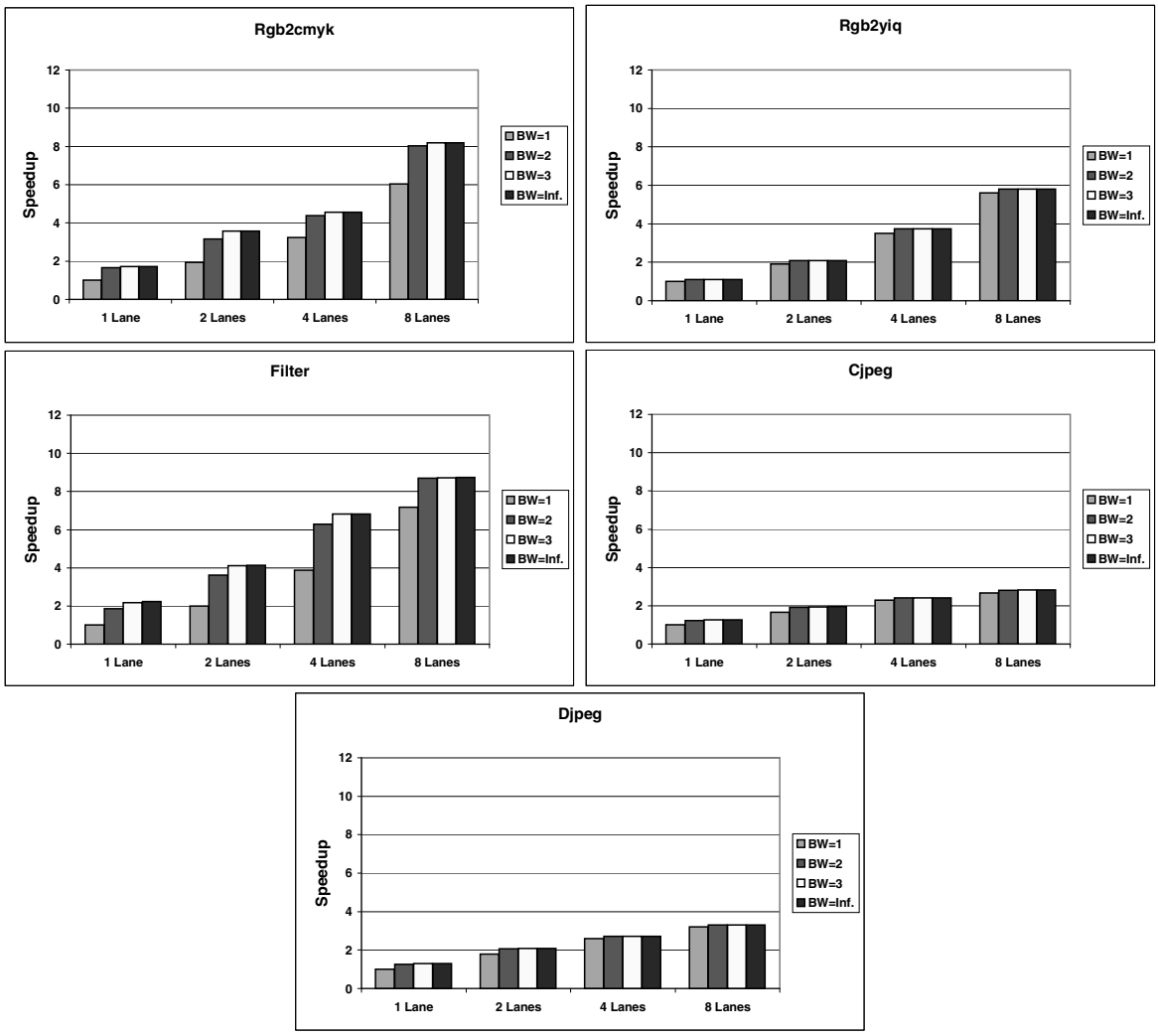


Figure 9.4: The performance of CODE for the consumer benchmarks as a function of the bandwidth (BW) available in the inter-core communication network. The unit of bandwidth is one 64-bit word per cycle. Each lane contains a separate copy of the communication network to interconnect the partitions of vector cores it includes. The last column in each group represents the case of infinite bandwidth. We present performance as speedup over the CODE configuration with 1 lane and network bandwidth $BW = 1$.

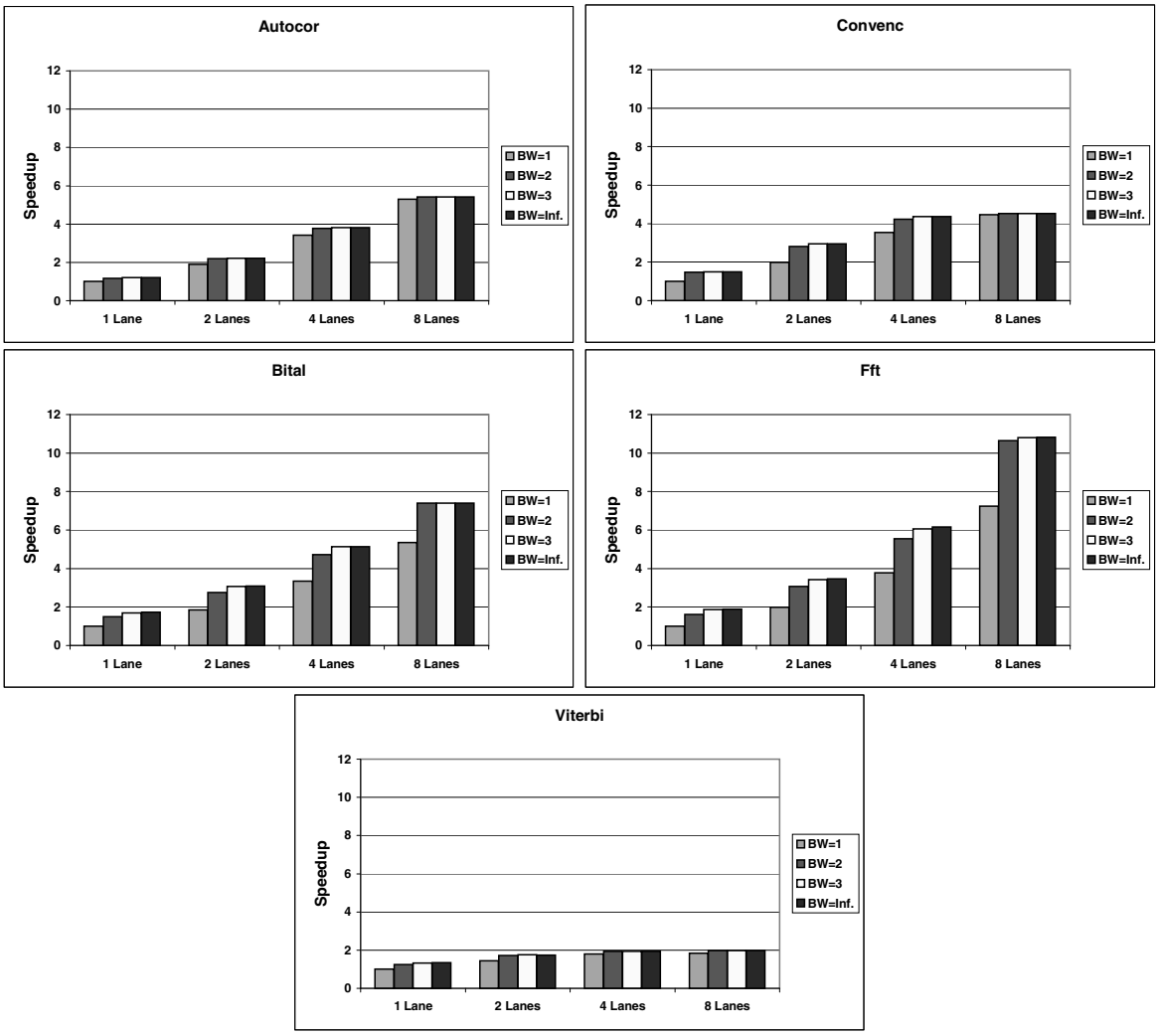


Figure 9.5: The performance of CODE for the telecommunications benchmarks as a function of the bandwidth (BW) available in the inter-core communication network. The unit of bandwidth is one 64-bit word per cycle. Each lane contains a separate copy of the communication network to interconnect the partitions of vector cores it includes. The last column in each group represents the case of infinite bandwidth. We present performance as speedup over the CODE configuration with 1 lane and network bandwidth $BW = 1$.

network, the decoupling capability of CODE hides the latency they introduce.

One would expect that `Filter`, the benchmark that requires more than 1.5 inter-core transfers per vector instruction, would be able to benefit from additional bandwidth on the communication network. However, the main bottleneck for executing a large number of concurrent transfers for `Filter` is the number of input and output interfaces per vector core. Specifically, we would have to introduce additional interfaces to the $IntFull$ execution core in the simulated CODE configuration in order to utilize network bandwidth higher than $BW = 2$. However, it is more efficient in terms of area and energy consumption to introduce more local vector register in the $IntFull$ core in order to reduce the number of transfers per instruction for `Filter` (see Chapter 6.7, page 76).

To support bandwidth $BW = 2$, we can implement two 64-bit buses or two 64-bit rings that connect the partitions of the vector cores in each lane. The latency of the communication network has little impact on performance. CODE can tolerate high latency in the communication network if sufficient bandwidth is available, in the same way it can tolerate high latency for memory accesses. For comparison, it is interesting to notice that the partition of the centralized vector register file in each lane of VIRAM-1 implements a 3×8, 64-bit, single-cycle switch for exchange of operands between its three functional units.

9.3 Scaling CODE with Cores and Lanes

We can scale CODE to exploit additional hardware resources in two orthogonal ways: more lanes in order to execute more operations per cycle for each vector instruction or more cores in order to execute more vector instructions in parallel. Additional lanes should be beneficial to the benchmarks with long vectors, whereas adding cores should accelerate the benchmarks with independent vector instructions within each basic block. Figures 9.6 and 9.7 present the performance of CODE for the ten EEMBC benchmarks as we scale the number of vector lanes and vector cores.

The reference point for performance is a CODE configuration with four vector cores (1 $IntFull$, 1 $LDFull$, 1 $ArithRest$, and 1 state core) and one lane, that can support 2 concurrent transfers on the communication network ($BW = 2$). From there, we scale the number of cores up to 16 according to the ratio of arithmetic to memory operations for each benchmark (see Table 4.4, page 31). For example, `Rgb2yiq` executes two arithmetic operations for every memory access, hence we introduce one load-store core for every two integer cores. We also increase the bandwidth available on the inter-core communication network by $BW = 2$ (2 64-bit words per cycle) for every 4 cores. The memory system for all configuration consists of 16 embedded DRAM banks with 8 sub-banks per bank and the timing characteristics of the IBM SA27E technology used with VIRAM-1.

Even though we simulate CODE configurations with up to 16 cores, it is important to understand that it is practically impossible to make efficient use of more than 10 to 12 cores for any of the benchmarks. The scalar core and the vector issue logic (VIL) of CODE can dispatch only one instruction per cycle, either vector or scalar. The average ratio of scalar to vector instructions for the EEMBC benchmarks is 1.5 (see Table 4.3, page 30). Therefore, CODE must issue 24 scalar instructions with every 16 vector instructions, one for each of the 16 vector core. This takes 40 clock cycles, assuming no other dependencies and stalls. However, even with a single vector lane and vectors of maximum length, one vector instruction can occupy a vector core for only 32 clock cycles. Hence, the available issue bandwidth of 1 instruction per cycle is not sufficient to keep 16 vector cores from idling, even under optimistic assumptions. Without additional instruction issue bandwidth, CODE can exploit up to 10 to 12 vector cores. Its effectiveness with using a large number of cores depends on several factors such as dependencies between vector instructions and the frequency of memory stalls.

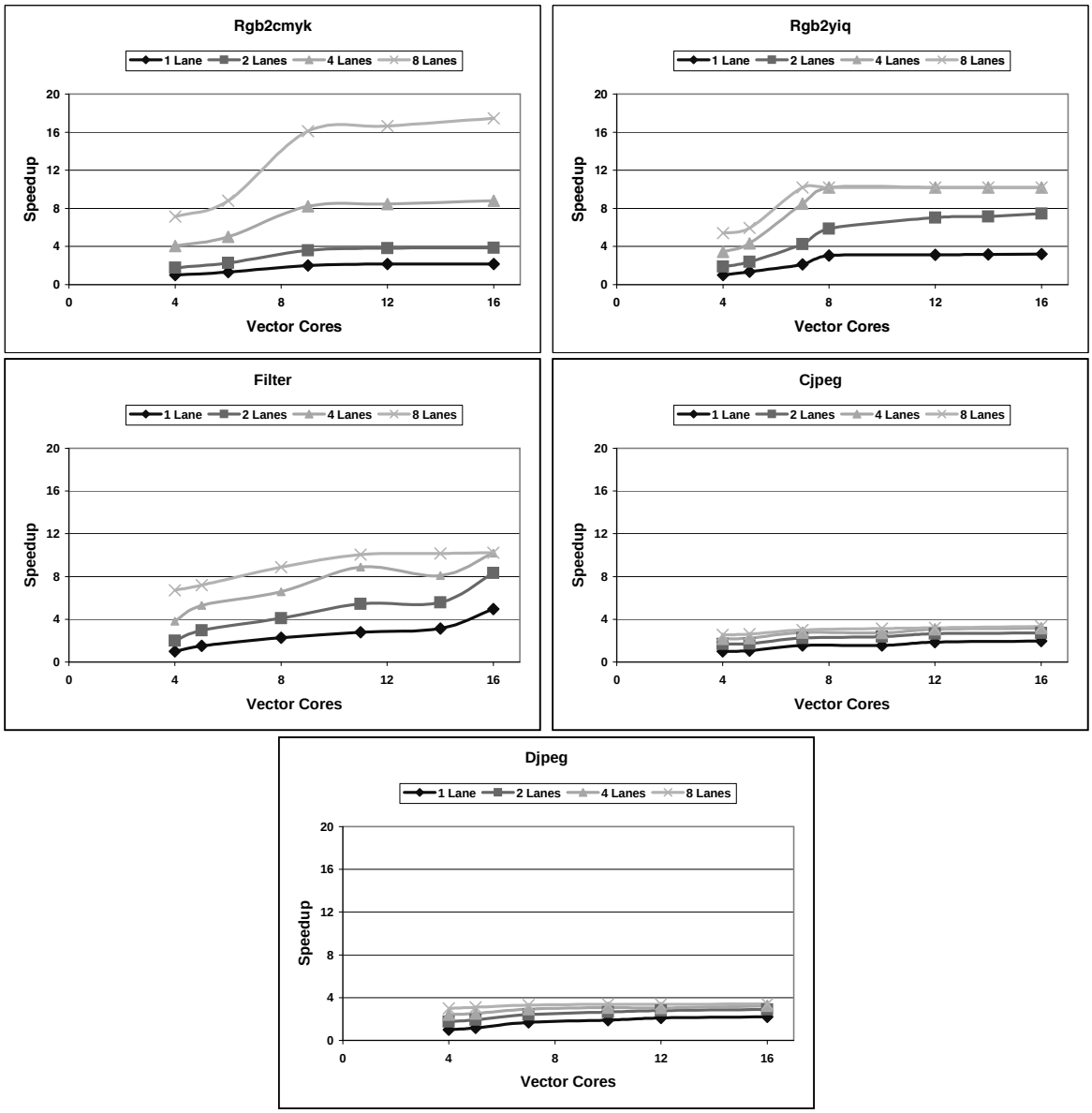


Figure 9.6: The performance of CODE for the consumer benchmarks as a function of the number of vector cores and lanes. We present performance as speedup over the CODE configuration with 4 vector cores and 1 lane. For `Cjpeg` and `Djpeg`, we only measure the vectorized portion of each benchmark.

9.3.1 Consumer Benchmarks

Figure 9.6 describes the scaling behavior for the consumer benchmarks in the EEMBC suite. Three out of five benchmarks (`Rgb2cmyk`, `Rgb2yiq`, and `Filter`) benefit significantly from both additional cores and lanes. `Cjpeg` and `Djpeg`, on the other hand, exhibit limited performance improvement with either scaling method.

The performance of `Rgb2cmyk` and `Rgb2yiq` scales almost linearly with the number of lanes because they perform operations on long vectors. They can also exploit up to 8 vectors cores. Scaling from 4 to 8 cores leads to a proportional performance improvement by a factor or 2, regardless of the number of lanes. Introducing more than 8 vector cores has no effect on performance. The bottleneck is the large number of bank conflicts that lead to serialization of memory accesses, as multiple load-store cores attempt to access data in the same bank, the same row, or even the same column in a single cycle. Additional banks and sub-banks, or the use of an alternative interleaving scheme would not alleviate the problem due to the nature of the memory access pattern of the two benchmarks. However, the ability to merge accesses to the same DRAM row or column, either at the interface of each DRAM bank or with a hardware structure similar to the rake cache for vector processors [Asa98], could allow CODE to scale efficiently to approximately 12 vector cores.

`Filter` behaves similarly to `Rgb2cmyk` and `Rgb2yiq`. Increasing the number of cores to 10 leads to performance improvements by a factor of 2 for up to 4 lanes. With 8 lanes, the improvement rate drops to 1.5 due to the lack of sufficient instruction issue bandwidth. `Filter` runs into the problem of frequent bank conflicts faster than `Rgb2cmyk` and `Rgb2yiq`. However, each extra core introduces 8 additional local vector registers and reduces the pressure on the local vector register files in the other cores. Therefore, `Filter` experiences a small benefit from additional cores due to the reduction in the frequency of inter-core register transfers despite the stalls in the memory system and the low instruction issue bandwidth.

`Cjpeg` and `Djpeg` exhibit little performance improvement when scaling the number of vector cores or lanes. Scaling to either 16 cores or 8 lanes leads to performance improvement of up to a factor of 2. Both benchmarks operate mostly on short vectors which limits the potential of additional lanes. The use of additional cores is limited due to two factors. First, there is an increased number of bank conflicts as multiple load-store cores attempt to access data in the same column in a single cycle. Second, the instruction sequence for the functions for DCT, which dominate the execution time in both cases, includes many dependencies which limits the potential for parallel execution of instructions in multiple cores. The only way to improve the performance of CODE would be to modify the benchmark code in order to allow outer-loop vectorization of the DCT functions or unroll the DCT code to operate on two 8×8 pixel blocks in parallel. Note that the latter would also lead to an increase in the static code size.

9.3.2 Telecommunications Benchmarks

Figure 9.7 describes the scaling behavior of CODE for the telecommunication benchmarks in the EEMBC suite. `Bital` and `Fft` exhibit significant performance improvements with both scaling methods. `Autocor` and `Convenc` benefit only from additional lanes. `Viterbi` cannot exploit additional hardware resources in either of the two forms.

`Autocor` executes series of short reductions on vectors of 64 32-bit elements. The instruction stream for a reduction includes a strictly sequential set of permutation and vector add operations that cannot execute in parallel in multiple vector cores. Hence, `Autocor` can only benefit from the use of multiple lanes. Its performance scales almost linearly with the number of lanes for up to 4 lanes. In order to exploit multiple cores, we would have to modify (unroll) the benchmark code to execute multiple reductions in parallel.

`Convenc` behaves similar to `Autocor`. It executes series of exclusive or reductions which also prevent the parallel execution of vector instructions in multiple cores without the use of unrolling.

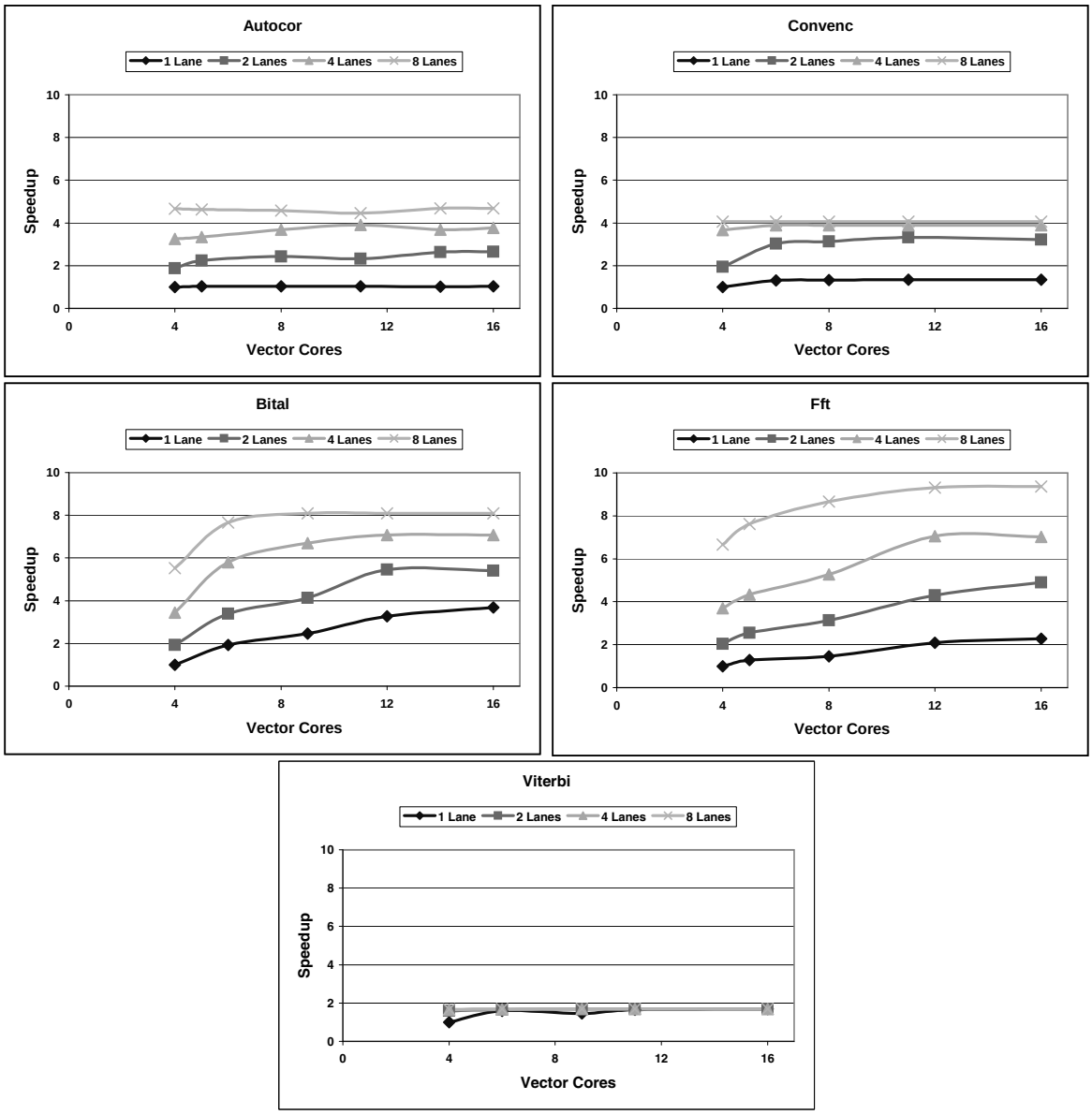


Figure 9.7: The performance of CODE for the telecommunications benchmarks as a function of the number of vector cores and lanes. We present performance as speedup over the CODE configuration with 4 vector cores and 1 lane.

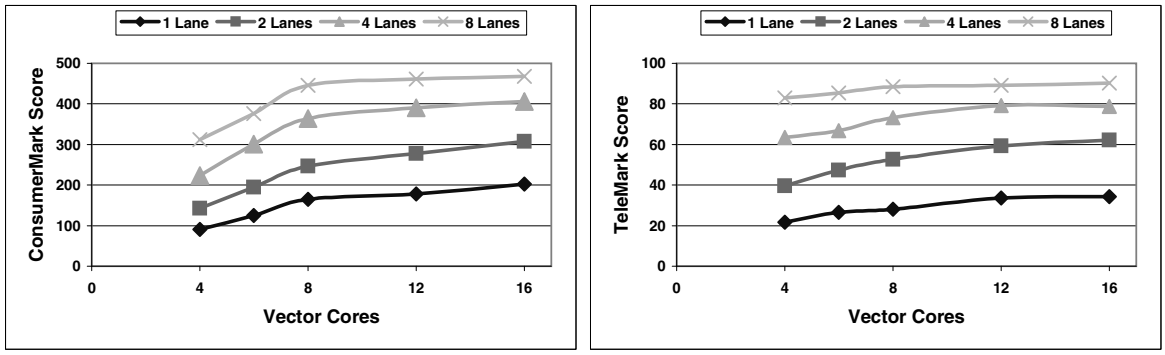


Figure 9.8: The composite score for CODE for the consumer and telecommunications benchmarks as a function of the number of vector cores and lanes. We assume a clock frequency of 200 MHz for all configurations. A higher benchmark score reflects higher performance.

It is interesting to note, however, that `Convenc` already executes as fast as possible for most of the design points in Figure 9.7. For example, with 4 cores and 4 lanes, CODE can execute the 222 instructions in the benchmark in 240 cycles. It is impossible to achieve dramatic improvements for this benchmark without resulting to parallel instruction issue for both scalar and vector instructions.

`Bital` and `Fft` can exploit additional hardware resources in the form of both vector cores and vector lanes. With 8 vector cores, performance for `Bital` and `Fft` improves by factors of 2.0 and 1.6 respectively for up to 4 lanes. With 8 lanes, the gain from additional cores is slightly lower due to lack of sufficient instruction issue bandwidth. The two benchmarks cannot make efficient use of more than 8 vector cores because their code includes a few strictly sequential portions that execute reductions or butterfly permutations.

Finally, `Viterbi` is the one benchmark for which CODE cannot exhibit performance improvements with additional hardware resources. `Viterbi` operates on short vectors which prohibits any gains from the use of multiple vector lanes. In addition, the code for the add-compare-select steps, which dominates the execution time, includes dependencies that prevent instructions from executing in parallel on multiple cores. Restructuring the benchmark code cannot help either as the decoding states in the `Viterbi` algorithm must be swept in order. Hence, we cannot use outer-loop vectorization or perform loop unrolling or software pipelining of the vector loops. The only way to accelerate `Viterbi` is to execute instructions faster by raising the clock frequency for the vector processor and reducing the effective latency of memory accesses.

9.3.3 Discussion

Figure 9.8 summarizes the scaling behavior of CODE for the consumer and telecommunications benchmarks by presenting the composite scores as a function of the number of vector cores and vector lanes.

The performance of CODE scales better for the consumer benchmarks. Doubling the number of vector cores from 4 to 8 leads to 82% and 64% performance improvement for CODE configurations with 1 and 4 lanes respectively. Similarly, each doubling of the number of lanes increases performance by approximately 55% for configurations with 4 to 8 cores. The benefits from the two scaling methods are nearly orthogonal. Hence, the configuration with 8 vector cores and 4 lanes performs 4 times better than the base-line configuration with 4 cores and 1 lane.

The telecommunications score, on the other hand, increases faster with the more lanes than with more cores. In a CODE configuration with 1 core, scaling the number of lanes from 1 to 4 leads to almost 3 times higher performance. In contrast, scaling the number of cores from 4 to 8 in

a configuration with 4 lanes leads to only 30% higher performance. Overall, the configuration with 8 vector cores and 4 lanes performs 3.6 times better than the base-line configuration with 4 cores and 1 lane.

Neither score exhibits a substantial improvement with more than 8 vector cores due to the limited instruction issue bandwidth and the high frequency of memory stalls. Similarly, increasing the number of lanes from 4 to 8 leads to less than 30% performance gains in any case.

9.4 Summary

In this chapter, we presented a detailed evaluation of the performance and scalability potential of the CODE microarchitecture for vector processors. Assuming equal die areas and clock frequencies, CODE outperforms VIRAM-1 by 26% for the consumer benchmarks and 12% for the telecommunications benchmarks. The performance advantage is in addition to the design complexity and energy efficiency benefits of CODE. We also established that the inter-core communication network in CODE must be able to support 2 concurrent register transfer for every 4 vector cores.

In terms of scalability, we demonstrated that five out of ten EEMBC benchmarks can efficiently use additional hardware resources in the form of both vector cores and vector lanes. These benchmarks can use up to 8 vector cores and exhibit nearly linear performance improvements. To exploit more than 8 cores or observe similar benefits for four of the remaining five benchmarks, we need to explore memory systems that allow concurrent load accesses to closely packed data and use code generation techniques that expose more independent instructions (loop unrolling) or create longer vectors (outer-loop vectorization). There is only one EEMBC benchmark (`Viterbi`) which cannot benefit from any of the two scaling methods available in the code microarchitecture, because it contains limited amounts of data-level and instruction-level level parallelism.

Chapter 10

Conclusions

"I hope for nothing,
I fear nothing,
I am free!"
Nikos Kazantzakis

Embedded multimedia systems require high performance for multimedia functions at low energy consumption and low design complexity. We have shown that this is possible by exploiting the data-level parallelism in multimedia applications with vector microprocessors. A vector architecture can express data-level parallelism explicitly, which allows for fast execution of operations on parallel hardware structures that are simple and modular. The integration of the processor and its main memory system on a single die provides the high memory bandwidth necessary for a vector processor in a cost-effective way. In addition, both the vector processor and its memory system scale well with CMOS technology and can translate improvements in circuits capacity to actual performance gains.

In summary, the main contributions of this dissertation are:

- We introduced the VIRAM vector instruction set architecture (ISA) for embedded multimedia systems. The vector instructions in the VIRAM ISA can express the data-level parallelism in multimedia application in an explicit and compact manner.
- We presented the microarchitecture, design, and evaluation of the VIRAM-1 media-processor. VIRAM-1 integrates a simple, yet highly parallel, vector processor with an embedded DRAM memory system. It demonstrates that a vector processor can provide high performance for multimedia tasks, at low energy consumption, and low design complexity.
- We proposed the CODE vector microarchitecture for the VIRAM ISA that combines composite organization with decoupled execution. The simplified vector register file and the ability to tolerate high memory latency allow CODE to extend the performance and energy advantages of VIRAM-1 across a larger design space. It can also support precise exceptions with a minimal impact on performance.
- We demonstrated that embedded DRAM is a suitable technology for the memory system of vector media-processors. Embedded DRAM provides the high memory bandwidth required by a vector processor at low energy consumption and moderate access latency.

Future Work

Even though we have made significant inroads towards efficient processors for embedded multimedia systems, a lot of work remains to be done. The following are some of the key areas for future research based on the conclusions of this thesis:

- **Development of complete multimedia applications**: The EEMBC benchmarks used in this thesis demonstrate the potential of vector processors with multimedia functions, even for tasks that are considered non-vectorizable, such as Viterbi decoding. However, it is important to proceed with the development of end-user applications in order to provide complete proof of concept. Some of the most interesting applications are 3-D graphics rendering, video encoding and decoding, and speech recognition.

- **Languages and compilers for multimedia**: Programming languages such as C and C++ provide limited support for expressing fixed-point arithmetic and the explicit parallelism in the computation, memory accesses, and IO in multimedia functions. The current approaches to this problems, such as processor-specific pragma statements and language extensions, are neither portable nor general. It is necessary to explore languages and compilers that overcome this obstacle and enable the full use of the features available in multimedia architectures like VIRAM.

- **Improved memory systems for large CODE configurations**: In Chapter 9, we saw that the ability of CODE to exploit additional hardware resources is often limited by conflicts in the memory system. Memory organizations that are able to merge accesses to neighboring data or to operate concurrently on groups of related data streams can lead to significant performance and energy improvements for highly parallel microarchitectures such as CODE.

- **Architectures for data-level and thread-level parallelism**: Apart from data-level parallelism, multimedia programs exhibit large amounts of thread-level parallelism. Applications such as video decoding consist of a number of media functions operating in a pipeline. In other cases, several media functions operate in parallel in order to process multiple streams of video or support both visual and audio effects. The next big improvement in multimedia performance will occur when we successfully merge architectural and microarchitectural techniques for exploiting both data-level and thread-level parallelism.

- **Specialized hardware engines for complicated tasks**: For certain important tasks such as Huffman encoding, it is difficult to exploit their data-level or instruction-level parallelism using conventional architectural approaches. Specialized instructions for such tasks lead to limited performance improvements and are difficult to implement across multiple generations of microprocessors. An alternative approach to explore is the execution of such difficult tasks on specialized hardware engines that are closely integrated with a regular microprocessor and its memory system.

- **Modular architectures for yield and reliability improvements**: In this thesis, we used modular implementations of vector processors to provide enhancements in performance, energy consumption, and design complexity. However, we can also use modularity to improve the yield of semiconductor chips and their reliability. A modular processor with replicated components can sustain some number of permanent or transient errors without becoming unusable. It is interesting to explore the testing and operation techniques that activate or deactivate modular components on demand in order to increase yield or improve reliability.

Bibliography

[ABHS89] M. August, G. Brost, C. Hsiung, and C. Schiffleger. Cray X-MP: The Birth of a Supercomputer. *IEEE Computer*, 22(1):45–52, January 1989.

[AEJ+02] A. Allan, D. Edenfeld, W. Joyner, M. Rodgers, and Y. Zorian. 2001 Technology Roadmap for Semiconductors. *IEEE Computer*, 35(1):42–53, January 2002.

[AF88] D. Alpert and M. Flynn. Performance Trade-offs for Microprocessor Cache Memories. *IEEE Micro*, 8(4):44–54, July 1988.

[AHBK00] V. Agarwal, S. Hrisikesh, D. Burger, and S.W. Keckler. Clock Rate vs IPC: The End of Road for Conventional Microarchitectures. In *the Proceedings of the 27th Intl. Symposium on Computer Architecture*, pages 248–259, Vancouver, BC, Canada, June 2000.

[AJ97] K. Asanović and D. Johnson. Torrent Architecture Manual, Revision 2.11. Technical Report CSD-97-930, Computer Science Division, University of California at Berkeley, 1997.

[Amd81] Amdahl Corporation. *Amdahl 470V/8 Computing System Machine Reference Manual*, October 1981.

[Asa98] K. Asanović. *Vector Microprocessors*. PhD thesis, Computer Science Division, University of California at Berkeley, 1998.

[AST67] D.W. Anderson, F.K. Sparacio, and R.M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction Handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.

[BCS93] R.W. Brodersen, A. Chandrakasan, and S. Sheng. Design Techniques for Portable Systems. In *the Digest of Technical Papers of the Intl. Solid-State Circuits Conference*, San Francisco, CA, February 1993.

[BG97] D. Burger and D. Goodman. Billion-Transistor Architectures - Guest Editors' Introduction. *IEEE Computer*, 30(9):46–48, September 1997.

[Buc62] W. Bucholz, editor. *Planning a Computer System*. McGraw-Hill, New York, NY, 1962.

[CB94] T. Chen and J.L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *the Proceedings of the 21st Intl. Symposium on Computer Architecture*, pages 223–232, Chicago, IL, April 1994.

[CDJ+97] T.M. Conte, P.K. Dubey, M.D. Jennings, R.B. Lee, A. Peleg, S. Rathnam, M. Schlansker, P. Song, and A. Wolfe. Challenges to Combining General-purpose and Multimedia Processors. *IEEE Computer*, 30(12):33–37, December 1997.

[Con81] Control Data Corporation, Arden Hills, MN. *CDC Cyber 200 Model 205 System Hardware Reference Manual*, 1981.

[Cra93] Cray Research Inc., Chippewa Falls, WI 54729. *Cray Y-MP C90 System Programmer Reference Manual*, June 1993.

[Cra00] Cray Research Inc., Chippewa Falls, WI 54729. *Cray Standard C and Cray C++ Reference Manual (004-2179-00)*, 2000.

[Cri97] Richard Crisp. Direct Rambus Technology: The Main Memory Standard. *IEEE Micro*, 17(6):18–28, December 1997.

[CSG98] D. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: a Hardware/software Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[CVE99] J. Corbal, M. Valero, and R. Espasa. Exploiting a New Level of DLP in Multimedia Applications. In *the Proceedings of the 32nd Intl. Symposium on Microarchitecture*, Haifa, Israel, November 1999.

[Dal98] W. Dally. Tomorrow's Computing Engines. Keynote Speech, the 4th Intl. Symposium on High-Performance Computer Architecture, February 1998.

[DD97] K. Diefendorff and P. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, 30(9):43–45, September 1997.

[Dig96] Digital Equipment Corp. *Digital Semiconductor Alpha 21064 and 21064A Microprocessors: Hardware Reference Manual (EC-Q9ZUC-TE)*, July 96.

[EB98] J. Eyre and J. Bier. DSP Processors Hit the Mainstream. *IEEE Computer*, 31(8):51–59, August 1998.

[ECCH00] D. Engler, B. Chelf, A. Chou, and S Hallem. Checking System Rules using System-specific, Programmer-written Compiler Extensions. In *the Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, CA, October 2000.

[Emb01] Embedded Processors Watch. Microprocessor Report, October 2001.

[Esp97] R. Espasa. *Advanced Vector Architectures*. PhD thesis, Universitat Politècnica de Catalunya, February 1997.

[ESSC99] D.G. Elliott, M. Stumm, W.M. Snelgrove, and C. Cojocaru. Computational RAM: implementing processors in memory. *IEEE Design and Test of Computers*, 16(1):32–41, January 1999.

[EV96] R. Espasa and M. Valero. Decoupled Vector Architecture. In *the Proceedings of the 2nd Intl. Symposium on High-Performance Computer Architecture*, pages 281–90, San Jose, CA, February 1996.

[EVS97] R. Espasa, M. Valero, and J.E. Smith. Out-of-order Vector Architectures. In *the Proceedings of the 30th Intl. Symposium on Microarchitecture*, pages 160–70, Research Triangle Park, NC, December 1997.

[Far97] K. I. Farkas. *Memory-System Design Considerations for Dynamically-Scheduled Microprocessors*. PhD thesis, University of Toronto, 1997.

[FBFD00] P. Faraboschi, G. Brown, J.A. Fisher, and G. Desoll. Lx: a Technology Platform for Customizable VLIW Embedded Processing. In *the Proceedings of the 27th Intl. Symposium on Computer Architecture*, pages 203–13, Vancouver, BC, Canada, June 2000.

[FCJV97] K.I. Farkas, P. Chow, N.P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Processor Cycle Time Through Partitioning. In *the Proceedings of the 30th Intl. Symposium on Microarchitecture*, pages 327–56, Research Triangle Park, NC, December 1997.

[Fis83] J.A Fisher. VLIW Architectures and the ELI-512. In *the Proceedings of the 10th Intl. Symposium on Computer Architecture*, pages 140–150, Stockholm, Sweden, June 1983.

[FJL85] J. Frailong, W Jalby, and J. Lenfant. XOR-schemes: A Flexible Data Organization in Parallel Memories. In *the Proceedings of the Intl. Conference on Parallel Processing*, pages 276–283, August 1985.

[FP91] J.W.C. Fu and J.H. Patel. Data Prefetching in Multiprocessor Vector Cache Memories. In *the Proceedings of the 18th Intl. Symposium on Computer Architecture*, pages 54–63, Toronto, Canada, May 1991.

[FPC+97] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, P. Patterson, T. Anderson, and K. Yelick. The Energy Efficiency of IRAM Architectures. In *the Proceedings of the 24th Intl. Symposium on Computer Architecture*, pages 327–37, Denver, CO, June 1997.

[FWL99] J. Fritts, W. Wolf, and B. Liu. Understanding Multimedia Applications Characteristics for Designing Programmable Media Processors. In *the Proceedings of the SPIE Photonics West*, San Jose, CA, January 1999.

[G+85] J.R. Goodman et al. PIPE: A VLSI Decoupled Architecture. In *the Proceedings of the 12th Intl. Symposium on Computer Architecture*, pages 20–17, Boston, MA, June 1985.

[GBS94] S.L. Graham, D.F Bacon, and O.J. Sharp. Compiler Transformations for High Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[Gre00] P. Green. A 1GHz IA-32 Microprocessor Implemented on 0.18um Technology with Aluminum Interconnect. In *the Digest of Technical Papers of the Intl. Solid-State Circuits Conference*, San Francisco, CA, February 2000.

[GRE+01] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: a Free, Commercially Representative Embedded Benchmark Suite. In *the Proceedings of the 4th Workshop on Workload Characterization*, Austin, TX, December 2001.

[Gro90] G.F. Grohoski. Machine Organization of the IBM RISC System/6000 Processor. *IBM Journal of Research and Development*, 34:37–58, January 1990.

[Gro98] G. Grohoski. Challenges and Trends in Processor Design: Reining in Complexity. *IEEE Computer*, 31(1):41–42, January 1998.

[GS92] J.D. Gee and A.J. Smith. The Performance Impact of Vector Processor Caches. In *the Proceedings of the 25th Hawaii Intl. Conference on System Sciences*, pages 437–449, January 1992.

[GW76] H. Gurnow and B. Wichmann. A Synthetic Benchmark. *IEEE Computer*, 19(1), February 1976.

[H+90] H. Hikada et al. The Cache DRAM Architecture: A DRAM with an On-Chip Cache Memory. *IEEE Micro*, 10(2):14–25, March 1990.

[Hal99a] R. Halfhill. MIPS Plays Hardball with Soft Cores. *Microprocessor Report*, 13(14):1–2, October 1999.

[Hal99b] T. Halfhill. Embedded Benchmarks Grow Up. *Microprocessor Report*, 13(8):1–5, June 1999.

[Hei94] J. Heinrich. *MIPS R4000 Microprocessor: User's Manual*. Silicon Graphics, Inc., 1994.

[Hei98] J. Heinrich. *MIPS RISC Architecture, 2nd Edition*. Silicon Graphics, Inc., 1998.

[Hen00] J. Henning. SPEC CPU2000: Measuring Performance in the New Millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[HJBG82] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill. Hardware/Software Tradeoffs for Increased Performance. In *the Proceedings of the 1st Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Palo Alto, CA, April 1982.

[HKK+99] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, Srivastava A., J. Shin, and P. Joonseok. Mapping Irregular Computations to DIVA, a Data-Intensive Architecture. In *the Proceedings of the Supercomputing Conference*, Portland, OR, November 1999.

[HL96] Steven W. Hammond and Richard D. Loft. Architecture and Application: The Performance of NEC SX-4 on the NCAR Benchmark Suite. In *the Proceedings the Intl. Conference on Supercomputing*, pages 17–22, Pittsburgh, PA, November 1996.

[HMH01] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[HP02] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, third edition, 2002.

[HS93] W. Hsu and J.E. Smith. Performance of Cached DRAM Organizations in Vector Supercomputers . In *the Proceedings of the 20th Intl. Symposium on Computer Architecture*, pages 327–336, San Diego, CA, May 1993.

[HT72] R. G. Hintz and D. P. Tate. Control Data STAR-100 Processor Design. In *the Proceedings of COMPCON*, pages 1–4, September 1972.

[IEE98] IEEE. *The Intl. Solid-State Circuits Conference, Digest of Technical Papers*, volume 41, San Francisco, CA, February 1998.

[IEE99] IEEE. *The Intl. Solid-State Circuits Conference, Digest of Technical Papers*, volume 42, San Francisco, CA, February 1999.

[IEE00] IEEE. *The Intl. Solid-State Circuits Conference, Digest of Technical Papers*, volume 43, San Francisco, CA, February 2000.

[Int00] Intel Corporation. *The IA-32 Intel Architecture Software Developer's Manual*, 2000.

[JM98] B. Jacob and T. Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, 18(4):60–75, July 1998.

[Jon89] Tom Jones. Engineering Design of the Convex C2. *IEEE Computer*, 22(1):36–44, January 1989.

[Jon92] F. Jones. A New Era of Fast Dynamic RAMs. *IEEE Spectrum*, 29(10):43–45, October 1992.

[JYK+00] D. Judd, K. Yelick, C. Kozyrakis, D. Martin, and D. Patterson. Exploiting On-Chip Bandwidth in the VIRAM Compiler. In *the Proceedings of the 2nd Workshop on Intelligent Memory Systems*, volume 2107 of *Lecture Notes in Computer Science*, pages 122–34, Cambridge, MA, 2000. Springer Verlag.

[Kes99] R.E Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[KGM+00] C. Kozyrakis, J. Gebis, D Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. VIRAM: A Media-oriented Vector Processor with Embedded DRAM. In *the Conference Record of the Hot Chips XII Symposium*, Palo Alto, CA, August 2000.

[KHY+99] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *the Proceedings of the Intl. Conference on Computer Design*, pages 192–201, Austin, TX, October 1999.

[Kil98] E. Killian. Challenges and Trends in Processor Design: Challenges, Not Roadblocks. *IEEE Computer*, 31(1):44–45, January 1998.

[KJG+01] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick. Hardware/compiler Codevelopment for an Embedded Media Processor. *Proceedings of the IEEE*, 89(11):1694–709, Nov 2001.

[KMK01] D. Kim, R. Managuli, and Y. Kim. Data Cache and Direct Memory Access in Programming Mediaprocessors. *IEEE Micro*, 21(4):33–41, July 2001.

[Koz99] C. Kozyrakis. A Media-enhanced Vector Architecture for Embedded Memory Systems. Technical Report CSD-99-1059, Computer Science Division, University of California at Berkeley, 1999.

[KSF+94] L. Kontothanassis, R. A. Sugumar, G.J. Faanes, J.E. Smith, and M.L. Scott. Cache Performance in Vector Supercomputers. In *the Proceedings of the Supercomputing Conference*, pages 255–264, Washington, DC, November 1994.

[Law75] D. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transactions on Computers*, C-24:1145–55, December 1975.

[LBSL97] P. Lapsley, J. Bier, A. Shoham, and E. Lee. *DSP Processor Fundamentals: Architectures and Features*. IEEE Press, 1997.

[Lev00] M. Levy. EEMBC 1.0 Scores, Part 1: Observations. *Microprocessor Report*, pages 1–7, August 2000.

[Lev01] M. Levy. ManArray Devours DSP Code. *Microprocessor Report*, pages 1–7, October 2001.

[Lew98] T. Lewis. Information Appliances: Gadget Netopia. *IEEE Computer*, 31(1):59–68, January 1998.

[LPMS97] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: a Tool for Evaluating and Synthesizing Multimedia and Communications systems. In *the Proceedings of the 30th Intl. Symposium on Microarchitecture*, pages 330–5, Research Triangle Park, NC, December 1997.

[Mar99] D. Martin. *Vector Extensions to the MIPS-IV Instruction Set Architecture*. Computer Science Division, University of California at Berkeley, January 1999.

[MIP01] MIPS Technologies, Inc. *MIPS64 Architecture for Programmers, Revision 0.95*, 2001.

[Moo65] G.E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38:114–117, April 1965.

[MS+99] Y. Matsui, K. Sakakibara, et al. 64Mbit Virtual Channel Synchronous DRAM. *NEC Research and Development Journal*, 40(3):282–6, July 1999.

[MU84] K. Miura and K. Uchida. FACOM Vector Processor System: VP-100/VP-200. In *the Proceedings of NATO Advanced Research Workshop on High Speed Computing*, volume F7. Springer-Verlag, 1984.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[MV96] M. Moudgill and S. Vassiliadis. Precise Interrupts. *IEEE Micro*, 16(1):58–67, February 1996.

[Org99] International Standards Organization. *ISO/IEC 9899: C Programming Language Standard*. ISO, 1999.

[PAC+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, 17(2):34–44, April 1997.

[PF00] M. Poess and C Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *the SIGMOD Record*, 29(4), December 2000.

[PH98] D. Patterson and J. Hennessy. *Computer Organization and Design : the Hardware/Software Interface*. Morgan Kaufmann, San Francisco, CA, second edition, 1998.

[Phi98] M. Phillip. A Second Generation SIMD Microprocessor Architecture. In *the Conference Record of the Hot Chips X Symposium*, Palo Alto, CA, August 1998.

[PJS96] S. Palacharla, N.P. Jouppi, and J.E. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-1996-1328, University of Wisconsin-Madison, November 1996.

[PK94] S. Paracharla and R.E Kessler. Evaluating Stream Buffers as A Secondary Cache Replacement. In *the Proceedings of the 21st Intl. Symposium on Computer Architecture*, pages 24–33, Chicago, IL, May 1994.

[PMSB88] A. Padegs, B. Moore, R. Smith, and W. Buchholz. The IBM/370 Vector Architecture: Design Considerations. *IEEE Transactions on Computers*, 37(5):509–19, May 1988.

[Pri96] B. Prince. *High Performance Memories : New Architecture DRAMs and SRAMs, Evolution and Function*. Chichester, 1996.

[Prz94] S. Przybylski. *New DRAM Technologies: A Comprehensive Analysis of the New Architectures*. MicroDesign Resources, Sebastopol, CA, 1994.

[Rau91] B. Rau. Pseudo-random Interleaved Memories. In *the Proceedings of the 18th Intl. Conference on Computer Architecture*, pages 74–83, Toronto, Canada, May 1991.

[RDK+98] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *the Proceedings of the 31st Intl. Symposium on Microarchitecture*, pages 3–13, Dallas, TX, November 1998.

[RDKM00] S. Rixner, W.J. Dally, B. Khailany, and P. Mattson. Register organization for media processing. In *the Proceedings of the 6th Intl. Symposium on High-Performance Computer Architecture*, pages 375–86, Touluse, France, January 2000.

[Ric96] D.S. Rice. High-Performance Image Processing Using Special-Purpose CPU Instructions: The UltraSPARC Visual Instruction Set. Technical Report CSD-96-901, University of California at Berkeley, 1996.

[Rus78] R. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

[Sez93] A. Seznec. A Case for Two-way Skewed Associative caches. In *the Proceedings of the 20th Intl. Symposium on Computer Architecture*, pages 169–178, San Diego, CA, May 1993.

[SFS00] J.E. Smith, G. Faanes, and R. Sugumar. Vector Instruction Set Support for Conditional Operations. In *the Proceedings of 27th Intl. Symposium on Computer Architecture*, pages 260–9, Vancouver, BC, Canada, June 2000.

[Shi98] T. Shimizu. M32R/D - A Single Chip Microcontroller with A High Capacity 4MB Internal DRAM. In *the Conference Record of Hot Chips X Symposium*, Palo Alto, CA, August 1998.

[Sit92] R. Sites. *Alpha Architecture Reference Manual*. Digital Press, Oct 1992.

[SK99] I.S. Subramanian and H.L. Kalter. Embedded DRAM Technology: Opportunities and Challenges. *IEEE Spectrum*, 36(4):56–64, April 1999.

[SKA01] M. Sung, R. Krashinsky, and K. Asanovic. Multithreading Decoupled Architectures for Complexity-Effective General Purpose Computing. In *the Workshop on Memory Access Decoupled Architectures, PACT'01*, Barcelona, Spain, September 2001.

[SL99] M.G. Stoodley and C.G Lee. Vector Microprocessors for Desktop Computing. In *the Proceedings of the 32nd Intl. Symposium on Microarchitecture*, Haifa, Israel, November 1999.

[Smi82] A.J Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smi84] J.E. Smith. Decoupled Access/Execute Computer Architecture. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[Smi89] J.E. Smith. Dynamic Instruction Scheduling and the Astronautics ZS-1. *IEEE Computer*, 22(7):21–35, July 1989.

[SN96] A. Saulsbury and A. Nowatzyk. Missing the memory wall: the case for processor/memory integration. In *the Proceedings of the 23rd Intl. Symposium on Computer Architecture*, pages 90–101, Philadelphia, PA, May 1996.

[Soh90] G. Sohi. Instruction Issue Logic for High-Performance, Interruptable, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39:349–359, March 1990.

[Soh93] G. Sohi. High-bandwidth Interleaved Memories for Vector Processors - A Simulation Study. *IEEE Transactions on Computers*, 42(1):34–45, January 1993.

[SP88] J.E. Smith and A.R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, 37(5):562–73, May 1988.

[SRD96] G. Slavenburg, S. Rathnam, and H. Dijkstra. The Trimedia TM-1 PCI VLIW Media Processor. In *the Conference Record of the Hot Chips VIII Symposium*, Palo Alto, CA, August 1996.

[SS95] J.E. Smith and G.S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83(12):1609–24, December 1995.

[SS01a] N. Slingerland and A.J. Smith. Cache Performance for Multimedia Applications. In *the Proceedings of the 15th Intl. Conference on Supercomputing*, pages 204–217, Sorrento, Italy, June 2001.

[SS01b] N. Slingerland and A.J. Smith. Performance Analysis of Instruction Set Architecture Extensions for Multimedia. In *the 3rd Workshop on Media and Stream Processors*, pages 204–217, Austin, TX, December 2001.

[ST92] J.E. Smith and W.R. Taylor. Characterizing Memory Performance in Vector Multiprocessors. In *the Proceedings of the Intl. Conference on Supercomputing*, pages 35–44, Minneapolis, MN, July 1992.

[SWP86] J.E. Smith, S. Weiss, and Y. Pang. A Simulation Study of Decoupled Architecture Computers. *IEEE Transactions on Computers*, C-35(8):692–701, August 1986.

[TCC+00] M. Tremblay, J. Chan, S. Chaundry, W. Conigliaro, and S. Tse. The MAJC Architecture: a Synthesis of Parallelism and Scalability. *IEEE Micro*, pages 12–25, November 2000.

[TD+00] O. Takahashi, S. Dhong, et al. 1-GHz Fully Pipelined 3.7-ns Address Access Time 8k-1024 Embedded Synchronous DRAM Macro. *IEEE Journal of Solid State Circuits*, 35(11):1673–8, November 2000.

[TFP92] G. Tyson, M. Farrens, and A. Pleszkun. MISC: A Multiple Instruction Stream Computer. In *the Proceedings of the 25th Intl. Symposium on Microarchitecture*, pages 193–96, Portland, OR, December 1992.

[Tho70] J.E. Thornton. *Design of a Computer - The Control Data 6600.* Scott, Foresman and Co, Glenview, IL, 1970.

[TM95] N. Topham and K. McDougall. Performance of the Decoupled ACRI-1 Architecture: the Perfect Club. In *the Proceedings of Intl. Conference on High-Performance Computing and Networking*, pages 472–80, Milan, Italy, May 1995.

[Tom67] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[TRMM95] N. Topham, A. Rawsthorne, C. McLean, and M. Mewissen. Compiling and optimizing for decoupled architectures. In *the Proceedings of the Intl. Conference on Supercomputing*, pages 1026–87, San Diego, CA, December 1995.

[Tru97] C. Truong. The VelociTi Architecture of the TMS230C6x. In *the Conference Record of Hot Chips IX Symposium*, Palo Alto, CA, August 1997.

[Tur98] J. Turley. NEC VR5400 Makes Media Debut. *Microprocessor Report*, 12(3):1–4, March 1998.

[UIT94] T. Utsumi, M. Ikeda, and M. Takamura. Architecture of the VPP500 Parallel Supercomputer. In *the Proceedings of the Intl. Conference on Supercomputing*, pages 478–487, Washington, DC, November 1994.

[Vit67] A.J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, IT(13):260–269, April 1967.

[WAK+96] J. Wawrzynek, K. Asanović, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. SPERT-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, March 1996.

[Wal91a] D. Wall. Limits of Instruction-level Parallelism. In *the Proceedings of the 4th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–88, Santa Clara, CA, April 1991.

[Wal91b] G.K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):30–44, April 1991.

[War82] W.P. Ward. Minicomputer Blasts Through 4 Million Instructions per Second. *Electronics*, pages 155–59, January 1982.

[Wei84] R. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(19):2013–1030, October 1984.

[Wil02] S. Williams. The VIRAM Verification Development. Master's thesis, Computer Science Division, University of California at Berkeley, 2002.

[WM95] W. Wulf and S. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[Wor01] Workstation Processors Watch. Microprocessor Report, December 2001.

[Wul92] W.M. Wulf. Evaluation of the WM Computer Architecture. In *the Proceedings of the 19th Intl. Symposium on Computer Architecture*, pages 382–390, Gold Coast, Australia, May 1992.

[Yea96] K.C Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[YHO97] T. Yamauchi, L. Hammond, and K. Olukotun. The Hierarchical Multi-bank DRAM: a High-performance Architecture for Memory Integrated with Processors. In *the Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 303–19, Ann Arbor, MI, September 1997.

[YP92] T. Yeh and Y. Patt. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. In *the Proceedings of the 25th Intl. Symposium on Microarchitecture*, pages 129–139, Portland, OR, December 1992.

[ZMM94] V. Zivojnovic, J. Martinez, and H. Meyr. DSPstone: A DSP-oriented Benchmarking Methodology. In *the Proceedings of the Intl. Conference on Signal Processing Applications and Technology*, Dallas, TX, October 1994.

[ZZZ01] Z. Zang, Z. Zhu, and X. Zhang. Cached DRAM for ILP Processor Memory Access Latency Reduction. *IEEE Micro*, 21(4):22–32, July 2001.