Vector Microprocessors

by

Krste Asanović

B.A. (University of Cambridge) 1987

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, BERKELEY

Committee in charge:

Professor John Wawrzynek, Chair
Professor David A. Patterson
Professor David Wessel

Spring 1998

The dissertation of Krste Asanović is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 1998

Vector Microprocessors

by
Krste Asanović

Abstract

Vector Microprocessors

by

Krste Asanović

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

Most previous research into vector architectures has concentrated on supercomputing applications and small enhancements to existing vector supercomputer implementations. This thesis expands the body of vector research by examining designs appropriate for single-chip full-custom *vector microprocessor* implementations targeting a much broader range of applications.

I present the design, implementation, and evaluation of T0 (Torrent-0): the first single-chip vector microprocessor. T0 is a compact but highly parallel processor that can sustain over 24 operations per cycle while issuing only a single 32-bit instruction per cycle. T0 demonstrates that vector architectures are well suited to full-custom VLSI implementation and that they perform well on many multimedia and human-machine interface tasks.

The remainder of the thesis contains proposals for future vector microprocessor designs. I show that the most area-efficient vector register file designs have several banks with several ports, rather than many banks with few ports as used by traditional vector supercomputers, or one bank with many ports as used by superscalar microprocessors. To extend the range of vector processing, I propose a vector flag processing model which enables speculative vectorization of "while" loops. To improve the performance of inexpensive vector memory systems, I introduce *virtual processor caches*, a new form of primary vector cache which can convert some forms of strided and indexed vector accesses into unit-stride bursts.

Professor John Wawrzynek
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

I first thank my advisor John Wawrzynek for supporting me throughout this work and for his advice and encouragement while I pursued my interest in vector architectures. I also thank Nelson Morgan for his advice and support over the years, and for providing the initial impetus for the development of the T0 vector microprocessor.

The T0 project was an incredibly fulfilling experience made possible by the exceptional talents of the core implementation team: Jim Beck, Bertrand Irissou, David Johnson, and Brian Kingsbury. Thank you all for persevering. Many others helped out with T0 and earlier projects along the way including Joe Chung, Bernt Habermeier, John Hauser, Randy Huang, Phil Kohn, John Lazzaro, Joel Libove, Thomas Schwair, Balraj Singh, and Su-Lin Wu. Also, thanks to the speech groups at ICSI and at other sites for giving the Spert-II boards a reason to exist, and thanks to the other students and visiting postdocs who performed various T0 application studies including Chris Bregler, Arno Formella, Ben Gomes, Todd Hodes, Tim Kanada, Paola Moretto, Philipp Pfärber, and Warner Warren.

I thank David Patterson for agreeing to be on my dissertation committee. Apart from detailed feedback on the thesis, I also thank Dave for his invaluable career advice and for setting a terrific example as both a teacher and a research project leader. The last few years of my thesis work benefited greatly from interactions with Dave and rest of the Berkeley IRAM group. I thank David Wessel for many fascinating late night conversations at the Triple Rock, as well as for his support and encouragement as part of my dissertation committee. I also thank David Hodges for finding time to be part of my Quals committee.

Thanks to John Hauser for many illuminating discussions, particularly regarding floating-point and exception handling. Thanks to Corinna Lee and the folks at Cray Research for providing feedback and moral support for work in vector architectures.

Thanks to Kathryn Crabtree and Theresa Lessard-Smith for helping me to navigate the tortuous complexities of graduate life at U. C. Berkeley.

The International Computer Science Institute (ICSI) provided the environment for much of this work. I'd particularly like to thank the ICSI computer system administrators, Bryan Costales and Sean Brennan, for keeping the machines running while we hammered away with our chip simulations. I'd also like to thank other members of the larger CNS-1 project and the ICSI community for discussions that helped shape parts of this work including David Bailey, Joachim Beer, Tim Callahan, Jerry Feldman, Karlheinz Hafner, Paul Mehring, Sven Meier, Silvia Müller, Stephan Murer, Nate McNamara, Steve Omohundro, Stelios Perissakis, Heinz Schmidt, Mike Shire, and David Stoutamire.

Special thanks to all the good friends who've made my stay in Berkeley so enjoyable, particularly Jeff Bilmes, David Johnson, the Hobbits, and the `hopheads`.

Funding for my graduate work came from a number of sources including ICSI, ONR URI Grant N00014-92-J-1617, ARPA contract number N0001493-C0249, NSF Grant No. MIP-9311980, and NSF PYI Award No. MIP-8958568NSF.

Finally, extra special thanks to Jim Beck and Su-Lin Wu for their help in the last few panicked

minutes before I filed the thesis!

Chapter 1

Introduction

Ever since their introduction over twenty five years ago, vector supercomputers have been the most powerful computers in the world. Recently, however, microprocessor-based systems have approached or even exceeded the performance of vector supercomputers on some tasks and at much lower costs. Modern microprocessors have superscalar architectures which appear more flexible than the vector model and so many now believe that vector machines are a dying breed, being pushed aside by the astonishingly rapid evolution of these "killer micros"[Bro89].

But the improved cost/performance of microprocessor-based systems is due primarily to their use of commodity silicon CMOS fabrication technology. What if we implement vector architectures using the same technology? In this thesis, I argue that the resulting *vector microprocessors* may actually be the fastest, cheapest, and most energy-efficient processors for many future applications. My argument is based on two claims. First, that these future compute-intensive tasks will contain an abundance of data parallelism. And second, that vector architectures are the most efficient way of exploiting that parallelism.

Over the last decade, there has been little published research into vector architecture compared with the large body of work on superscalar and VLIW architectures. Further, most of that vector research has concentrated on supercomputing applications and small enhancements to existing vector supercomputer designs. This thesis is a step towards expanding the body of vector research, examining designs appropriate for single-chip CMOS vector microprocessors targeting a broader range of applications.

Thesis Overview

Chapter 2 provides motivation and background for this work. I review vector instruction sets and show that they are a compact, expressive, and scalable method for executing data parallel code. I also discuss how vectors can be applied to much wider range of tasks than traditional scientific and engineering supercomputing, including new application areas such as multimedia and human-machine interface processing. I then show that most of the rapid improvement in microprocessor performance on non-vector code over the last decade can be ascribed to technology improvements rather than microarchitectural innovation. I also present results that show that modern superscalar microprocessors are inefficient at driving existing memory systems. Because of the expense of supporting increasing degrees of scalar instruction-level parallelism, current superscalar designs cannot manage enough parallelism to saturate available pin bandwidth and memory bandwidth while tolerating memory latencies. By adding a vector unit to a microprocessor design, we sacrifice a small improvement in scalar performance for a large improvement in performance on data parallel code. The resulting vector microprocessor has different characteristics than existing vector supercomputers. In particular, intra-CPU latencies are much better while memory bandwidths are relatively worse.

Chapter 3 describes the design and implementation of T0: the first single-chip vector microprocessor. T0 is a highly parallel processor that can sustain over 24 operations per cycle while issuing only a single 32-bit instruction per clock cycle. The reduction in startup overhead made possible by a single-chip implementation allows T0 to achieve high efficiency with vector instructions that last only four clock cycles. The duration of a vector instruction in clock cycles is known as a *chime*, and these short chimes are a key difference from previous vector supercomputer implementations.

The next few chapters build on the experience of the T0 implementation and propose designs for various components within future vector microprocessors. Chapter 4 describes pipeline designs for vector microprocessors and shows how short chimes simplify the implementation of virtual memory. The chapter also discusses pipeline designs that can mostly hide memory latency even with short chimes. An important topic is techniques to handle the cases where memory latency is exposed to software. Chapter 5 describes vector register files and presents the design of a compact vector register file suitable for a full-custom VLSI vector microprocessor implementation. Chapter 6 presents a vector flag processing model which supports vector speculative execution in addition to the traditional use of flags for conditional execution. Vector speculative execution expands the range of vector processors by enabling the vectorization of loops containing data-dependent loop exits ("while" loops) while preserving correct exception behavior. Chapter 7 discusses the implementation of vector arithmetic units, including IEEE floating-point and support for narrower data types, as well as comparing vector instruction set extensions with commercial multimedia extensions.

Chapter 8 discusses the implementation of vector memory systems. The chapter includes a review of previous studies that have measured vector memory access patterns at the level of individual instructions. I then identify higher level access patterns contained in sequences of vector memory instructions and discuss how these can be used to improve the performance of memory systems. Chapter 9 introduces virtual processor caches, a new type of vector cache which can take advantage of certain forms of these higher level vector memory access patterns to reduce address and data bandwidth demands. In particular, virtual processor caches can convert some forms of strided and indexed vector accesses into unit-stride bursts. This reduction in address bandwidth is particularly important for low cost memory systems. Chapter 10 describes vector instructions that allow communication between element positions without passing through memory.

Chapter 11 evaluates the T0 design by presenting performance results for several applications which have been ported to T0. The applications are drawn from a wide range of mostly non-supercomputing applications, with an emphasis on tasks likely to form the bulk of future workloads, including multimedia and human-machine interface tasks. Chapter 12 analyzes these application ports to extract statistics to help guide future vector microprocessor designs.

Chapter 13 concludes the thesis, summarizing its contributions and suggesting future work.

Chapter 2

Background and Motivation

Microprocessors have adopted many architectural techniques originally developed for earlier supercomputers and mainframes including pipelining, caching, branch prediction, superscalar, and out-of-order execution [HP96]. But commercial microprocessors have not adopted vector units, even though vector supercomputers were introduced over twenty five years ago and remain the fastest computers for many tasks.

All current high-performance microprocessors [Gwe94a, VKY$^+$96, CDd$^+$95, Gwe96c, Gwe95b, Gwe95c] have superscalar architectures, where hardware dynamically extracts *instruction-level parallelism* (ILP) from a single instruction stream [Joh91]. Because superscalar architectures can extract parallelism from both vectorizable and non-vectorizable code and because the cost/performance of microprocessors has been improving at a dramatic rate, the prevailing belief in the computer architecture community today is that vector instructions are now redundant. In this chapter, I argue that vector architectures retain unique advantages and deserve further study, particularly when implemented in the form of single-chip vector microprocessors.

Section 2.1 compares various forms of machine parallelism, including instruction-level parallelism, thread-level parallelism, and vector data parallelism. Although vectors are the *least* flexible form of machine parallelism they suffice to capture the parallelism present in many tasks, and their inflexibility makes vectors the *cheapest* form of machine parallelism.

Section 2.2 reviews vector instruction sets and shows how they package large numbers of parallel operations into single short instructions. The regular nature of vector instructions allows implementations with simple control logic and compact datapaths to sustain high levels of *operation-level parallelism*.

Of course, vector instructions can only improve throughput on vectorizable code. Section 2.3 discusses how many new compute-intensive applications outside of the traditional areas of scientific and engineering supercomputing are also amenable to vector execution. With the increasing emphasis on multimedia processing and intelligent human-machine interfaces, it is likely that the fraction of vectorizable code in computing workloads will grow.

As well as determining the benefit of including a vector unit for codes that can be vectorized, it is important to determine the penalty for codes that cannot. The main impact of adding a vector unit is the additional die area required, which could otherwise be used to improve scalar performance. Section 2.4 examines the performance of superscalar processors on SPECint95, a set of benchmarks with low levels of vectorization. The results show that sophisticated superscalar microarchitectures require a large growth in die area but yield only a modest improvement in performance on these types of code. Because it is expensive to scale superscalar hardware to support more parallelism and because it becomes increasingly difficult to find more ILP, further increases in die area yield ever diminishing returns in scalar performance. In the worst case, when an application has no vectorizable code, allocating die area for a compact vector unit should have only a minor impact on scalar performance relative to a machine that uses the additional die area for scalar processing. In practice, even "scalar" codes often contain some level of data parallelism that can be exploited by vector units.

Because vector execution is effective at removing other obstacles to high performance on data parallel code, raw functional unit throughput is exposed as the primary constraint. The memory system often limits vector performance because memory bandwidth is expensive to provide. Although it is widely believed that current superscalar microprocessors are often constrained by pin and memory bandwidths, I show that existing microprocessors can use only a fraction of available bandwidths; *pin bandwidths and memory bandwidths are not the bottleneck!*

This surprising result is a consequence of the tremendous cost of managing parallelism expressed solely with scalar instructions, each of which usually describes only a single primitive operation. Because of this cost, current superscalar microprocessors cannot exploit enough parallelism to maintain high throughput while tolerating large main memory latencies. For data parallel code, vector units are an inexpensive way of managing large numbers of parallel operations and hence allow vector machines to saturate available bandwidths. This should allow vector microprocessors to provide higher throughput from existing microprocessor memory systems. Recent developments in high-bandwidth DRAMs, described in Section 2.5, enable low-cost high-bandwidth but high latency memory systems which should further increase the relative advantage of vector processing.

Portable computing is becoming increasingly important, and Section 2.6 discusses how vectors could provide improvements in energy efficiency, by simultaneously improving throughput while reducing switched capacitance per operation.

Section 2.7 summarizes the main arguments made in this chapter.

2.1 Alternative Forms of Machine Parallelism

Computer architects have employed various forms of parallelism to provide increases in performance above and beyond those made possible just by improvements in underlying circuit technologies. Pipelining [HP96, Chapter 3] is the simplest form of machine parallelism and is now universally applied in all types of computing system. Beyond simple pipelining, there are several ways in which processor designs can exploit parallelism to improve performance. Figure 2.1 illustrates the three major categories:

- **Instruction-level parallelism** (ILP) is where multiple instructions from one instruction stream are executed simultaneously. Superscalar machines dynamically extract ILP from a scalar instruction stream.

- **Thread-level parallelism** (TLP) is where multiple instruction streams are executed simultaneously. Multiprocessor machines exploit TLP by scheduling multiple instruction streams onto separate processors.

- **Vector data parallelism** (DP) is where the same operation is performed simultaneously on arrays of elements. Vector machines exploit DP by executing multiple homogeneous operations within one vector instruction at the same time.

These various forms of machine parallelism are not mutually exclusive and can be combined to yield systems that can exploit all forms of application parallelism. For example, the NEC SX-4 vector supercomputer [HL96] is a pipelined superscalar vector multiprocessor architecture which can exploit ILP, TLP, and DP.

Data parallelism is the *least* flexible form of machine parallelism. Any parallelism that can be expressed in a DP manner can also be expressed using ILP or TLP. For example, a superscalar processor can perform DP operations by scheduling multiple independent scalar instructions to execute in parallel, while a multiprocessor can perform DP operations by dividing elements among separate parallel instruction streams. But it is precisely this inflexibility that makes DP the *cheapest* form of machine parallelism. A data parallel

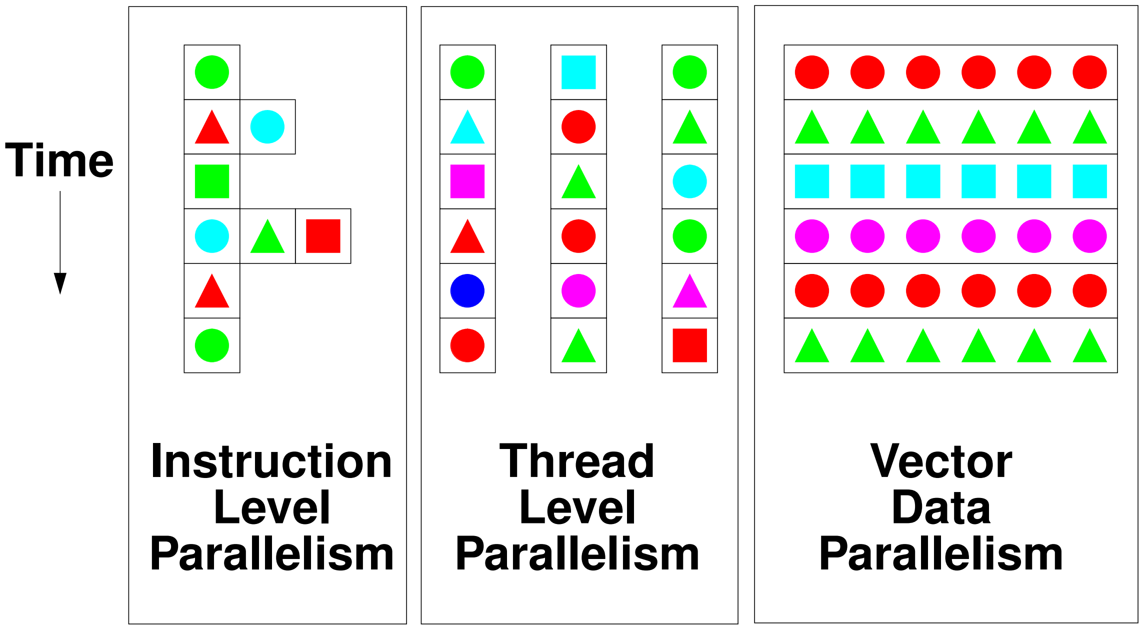

Figure 2.1: Different forms of machine parallelism. Within each diagram, each box represents one instruction, and each shape within a box represents one operation. Boxes are grouped by instruction stream.

machine need only fetch and decode a single instruction to describe a whole array of parallel operations. This reduces control logic complexity, while the regular nature of vector instructions allows compact parallel datapath structures.

Vector machines scale well to more highly parallel implementations. A single instruction management unit can be amortized over greater numbers of independent parallel datapaths. ILP machines scale poorly because the possible interactions between concurrent instruction grows quadratically with the number of parallel instructions. TLP machines scale better than ILP machines, but incur the expense of duplicating instruction management logic for each instruction stream. TLP machines also suffer overheads for inter-thread synchronization and communication. For these reasons, machines without vector support should have poorer cost/performance on data parallel codes compared to vector machines.

2.2 Vector Processing

This section reviews the vector processing model and vector instruction set architectures (ISAs) in general, and defines some terms used in the rest of the thesis. Patterson and Hennessy have published a longer introduction to vector processors [HP96, Appendix B].

Most current microprocessors have *scalar* instruction sets. A scalar instruction set is one which requires a separate opcode and related operand specifiers for every operation to be performed. VLIW (Very Long Instruction Word) [Fis83] machines typically also have scalar instruction sets, with multiple separate scalar operations packed together into one long instruction.

Vector processors provide vector instructions in addition to scalar instructions. A vector instruction specifies operand vectors and a vector length, and an operation to be applied element-wise to these vector operands. For example, a vector addition instruction would take two vectors A and B, and produce a result vector C:

$$C_i = A_i + B_i, \qquad i = 0, ..., \mathrm{VL} - 1$$

where VL is the vector length. A single vector instruction specifies VL independent operations.

In this thesis, the term *scalar processor* is used to describe both a system that can only execute scalar instructions and the scalar component of a vector processor. To avoid confusion, the term *vectorless processor* is used when describing a system that does not provide vector instructions.

2.2.1 Vector Memory-Memory versus Vector Register Architectures

There are two main classes of vector architecture distinguished by the location of the vector operands. Vector memory-memory architectures, such as the CDC STAR 100 [HT72] and successors [Lin82], provide instructions that operate on memory-resident vectors, reading source operands from vectors located in memory and writing results to a destination vector in memory. Vector register architectures, including the Cray series and all of the supercomputers from Japan, provide arithmetic instructions that operate on vector registers, reading source operands from vector registers and writing results to vector registers, while separate vector load and store instructions move data between vector registers and memory.

Vector register architectures have several advantages over vector memory-memory architectures. A vector memory-memory architecture has to write all intermediate results to memory and then has to read them back from memory. A vector register architecture can keep intermediate results in the vector registers close to the vector functional units, reducing temporary storage requirements, memory bandwidth requirements, and inter-instruction latency. If a vector result is needed by multiple other vector instructions, a memory-memory architecture must read it from memory multiple times, whereas a vector register machine can reuse the value from vector registers, further reducing memory bandwidth requirements. For these reasons, vector register machines have proven more effective in practice, and I restrict discussion to vector register architectures for the rest of this thesis.

2.2.2 Vector Register ISA

The user programming model of a typical vector register architecture is shown in Figure 2.2. The scalar unit of a vector processor is similar to a conventional vectorless processor. It contains some number of general purpose scalar registers, which may be sub-divided into special-purpose sets, e.g., integer and floating-point, or address and data. The scalar processor instruction set defines the usual complement of scalar instructions that operate on scalar registers and that transfer data between scalar registers and memory. The vector machines described in this thesis are based on the scalar MIPS RISC ISA [Kan89].

The vector unit contains some number of vector data registers, each comprised of VLMAX elements, where VLMAX is a power of 2. A special control register, the vector length register (VLR), is used to set the number of elements processed by each vector instruction. VLR can be set to values between 0 and VLMAX. In addition, some machines may provide a separate set of vector flag registers which have a single bit per element and which are used to control conditional execution.

Vector arithmetic instructions take source values from vector registers and write results to a vector register. In addition, *vector-scalar* instruction forms are provided that replace one operand with a scalar value, e.g.,

$$C_i = A_i \; op \; S, \qquad i = 0, ..., \mathrm{VL} - 1$$

For non-commutative operations, such as subtraction, a *scalar-vector* form may also be provided,

$$C_i = S \; op \; A_i, \qquad i = 0, ..., \mathrm{VL} - 1$$

The vector-scalar and scalar-vector forms could be synthesized with a single "copy scalar to vector" instruction followed by a regular vector-vector instruction, but this would require an extra instruction and an extra temporary vector register. These operations are used frequently enough to justify both forms.

Vector memory instructions transfer values between memory and a vector register. There are three ways in which the elements can be addressed. If the elements are contiguous in memory, then this is termed a *unit-stride* access. If the elements are separated by a constant displacement then this is a *strided* access.

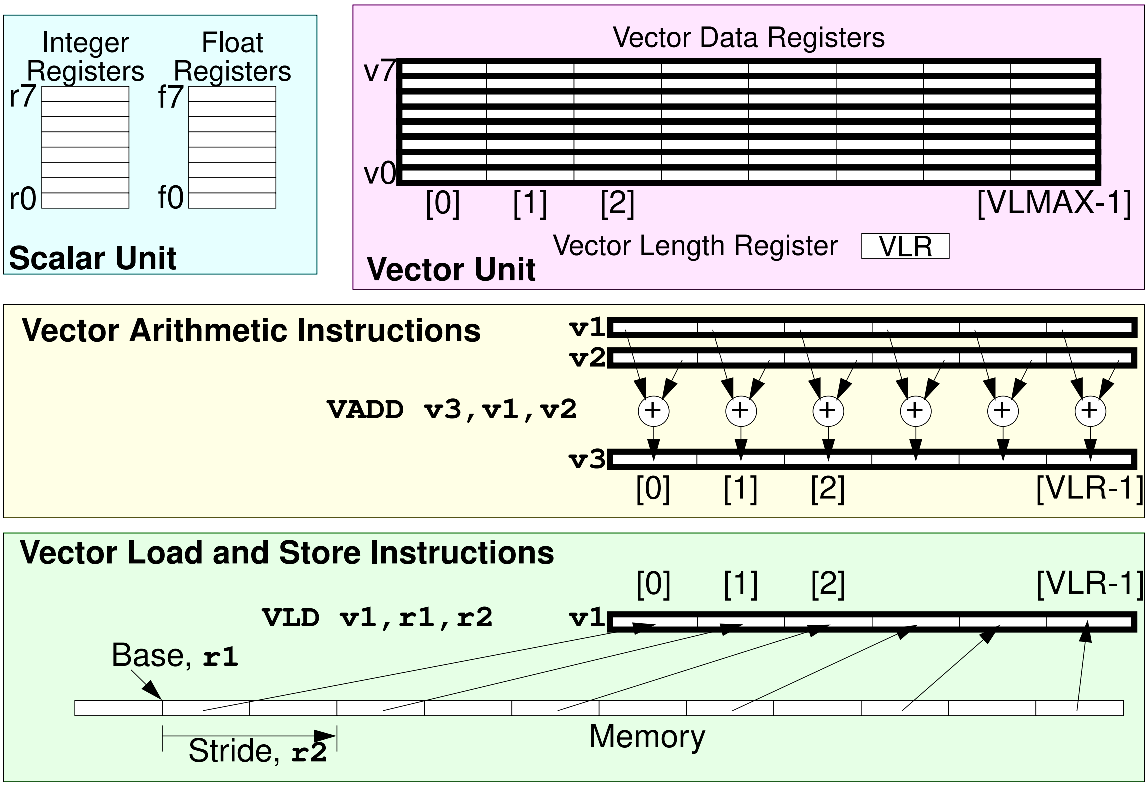

Figure 2.2: Typical user programming model of a vector register architecture. In addition to the standard scalar integer and floating-point registers, there is a bank of vector registers. In this case there are eight vector data registers VR0–VR7 each with a capacity of VLMAX elements. Vector arithmetic instructions perform element-wise operations on vector registers while vector load and store instructions transfer data between vector registers and memory. A vector addition instruction is shown as an example of a vector arithmetic instruction, while a strided load is shown as an example of a vector memory instruction. The effective length of a vector instruction is set by writing the VLR register.

Unit-stride accesses are obviously just a special case of strided accesses and are sometimes handled as such in vector ISAs, but, because they occur so often and are amenable to special case handling, I retain the distinction between *unit* and *non-unit* stride. In *indexed* accesses, the elements may be located at arbitrary locations in memory with the addresses of the elements indicated by the contents of a second vector register. This is also known as *gather* or *scatter* in its load or store forms respectively.

Masked vector instruction execution is usually provided to allow vectorization of loops containing conditionally executed statements. A mask vector controls the element positions where a vector instruction is allowed to update the result vector. The mask vector may be held in one or more special flag or mask registers, or may be held in another vector register. If there is more than a single flag register, then instructions may be provided to perform boolean operations between flag values, or alternatively the scalar unit can operate on packed flag bit vectors.

Instructions are usually provided to allow the scalar processor to access single elements within a vector register. These are useful for partially vectorizable loops. A common example is where a loop contains memory accesses that can be vectorized but where the computation contains a dependency between loop iterations that requires scalar execution.

Compress and expand are another commonly implemented pair of vector instructions. The compress instruction compacts elements at locations indicated by a flag register from a source vector register to contiguous elements at the start of a destination vector register. The packed elements can then be processed with subsequent vector instructions more efficiently than if masked instructions were used throughout. The expand operation is used to insert contiguous elements from the start of a vector register to locations indicated by a flag register in the destination vector register. Expand can be used to write elements selected with a compress back into the original vector. An alternative way to implement the functionality of compress, as used on Crays, is to calculate a compressed index vector based on the flag bits, then use a gather to bring in the compressed data. Similarly, scatters can be used to perform expands.

The above instructions represent a basic vector ISA, and are sufficient to vectorize most loops. To increase the range of loops that can be vectorized, various other more specialized instructions have been proposed and implemented on vector machines.

2.2.3 The Virtual Processor View

While the above gives the traditional view of a vector register architecture, an important alternative view is to regard the vector unit as a collection of *virtual processors* (VPs) [ZB91], as shown in Figure 2.3. The number of VPs is equal to the maximum vector length (VLMAX). This view treats the vector register file as an array of local register files, one per VP. Each vector instruction performs a SIMD (Single Instruction Multiple Data) [Fly66] operation, one operation per VP.

These processors are "virtual" because their operations are time multiplexed across the available physical processing elements in the vector unit. For example, the Cray-1 [Rus78] vector unit can be viewed as containing 64 virtual processors, whose addition instructions are time multiplexed over a single physical

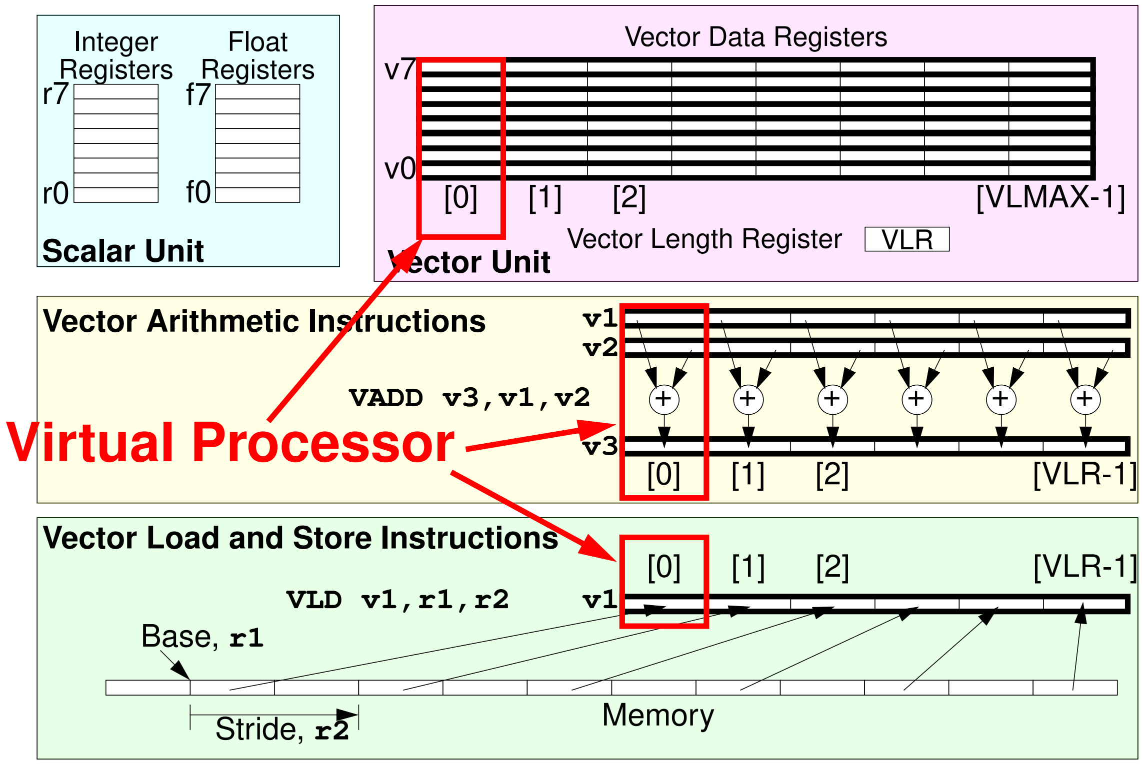

Figure 2.3: Virtual processor view of the vector machine shown in Figure 2.2. This view treats a vector machine as a collection of VLMAX virtual processors, each of which contains a set of scalar data registers. Vector instructions specify SIMD operations across the array of virtual processors.

hardware addition pipeline.

Throughout this thesis I use either the traditional or the virtual processor view of a vector unit depending on the topic under discussion.

2.2.4 Vector ISA Advantages

A vector ISA packages multiple homogeneous, independent operations into a single short instruction. The result is *compact*, *expressive*, and *scalable* code.

The code is *compact* because a single short vector instruction can describe N operations and address $3N$ register operands. This dramatically reduces instruction bandwidth requirements. Moreover, many of the looping constructs required to iterate a scalar processor over the N operations are implicit in the vector instruction, reducing instruction bandwidth requirements even further.

The code is *expressive* because software can pass on much valuable information to hardware about this group of N operations. In particular:

- **These N operations are homogeneous.** This saves hardware in the decode and issue stage. The opcode is decoded once, and all N operations can be issued as a group to the same functional unit.

- **These N operations are independent.** This avoids hardware for dependency checking between the N operations within one instruction. The N operations can be executed using an array of N parallel functional units, or a single very deeply pipelined functional unit, or any intermediate configuration of parallel and pipelined functional units.

- **These N operations will access elements within the vector registers in a regular pattern.** For example, a vector multiply instruction can only multiply element i of one source vector register by element i of a second source vector register, and must store the result in element i of the destination vector register. A subsequent vector add will have similar restrictions on the location of its operands. This constraint allows a highly parallel vector unit to be implemented as multiple independent *lanes* each containing one portion of the vector register file and an associated group of functional unit pipelines. Section 3.4 describes the implementation of lanes in the T0 vector microprocessor. This regular access pattern also allows a highly multi-ported vector register file to be constructed inexpensively from multiple interleaved storage banks with fewer ports per bank as described later in Section 5.2. Another advantage of this regular access pattern is that dependencies between instructions need only be checked once per vector register operand, not for every elemental operand within each instruction. This dramatically reduces the amount of control logic required to manage large numbers of parallel operations.

- **Vector memory instructions touch N operands in a known pattern.** A memory system can implement important optimizations if it is given accurate information on the reference stream. In particular, a stream of N unit-stride accesses can be performed very efficiently using large block transfers. Chapters 8 and 9 describe how knowledge of various other vector memory access patterns can be used to improve memory system performance.

The code is *scalable* because performance can be increased by adding parallel pipelines, or costs reduced by decreasing the number of pipelines, while retaining full object-code compatibility. Figure 2.4 compares execution of the same vector addition instruction on two different vector functional unit (VFU) implementations. The first VFU has a single pipeline, while the second VFU has four pipelines. The component operations of a vector instruction are striped across the available pipelines and peak throughput improves linearly with the number of pipelines. For a VFU with P parallel pipelines, on clock cycle 0 of an instruction's execution, operation 0 begins execution in pipeline 0, operation 1 begins execution in pipeline 1, ..., and operation $P - 1$ begins execution in pipeline $P - 1$. On clock cycle 1, operation P begins execution in pipeline 0, operation $P + 1$ in pipeline 1, and so on. I use the term *element group* to refer to the set of elements that are processed at the same time by a set of parallel pipelines. While most element groups will have P elements, an element group may contain fewer than P elements if it is the last element group from a vector instruction with a vector length that is not a multiple of P.

One final important advantage is that vector instructions can be added as extensions to existing standard scalar instruction sets. This preserves software investment and enables a smooth transition to vector code.

2.3 Applications

To what extent can future workloads can be vectorized? This is the single most important question regarding the viability of vector machines. Although the success of vector supercomputers has proven that scientific and engineering applications often contain substantial portions of vectorizable code, there has been little investigation of vectorizability outside this domain.

Due to the lack of inexpensive vector machines, there has been little incentive to write non-supercomputing codes with vectorization in mind. It is therefore unsurprising that many existing codes appear unvectorizable at first. But, as shown in Chapter 11, many of the compute-intensive tasks envisaged in future workloads can be readily vectorized. In particular, the growing interest in multimedia and intelligent human-machine interfaces seems likely to cause a continuing growth in the fraction of desktop and portable computing workloads which can be vectorized.

Databases are commercially the most important application for large servers. While there has been little work in vectorizing database applications, a significant fraction of this workload also appears to be vectorizable. Sorting [ZB91] and hash-join operators [Mar96] are vectorizable. Database mining is highly compute-intensive, and involves applying multiple statistical techniques to features extracted from database entries. Many of the basic statistical algorithms, including neural networks (Section 11.5), are vectorizable. Databases are now being extended to manage richer media types including video and audio. Querying such

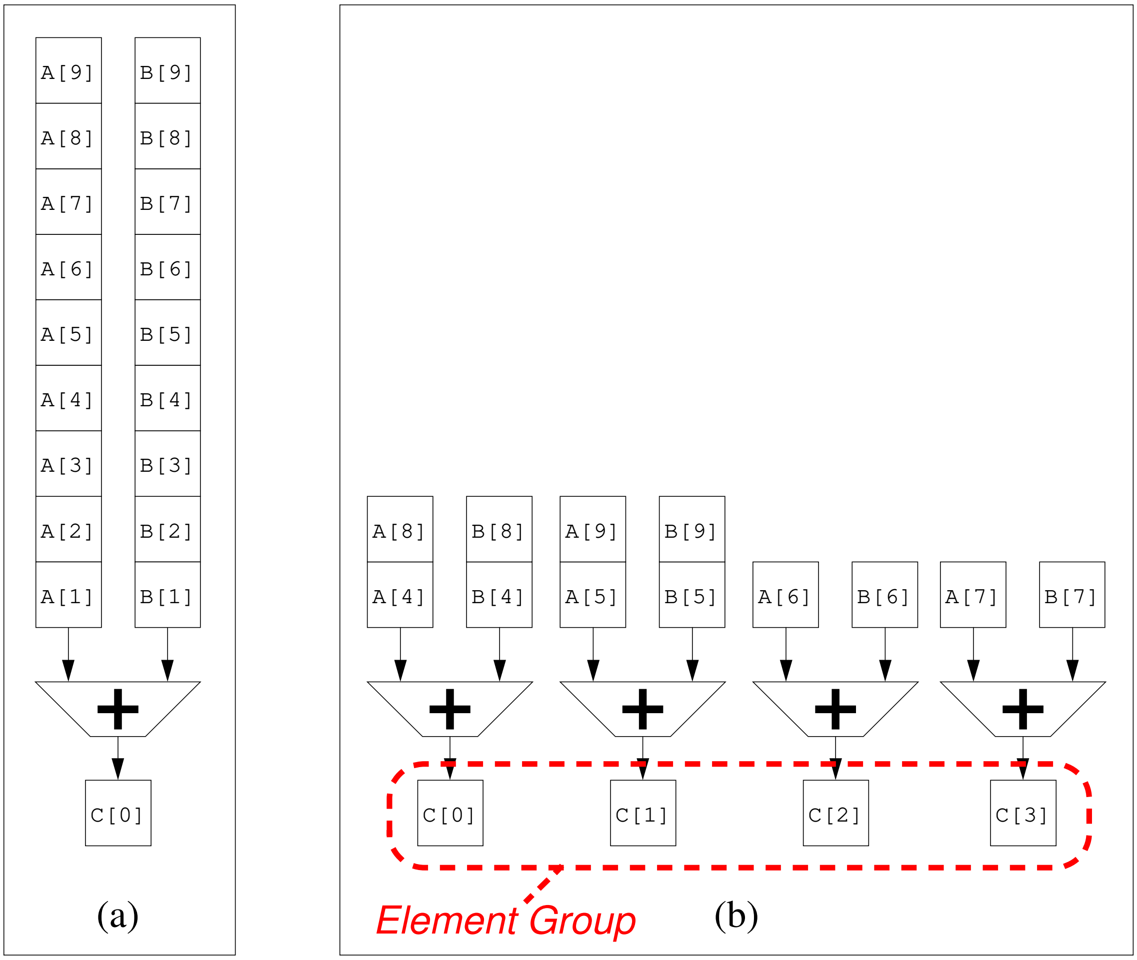

Figure 2.4: Execution of the same vector addition instruction (C = A + B) in two different vector functional units. The first unit (a) has a single pipeline and completes one operation per cycle, while the second unit (b) has four parallel pipelines and completes up to four operations per cycle. An *element group* is the set of elements that are processed at the same time but in different pipelines.

databases involves various forms of pattern recognition and many such algorithms are also vectorizable.

2.3.1 Software Effort

Even when application tasks are vectorizable, a further issue is the software effort required to gain the benefit of vector execution. One lesson learned from the scientific and engineering communities' experience in using vector supercomputers is that many "dusty-deck" codes cannot be *automatically* vectorized as written. Old programs often require some degree of annotation, modification, or even a complete rewrite. Nevertheless, many scientific and engineering applications have been successfully converted to vector form. The incentive for this coding effort was to obtain efficient code for the fastest computers available.

If inexpensive vector microprocessors of the type described in this thesis become commonplace, the effort of vectorizing software should be rewarded in several ways:

- Vector architectures should remain the highest-performance, lowest-cost, and most energy-efficient way to execute data parallel tasks.
- Vectorized source code should provide portable high performance across a range of different vector machines.
- Vectorized source code can scale to take advantage of future vector execution engines with increased numbers of parallel pipelines.

Vectorization can be accomplished incrementally. If vector extensions are added to a standard scalar instruction set, then existing scalar binary code can run unchanged. Even in this case, a vector unit can improve performance by accelerating dynamically-linked libraries and operating system functions. Unmodified source code can be recompiled with a vectorizing compiler to yield some degree of performance improvement through automatic vectorization. Further improvements in vectorization will require source code modifications, ranging from programmer annotations through to major restructuring.

If a new application is being written, the combination of mature vectorizing compilers with a vector execution engine offers a straightforward performance-programming model with a clear relationship between code structure and execution time.

2.3.2 The Continuing Importance of Assembly Coding

While the above discussion and much contemporary computer architecture research focuses on applications written in a portable high-level language, for some tasks a significant fraction of time is spent executing routines written in assembly code. Such routines are written in assembly because they are performance-critical and because modern compilers do not generate optimal code. Examples include database software, desktop productivity applications, and libraries for scientific, graphics, multimedia, and cryptographic operations. Indeed, during the recent spate of multimedia extensions for existing ISAs, companies have assumed coding will be performed in assembler and thus have no plans to provide full compiler support.

While it is currently unfashionable to discuss ease of assembly coding in instruction set design, a vector ISA does have advantages over other highly parallel alternatives. Compared with assembly programming of wide out-of-order superscalar architectures, or tightly-coupled multiprocessors, or VLIW machines with exposed functional unit latencies, vectors offer a compact, predictable, single-threaded assembly programming model with loop unrolling performed automatically in hardware. Moreover, scheduled vector object code can directly benefit from new implementations with greater parallelism, though achieving optimal performance on new microarchitectures will likely require some rescheduling.

2.4 Scalar Performance

The industry standard SPECint benchmarks are often used to measure performance on scalar integer code. In the ten year period between 1986 and 1996, the integer performance of microprocessors as measured by the highest performing system on these benchmarks has improved over 100 fold, or roughly 60% per year [HP96].

It is difficult to apportion credit for this dramatic improvement among advances in device technology, circuit design, instruction set architecture, memory systems, CPU core microarchitecture, and compiler technology. But in this section, I show that sophisticated superscalar microarchitectures have had only a modest impact when compared to these other factors.

In Table 2.1, I compare the performance of two systems from Silicon Graphics Incorporated (SGI): the SGI O2 using the QED R5000 microprocessor, and the SGI Origin 200 using the MIPS R10000 microprocessor. The R5000 is an in-order dual-issue superscalar processor, but it can only issue a single integer instruction per cycle and the processor stalls during cache misses [Gwe96e]. In contrast, the R10000 has a sophisticated microarchitecture which can issue up to four integer instructions per cycle, with register renaming, speculative and out-of-order execution, and non-blocking primary and secondary caches [Yea96, VKY$^+$96]. The R10000 can only execute and retire up to three integer instructions per cycle.

The more sophisticated microarchitecture of the R10000 incurs a considerable expense in die area, design time, and energy consumption. The R10000 requires over 3.4 times the die area of the R5000 as well as using an extra metal layer. If we ignore caches, memory management units, external interfaces, and pad ring, and consider only the CPU core circuitry, the area ratio is closer to 5. Most of the area increase is due to the complex structures required to support speculative and out-of-order superscalar execution. Approximately 122 mm^2 out of the 162 mm^2 of CPU logic is devoted to instruction fetch, decode, register renaming control, and the instruction window components. The integer and floating-point datapaths, including the area for the renamed register file and bypass circuitry, occupy only around 40 mm^2.

The two machines are otherwise very similar. Both microprocessors implement the MIPS-IV ISA, are built in 0.35 μm CMOS processes, and are available at the same clock rate. They have the same on-chip

	SGI O2 R5000 SC	SGI Origin 200	Origin/O2
Processor Die Details			
Processor	R5000	R10000	
Clock Rate (MHz)	180	180	1
ISA	MIPS-IV	MIPS-IV	
Process Technology (μm)	0.35	0.35	
Metal layers	3	4	1.33
Polysilicon layers	1	1	
Total Die Area (mm^2)	87	298	3.43
CPU Die Area (mm^2)	$\approx$33	$\approx$162	4.9
Power Supply (V)	3.3	3.3	
Maximum Power at 200 MHz (W)	10	30	3
Out-of-order execution?	No	Yes	
Branch prediction?	No	Yes	
L1 caches (I/D KB)	32/32	32/32	
L1 associativity	2-way	2-way	
L1 non-blocking?	No	Yes	
Integer instructions per cycle	1	3	3
System Details			
L2 capacity (off-chip, unified, KB)	512	1024	2
L2 associativity	direct mapped	2-way	
L2 non-blocking?	No	Yes	
SPEC disclosure date	October '96	November '96	
Compilers	MIPSPRO 7.1	MIPSPRO 7.1	
SPECint95 (peak)	4.82	8.59	1.78
SPECint95 (base)	4.76	7.85	1.65

Table 2.1: Comparison between the QED R5000 and MIPS R10000 microprocessors. The R5000 area number is for the version manufactured in a single-poly process with 6 transistor cache SRAM cells, some vendors manufacture the R5000 in a dual-poly process with smaller 4 transistor cache SRAM cells [Gwe96e]. The CPU area numbers are estimates obtained from annotated die micrographs, and exclude area for clock drivers, caches, memory management units, and external interfaces.

L1 cache organization with split 32 KB two-way set-associative instruction and data caches. Both systems are built by the same company and benchmark results were reported at almost the same time using the same compiler. The R10000 system has a faster, larger, and more set-associative L2 cache.

The SPECint95 benchmark results show that the R10000 system, even with a superior L2 cache, is only 1.65–1.78 times faster than the R5000 system. The R5000 has an integer microarchitecture very similar to that of one of the earliest commercial RISC microprocessors, the MIPS R2000 which shipped in 1986 running at 8 MHz. The R5000 shipped in 1996 with a clock rate of 200 MHz, a factor of 25 improvement in clock frequency. It is clear that superscalar microarchitecture has had far less impact than simple clock frequency scaling over this decade.

This large expenditure of die area for a sophisticated superscalar microarchitecture gives poor return in performance because it is difficult to extract ILP from these codes. While microarchitectures and compilers will continue to improve, future increases in die area will likely result in diminishing returns for this class of codes.

2.4.1 Scalar Performance of Vector Supercomputers

The scalar performance of vector supercomputers has improved much more slowly than that of microprocessor systems. The Cray-1 was the fastest scalar processor at the time it was introduced, but in a recent study [AAW+96], a 300 MHz Digital Alpha 21164 microprocessor was found to be as much as five times faster than a 240 MHz Cray C90 vector supercomputer on non-vectorized codes taken from the SPECfp92 suite.[1] Table 2.2 lists the performance data from the study and Figure 2.5 plots the speedup of the C90 over the 21164. On the vectorized codes, the C90 was up to four times faster even though the SPECfp92 benchmarks fit into the caches of the Alpha 21164 [GHPS93, CB94b].

There are many factors that contribute to the Alpha's superior scalar performance. The 21164 has both a higher clock rate and shorter latencies in clock cycles through almost all functional units[2]. The 21164 can issue up to four instructions in one cycle whereas the C90 cannot issue more than one instruction per cycle and requires multiple cycles to issue many common scalar instructions including scalar memory references and branches. The 21164's larger primary instruction cache and underlying cache hierarchy combined with instruction prefetching reduces the impact of instruction cache misses. The C90 experiences full memory latency on every miss in its small primary instruction cache, and has no instruction prefetching beyond that provided by long instruction cache lines. Even on cache hits, extra delay cycles are incurred when switching between the eight banks of the C90 instruction cache. The Alpha architecture has 32 integer and 32 floating-point registers accessible by any instruction, while the Cray architecture has only 8 primary address and 8 primary scalar registers, with a further 64 address backup and 64 scalar backup registers that require additional instructions to access. The 21164 has a data cache hierarchy which reduces the impact of memory latency for applications with smaller working sets. The C90 has no scalar data cache (apart from the explicitly managed

[1] At the time of writing, faster versions of both architectures are in production, the 600 MHz Digital 21164A, and the 460 MHz Cray T90.

[2] Detailed functional unit timings for the C90 are considered Cray proprietary and are not reproduced here [Cra93].

System Details			
System	Cray C90	Digital 8400	
Processor	Cray C90	Alpha 21164	
Clock Rate (MHz)	240	300	
Peak 64-bit MFLOPS	960	600	
Peak MIPS	240	1200	
Benchmark	Benchmark Times (seconds)		C90 speedup
`spice2g6`	498.6	102.5	0.21
`doduc`	15.0	4.9	0.33
`fpppp`	75.8	14.1	0.19
`ora`	32.8	19.7	0.60
`mdljdp2`	21.1	15.6	0.74
`wave5`	14.2	11.8	0.83
`mdljsp2`	21.7	14.1	0.65
`alvinn`	4.8	8.0	1.67
`nasa7`	10.5	26.5	2.52
`ear`	12.9	20.0	1.55
`hydro2d`	6.2	23.8	3.84
`su2cor`	4.3	17.6	4.09
`tomcatv`	1.0	3.7	3.70
`swm256`	8.2	29.0	3.54

Table 2.2: Performance results for SPECfp92 [AAW+96]. Two results were reported for the Alpha in this study, the vendor-optimized SPECfp92 disclosure and results using the experimental SUIF compiler, and here I use the better number for each benchmark. The Cray results were not optimized by the vendor and could potentially be further improved.

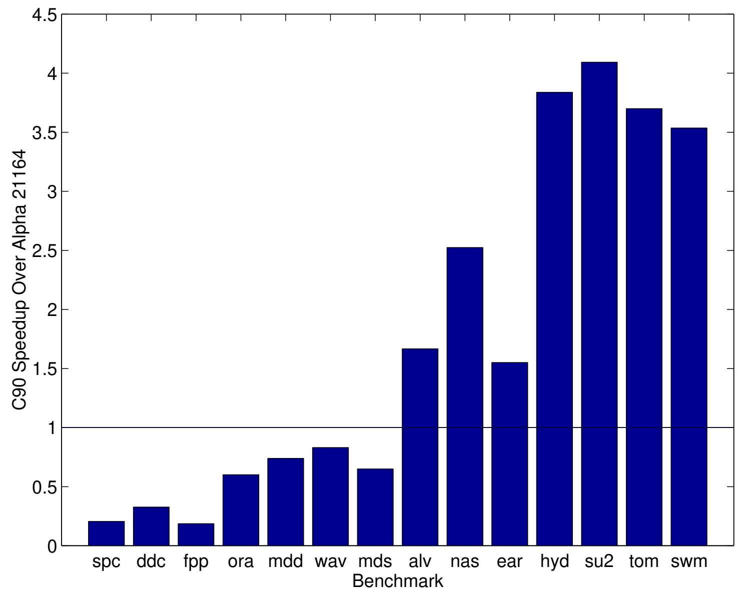

Figure 2.5: Speedup of a single Cray C90 processor against single Alpha 21164 processor on the SPECfp92 benchmark suite [AAW+96].

scalar backup registers) and experiences the full memory latency (over 20 cycles) on every scalar load.

The Cray vector supercomputers suffer from the need to keep backwards compatibility with an ISA designed two decades earlier, while the Alpha ISA benefits from recent research into the interaction of scalar CPU design and compilation technology. The Cray supercomputers are built with a low-density gate array technology that requires the CPU be split across multiple chips. In contrast, the 21164 is a full-custom design implemented in high-density CMOS process, which enables a high-performance scalar CPU plus the upper levels of the memory hierarchy to be integrated together on the same die, significantly lowering latencies.

2.4.2 Scalar versus Vector Performance Tradeoff

If we base a vector architecture on a modern scalar ISA and implement this in a modern CMOS process, we can expect the scalar core of the resulting vector microprocessor to have scalar performance comparable to that of a conventional scalar microprocessor built in the same die area. But adding a vector unit takes away die area that could otherwise be used to increase scalar performance.

The R10000 SPECint95 results above suggest that it is difficult to improve scalar performance significantly even with large increases in die area. At some point, depending on the workload, adding a vector unit to give large increases in throughput on data parallel code should be more attractive than continuing to add hardware to give incremental speedups on scalar code. Furthermore, in Chapter 11, I present results for vectorizing SPECint95 which show that *vector units can improve cost/performance even for codes with low levels of vectorization*.

2.5 Memory Systems

By removing other bottlenecks, vector architectures expose raw functional unit throughput as the primary limitation on vector performance. As device density increases, it is straightforward to increase vector arithmetic throughput by replicating pipelines. Balancing this arithmetic throughput with sufficient memory system bandwidth is more difficult, particularly in low-cost systems which must use DRAM main memories.

In Section 2.5.1, I use numbers from the STREAM benchmark to show that current superscalar microprocessors make inefficient use of available pin and memory bandwidths, even on code that does nothing except copy memory. Because of the expense of managing scalar instructions, existing superscalar machines cannot exploit enough parallelism to saturate pin bandwidths and memory bandwidths while tolerating main memory latency. Even with current memory system designs, a vector machine could potentially improve application memory bandwidth by supporting enough parallelism to saturate pins and DRAMs.

Section 2.5.2 describes advances in high-bandwidth DRAMs that should enable even greater memory bandwidths at low cost, while Section 2.5.3 describes the possibility of removing pin and memory bandwidth bottlenecks completely by moving main memory on to the same die as the processor.

```
C     Copy kernel                C    Sum kernel
      DO 30 j = 1,n                   DO 50 j = 1,n
          c(j) = a(j)                     c(j) = a(j) + b(j)
 30   CONTINUE                    50 CONTINUE

C     Scale kernel               C    Triad kernel
      DO 40 j = 1,n                   DO 60 j = 1,n
          b(j) = scalar*c(j)              a(j) = b(j) + scalar*c(j)
 40   CONTINUE                    60 CONTINUE
```

Figure 2.6: STREAM benchmark kernels [McC95].

2.5.1 STREAM Benchmark

The STREAM benchmark [McC95] measures the sustainable application memory bandwidth during long unit-stride vector operations within the four FORTRAN kernels shown in Figure 2.6. Performance on this benchmark correlates well with performance measured on certain scientific applications [McC].

The data in Table 2.3 is for three multiprocessor servers [McC97b], and two parallel vector processor systems. I use multiprocessor systems in this analysis because measurements made by running multiple processors simultaneously can saturate the shared memory system and yield the maximum sustainable memory bandwidth. Each server is built around a shared backplane interconnect, with all memory modules equidistant from all processors. Cache coherence across all processors and memory modules is managed by bus snooping hardware. The Sun UE 6000 system has two processors on each system board sharing a single 1.328 GB/s connection to the backplane.

The third column gives the peak *data pin bandwidth* for the CPU. This is the maximum rate at which data can cross the chip boundary. The 300 MHz 21164 processors in the AlphaServer 8400 have a 16-byte L3 cache bus that cycles at a 75 MHz clock rate [FFG+95]. The 75 MHz R8000 processors in the Power Challenge have a 16-byte L2 cache bus cycling at 75 MHz. The R8000 is split into two chips with the separate floating-point unit chip connected directly to the L2 cache [Gwe93]. All the floating-point loads and stores used in the STREAM benchmark do not access the L1 cache on the integer unit chip. The 167 MHz UltraSPARC processors in the Sun UE 6000 system have a 16-byte L2 cache bus cycling at 167 MHz.

The fourth column gives the maximum aggregate application memory bandwidth sustained on any one kernel of the STREAM benchmark when using all of the given number of CPUs running in parallel. This is a measure of the usable bandwidth of the shared memory system on these machines. The write allocate policy of the caches in the Digital and SGI systems implies that each destination vector cache line has to be read before being immediately overwritten. This raises the actual physical memory traffic by up to 50% over that measured by the STREAM kernels, but we use the same measurement for multiple and single processors, so this effect cancels out when we take ratios. The Sun results use VIS assembly code to bypass the cache on writes to the destination vector.

The fifth column gives the maximum application memory bandwidth sustained by a single CPU running the STREAM benchmark with other processors idle. In all cases, a single CPU can only exploit a small fraction of both its available pin bandwidth and the available shared memory system bandwidth. The ratio of sustained single CPU application memory bandwidth to single CPU data pin bandwidth varies between 11.2–16.5%. The ratio of sustained single CPU application memory bandwidth to sustainable memory system bandwidth varies between 14.3–20.3%.

These numbers suggest that current microprocessors cannot efficiently convert available CPU data pin bandwidth and available memory system bandwidth into sustained application memory bandwidth. The primary reason is that current microprocessors cannot control enough parallelism to tolerate latencies to main memory.

The vector processors shown in Table 2.3 sustain a much greater fraction of raw processor pin bandwidth. The J90 processor has two memory pipelines each of which can move one 8-byte word per cycle, giving a peak memory bandwidth of 1600 MB/s. The J90 processor delivers up to 1442 MB/s STREAM bandwidth, 90% of its peak pin bandwidth. This is over a factor of seven greater than the Alpha 21164 processor in the Digital 8400, despite both machines having a memory system built from commodity DRAMs with roughly the same system cost per CPU [SB95]. The memory pipelines in a C90 processor can move 6 words per cycle, for a peak of 11.5 GB/s, and delivers 9.5 GB/s STREAM bandwidth, 82% of peak pin bandwidth. Compared to the microprocessor systems, the C90 has a much higher cost due to the use of SRAM for main memory.

System	CPUs	Single CPU Data Pin Bandwidth (MB/s)	Maximum Aggregate STREAM Bandwidth (MB/s)	Maximum Single CPU STREAM Bandwidth (MB/s)
Digital 8400	8×300 MHz Alpha 21164	1,200	978.8	198.3
SGI PowerChallenge	8×75 MHz MIPS R8000	1,200	749.3	134.9
Sun UE 6000 (VIS assembly code)	16×167 MHz UltraSPARC	2,672	2,551.0 (Maximum to one CPU is 1,328)	366.8
Cray J916	8×100 MHz J90	1,600	10,274.4	1,441.7
Cray C916	16×240 MHz C90	11,520	105,497.0	9,500.7

Table 2.3: STREAM benchmark results. These results show that commercial microprocessors can only use a fraction of the pin and memory system bandwidth available to them. The Sun UE6000 bus structure limits the shared bandwidth available to one CPU. Vector processors, such as the Cray J90 and Cray C90, sustain a much greater fraction of peak processor pin bandwidth.

A popular fallacy is that superscalar microprocessors would deliver vector-like performance if given high-performance vector memory systems [HP96, Appendix B.8]. But the STREAM results (Table 2.3) demonstrate that superscalar microprocessors make poor use of *existing* low-bandwidth memory systems. The problem is that the more flexible parallel execution model of the superscalar processors comes at considerable cost, which limits the number of parallel operations that can be supported and hence the memory latency that can be tolerated. In contrast, Espasa [EV96] presents simulation results for vector machines which show that, even with relatively simple microarchitectures, they can attain high memory system efficiencies while tolerating 100 cycle memory latencies.

2.5.2 High Performance DRAM Interfaces

With each DRAM generation, capacity has increased by a factor of four while cost per bit has only dropped by a factor of two [Prz96]. Because memory sizes are often constrained by cost, the number of DRAMs per system halves with each DRAM generation [PAC+97]. This exacerbates the bandwidth problem by reducing the number of separate DRAMs across which memory accesses can be interleaved.

Driven by the needs of graphics adaptors, which demand high bandwidth from small memories, various high-performance DRAM interfaces have been developed, including synchronous DRAM (SDRAM), Rambus DRAM (RDRAM), MoSys DRAM (MDRAM), and SyncLink (SLDRAM) [Prz96]. These can deliver hundreds of megabytes per second of raw main memory bandwidth per device when driven with an appropriate stream of memory references.

One outstanding demonstration of the capabilities of these new high-performance DRAMs is the Nintendo N64 video game [Top97] currently selling for less than $200. The N64 has greater main memory bandwidth than most contemporary workstations. The single 562.5 MHz Rambus DRAM channel in the N64 can provide sustained unit-stride bandwidths of around 390 MB/s while requiring only 31 pins on the memory controller. As of September 1997 [McC97b], the leading microprocessor figures for STREAM were the IBM RS6000 family led by the 591 model at 800 MB/s, and the Fujitsu HAL family with the HAL385 at 523 MB/s. Other systems, such as those from Digital, HP, Sun, SGI, and Intel, achieve well under 400 MB/s. Note that STREAM operates on long unit-stride vectors — an access pattern that should allow any high-performance DRAM to attain peak transfer rates.

A further example is the low-cost SGI O2 workstation, which is built around a single bank of synchronous DRAM memory used both for graphics acceleration as well as for CPU main memory [Kil97]. This memory system has a 256-bit data bus cycling at 66 MHz providing up to 2.1 GB/s of main memory bandwidth. Unfortunately, the R5000 processor in the O2 only manages to deliver a paltry 70 MB/s on STREAM [McC97a].[3]

There is a natural synergy between these new DRAM technologies and vector architectures. DRAMs can provide high bandwidth but at high latencies, and for optimum performance they require that as many op-

[3] The SGI O2 also supports a higher performance R10000 processor, but a bad interaction between the R10000 and the O2 memory controller limits STREAM memory bandwidth of this configuration to around 55 MB/s [McC97a].

erations as possible are performed on one DRAM row before the row address is changed. Vector architectures can saturate the available memory bandwidth while tolerating high latency, and vector memory instructions naturally group scores of related accesses to improve spatial locality. Even where DRAM accesses remain the bottleneck, the vector memory unit should be able to deliver reference streams which maximize the sustained bandwidth.

2.5.3 IRAM: Processor-Memory Integration

As the number of DRAMs per system continues to drop, an increasingly large fraction of applications will have working sets that fit onto a single DRAM. This observation motivates the development of IRAM technology which integrates processor and DRAM together on the same die [PAC+97]. With the removal of the pin bottleneck, attainable bandwidths may increase 50-fold while latency is reduced 5-fold.

Vectors appear a natural match to IRAM technology [KPP+97]. An IRAM processor must be small to leave room for DRAM, but must also make good use of the increased memory bandwidth to justify development of the technology. A vector processor is a compact device that can nevertheless convert available memory bandwidth into application speedup.

2.6 Energy Efficiency

While the previous arguments have concentrated on performance, another increasingly important factor in computer system design is energy consumption. As their capabilities improve, portable computing devices, such as laptops, palmtops, and personal digital assistants (PDAs) are becomingly increasingly popular. Unfortunately, battery technology is improving much more slowly than CMOS technology [Rei95], so emphasis is now being placed on architectural techniques that exploit increased transistor counts to reduce power [CCB92]. Even for line-powered computers, lowering energy consumption is important to reduce die packaging and cooling costs, to improve ergnomics by eliminating fan noise, and to support “green” computing.

Energy consumption is a measure of how much energy is required to perform a certain task. The lower the energy consumption, the greater the battery life in a portable device. But most simple techniques to lower energy consumption also lower performance, and response time may be as important as energy per operation for portable computers. Burd and Broderson [BB96] show that Energy to Throughput Ratio (ETR) is an appropriate measure for the energy efficiency of a low-power processor operating in either maximum throughput mode, or burst mode with some form of idle mode power reduction:

$$\mathrm{ETR} = \frac{\mathrm{Energy}}{\mathrm{Throughput}} = \frac{\mathrm{Power}}{\mathrm{Throughput}^2}$$

ETR is a measure similar to MIPS^2/Watt or the energy-delay product. The power consumption in a well-designed CMOS circuit can be approximated as:

$$\mathrm{Power} = {V_{\mathrm{DD}}}^2 \cdot f_{\mathrm{CLK}} \cdot C_{\mathrm{EFF}}$$

where f_{CLK} is the system clock frequency, V_{DD} is the supply voltage, and C_{EFF} is the effective capacitance switched each cycle [CCB92].

One way to increase energy efficiency is to increase the parallelism present in an microarchitecture [CCB92]. With perfect parallelization, if N operations are completed in parallel, the energy consumption per operation will be unchanged, but throughput will increase, hence ETR improves. In practice, making hardware more parallel incurs overheads which reduce ETR.

For example, an in-order superscalar processor exploits parallelism but increases C_{EFF} per instruction. In the issue stages, multiple fetched instructions must be aligned to multiple instruction decoders, inter-instruction dependencies must be checked, and multiple instructions must be dispatched over an instruction crossbar to the multiple functional units. In the execute stage, there are larger multiported register files and more capacitative bypass busses between multiple functional units. Also, the larger size of the core increases clock distribution energy.

An out-of-order processor further increases C_{EFF} per instruction. Register renaming, buffering instructions in the instruction window before issue, arbitrating amongst ready instructions for issue, broadcasting results back to the instruction window, buffering completed results in the reorder buffer before commit, and comparing load addresses to bypass outstanding stores in the store buffer, all cause additional energy overhead.

Both in-order and out-of-order superscalar processors almost invariably perform speculative execution, which also increases the C_{EFF} overhead per completed instruction, by adding branch prediction hardware and by performing and buffering operations whose results will later be discarded.

For there to be an improvement in ETR, the speedup obtained from these techniques must be greater than the increase in C_{EFF} per operation [BB96]. There is evidence that even for a limited dual-issue superscalar processor, the speedup is matched by increased C_{EFF} overhead [GH95]. Because C_{EFF} tends to increase quadratically with issue width while speedup increases sublinearly, it is unlikely that more aggressive superscalar designs will improve ETR.

In contrast, a vector machine has the potential to both increase performance *and* reduce C_{EFF} per operation. One distinct advantage of adding a vector unit is that it can impose little overhead on purely scalar code. If the clocks to the vector unit are gated off when there are no vector instructions executing, the only additional energy consumption should be a few gates in the instruction decoder checking for the presence of new vector instructions. In contrast, aggressive superscalar architectures consume extra energy per instruction *even when there is little parallelism to exploit*.

While a complete quantitative analysis of the energy efficiency of vector architectures is beyond the scope of this thesis, it is possible to highlight several areas where there are potential savings in C_{EFF} overhead:

- **Instruction fetch.** Perhaps the most obvious reduction is in instruction fetch, decode, and dispatch. For vectorizable code, a vector unit drastically reduces the number of instruction fetches. Vector instructions also remove much of the interlock and dispatch logic overhead. To illustrate the potential

Unit	Power
I-cache	27%
I-box	18%
D-cache	16%
Clock	10%
IMMU	9%
E-box	8%
DMMU	8%
Write Buffer	2%
Bus Interface	2%
PLL	1%

Table 2.4: Breakdown of power consumption of Digital SA-110 StrongARM processor when running Dhrystone [MWA+96].

for energy savings, Table 2.4 shows a breakdown of the power consumption of the Digital SA-110 StrongARM microprocessor while running the Dhrystone benchmark [MWA+96]. If we add together the power consumed in the instruction cache, the instruction MMU, and the instruction decode (I-box), we see that 54% of the power is consumed in fetching and decoding instructions.

- **Register file access.** Because operations within a vector instruction access the vector register file in a regular pattern, a high-bandwidth vector register file can be built from smaller, fewer-ported banks (Chapter 5). In contrast, a superscalar architecture with its flexibility to access any combination of registers for any operation requires full multiported access to the entire register file storage.

- **Datapath data.** Because vector instructions group similar operations, it is likely that there is much greater bit-level correlation between successive elements in a vector than between successive instructions executed in a vectorless processor. This should reduce datapath switching energy [BB96].

- **Datapath control lines.** Because a vector functional unit executes the same operation on a set of elements, datapath control signals are only switched once per vector. This should reduce C_{EFF} compared to a vectorless architecture where different types of operation are time multiplexed over the same functional units, and hence datapath control lines are toggled more frequently.

- **Memory accesses.** Vector memory operations present regular access patterns to the memory system, which enables further energy savings. For example, unit-stride vector memory accesses may only require one, or at most two, TLB accesses per vector of operands.

The main sources of potential increases in C_{EFF} per operation are in structures that provide inter-lane communication, including control broadcast and the memory system crossbar. This inter-lane cost can be reduced by using highly vectorizable algorithms that avoid inter-lane communication, and by adding VP caches (Chapter 9) that reduce lane interactions with memory crossbars.

2.7 Summary

This chapter presented a case for including vector units on future microprocessors. Vector instructions are sufficient to represent much of the parallelism present in future compute-intensive applications, such as media processing and intelligent human-machine interfaces. Rather than only adding hardware to provide incremental improvement in scalar performance, we can instead spend some of the hardware budget on a vector unit to attain large speedups on data parallel codes.

Because scalar instructions are an expensive way to implement machine parallelism, current microprocessors cannot exploit enough parallelism to saturate existing pin and memory bandwidths while tolerating memory latency. Vector instructions allow a small amount of logic to control large amounts of operation-level parallelism, thereby enabling the processor to tolerate latency and saturate available memory bandwidths. This facility will become increasingly important both for off-chip memory, where tolerating latency is important, and for high-bandwidth on-chip memory, where providing high throughput is important.

Vector architectures exhibit a natural synergy with IRAM technologies, providing a low-cost, high-performance, energy-efficient processor to match the low-cost, high-bandwidth and energy-efficient on-chip memory system. Vector IRAMs have the potential to become the standard processors for low-cost portable computing systems.

Vector architectures have been commercially available for twenty five years, but only in the form of multi-chip vector supercomputers. This thesis expands the body of vector research by investigating the changes in the design space when vector architectures are integrated on a single die to form low-cost vector microprocessors. The most important changes are that intra-CPU latencies are improved while off-chip memory bandwidths are reduced.

Chapter 3

T0: A Vector Microprocessor

A typical vector supercomputer is built from hundreds of low-density ECL logic chips coupled with several thousand BiCMOS SRAM memory chips, supports multiple CPUs, resides in a purpose-built room, requires a liquid cooling system, and is in continual use running scientific and engineering batch jobs. In contrast, a typical future vector microprocessor will be built on a single high-density CMOS die, possibly coupled with a few commodity DRAM memory chips, and will occasionally be turned on by a single user to run multimedia applications in a battery-powered portable computer. This chapter presents a detailed description of the design and implementation of T0 (Torrent-0), the first complete single-chip vector microprocessor.[1]

Section 3.1 provides the background to the project. Section 3.2 gives a short overview of the Torrent vector ISA implemented by T0, while Section 3.3 is a detailed description of the T0 microarchitecture. Section 3.4 describes the VLSI implementation of T0 and shows how vector processors are well suited to VLSI implementation with a regular compact datapath structure and minimal control logic. Section 3.5 relates the design methodology used to build T0.

Section 3.6 discusses the most important difference between T0 and vector supercomputer designs. Because T0 is a single-chip implementation it has much lower intra-CPU latencies, and can avoid the startup penalties which force designers to use long running vector instructions in vector supercomputers. This allows T0 to use much shorter chimes, with most maximum length vector instructions completing in only four clock cycles. As will be described in the next chapter, short chimes simplify the implementation of virtual memory as well as reduce the size of the vector register file.

3.1 Project Background

The background to the T0 project was a series of systems developed within ICSI, and later in collaboration with U. C. Berkeley, to train neural networks for phoneme probability estimation in speech

[1] The implementation of T0 was joint work with James Beck, Bertrand Irissou, David Johnson, Brian Kingsbury, and John Wawrzynek.

recognition research. The first system was the RAP (Ring Array Processor) [MBK+92], which contained multiple off-the-shelf TMS320C30 DSPs connected in a ring topology. The RAP proved to be much more cost-effective than contemporary workstations for these computationally intensive algorithms. We began investigating application-specific VLSI architectures [AKMW90] to provide even larger cost/performance improvements, but soon realized that the algorithms used in speech recognition research were changing far more rapidly than we could redesign silicon. We then changed our emphasis to more general-purpose programmable processors, and designed a programmable processor named SPERT [ABK+92]. SPERT was a VLIW/SIMD processor that was partially realized in the form of the SQUIRT test chip [WAM93]. For tackling larger problems, we were also interested in building large scale parallel machines using our processor nodes [ABC+93]. Because it exposed the machine organization to software, the VLIW/SIMD SPERT design was difficult to program and would not have been object-code compatible with future enhanced implementations. We began considering the idea of extending a conventional scalar instruction set architecture (ISA) with a vector instruction set to reduce software effort while retaining the advantage of efficient parallel execution of critical kernels. The subsequent development of the Torrent ISA [AJ97] and the T0 implementation [AB97] led directly to this thesis work.

Our goals for the system design were to provide a high-performance workstation accelerator that could be replicated inexpensively. Ideally, the entire system would be just a single internally-mounted card to save the additional engineering work of constructing an external enclosure.

As we began to work through possible system designs, one proposal was to just implement a vector coprocessor attached to a commercial MIPS R3000 core, in a manner similar to that of the Titan graphics supercomputer [DHM+88]. R3000 cores available at the time had clock rates up to 40 MHz with a well-defined coprocessor interface. Although the later MIPS R4000 was also becoming available around this time, it's external interfaces could not have supported a tightly coupled off-chip coprocessor. We rejected this idea of attaching our vector coprocessor to a commercial scalar unit because it would have seriously complicated overall system design, particularly the design of the memory system. The scalar CPU runs out of caches, and the vector coprocessor would require its own wide, fast, and coherent connection to memory around the side of the high speed cache interface used by the scalar unit. This would add considerably size, complexity, and cost to the resulting circuit board. Also, the coprocessor package would have required more pins to connect to both the processor cache bus to receive instructions and the memory system to receive data.

Another drawback would have been the relatively loose coupling between the vector unit and the scalar CPU. The limited coprocessor semantics meant that some vector instructions would have required two or more scalar instructions to issue. This reduction in effective vector issue rate would raise the vector lengths required to sustain peak performance.

A further problem would be the speed of inter-chip signal paths. On a more practical note, our simulation environment would not have been able to simulate the complete system, and so we faced the possibility of requiring several lengthy silicon design revisions to fix bugs.

We decided that T0 would be a complete single chip microprocessor requiring minimal external support logic. Our initial estimates suggested we could make the design fit onto a large die in the process available to us. The scalar MIPS core would not require much die area, and by designing our own scalar unit we retained the maximum flexibility when adding our tightly-coupled vector unit. The high level of integration dramatically simplified the system board design, and allowed us to simulate the complete system's operation using just our own chip simulators. The resulting system fits on a double-slot SBus card which we named Spert-II [WAK+96].

3.2 Torrent Instruction Set Architecture

The goals of the Torrent ISA design [AJ97] were to provide a flexible, scalable architecture suitable for a wide range of digital signal processing tasks. Our first design decision was to base the vector ISA on an existing industry standard scalar ISA. This would enable us to leverage an existing software base for development. We considered several RISC instruction set architectures, including SPARC [WG94], HP Precision Architecture [Hew89], PowerPC [WS94], and Alpha [Sit92], before deciding on MIPS [Kan89]. Our main reasons for selecting MIPS were the simplicity of the architecture, the well-defined coprocessor interface, and the large body of freely available development tools. Other contributing reasons included our own familiarity with the MIPS ISA, the availability of several MIPS-based Unix workstations for development purposes, and the existence of a 64-bit instruction set extension.

We added the vector unit instructions as coprocessor 2 in the MIPS model. Coprocessors 0 and 1 were already in use as the standard system coprocessor and floating-point coprocessor respectively, and coprocessor 3 was used by other MIPS extensions.

Torrent is mostly a conventional vector register architecture [Rus78]. The instruction encoding allows for 32 vector registers, but T0 only implements 16 to save area. Vector register zero is hardwired to return zero. The T0 implementation has a maximum vector length of 32, with each element holding 32 bits.

Vector memory instructions are divided into separate unit-stride, strided, and indexed classes. Unit-stride operations could have been encoded as strided operations with a stride of 1. However, the ISA design anticipates that most hardware would be optimized for unit-stride accesses. The separate encoding simplifies dispatch and also allows a post-increment of the scalar base address to be specified in the same instruction. This post-increment is an arbitrary amount taken from any scalar register, and register zero can be specified if no post-increment is required. The post-increment can be executed in the scalar adder as the vector instruction passes down the scalar pipeline. The post-increment avoids a separate scalar add instruction, and so saves instruction issue bandwidth. Without the folded auto-increment, the T0 implementation would have required either longer vector registers or superscalar issue to keep the vector functional units busy during many important loops. Vector memory instructions can load and store 8-bit, 16-bit, or 32-bit values. Both sign- and zero-extended versions of 8-bit and 16-bit loads are provided, and memory store operands are taken from the least significant bits of the vector register element.

Perhaps the biggest difference from a conventional vector ISA is that only fixed-point arithmetic instructions were defined. Our primary intended application domain was neural net training for speech recognition work, and our previous studies [AM91] had shown that 16-bit fixed-point multipliers and 32-bit ALUs were sufficient for this task. A full set of integer arithmetic, logical, and shift instructions operate on the full 32-bit element width. The multiply instruction was defined to multiply the low 16 bits from the source operands producing a 32-bit result. In addition, support for fixed-point arithmetic with scaling, rounding, and result saturation was added. The fixed-point support allows multiple cascaded arithmetic operations to be performed in one vector instruction under control of configuration bits in a separate scalar configuration register.

Another difference from conventional vector ISAs is that conditional execution is performed using conditional move operations with mask values held in regular vector registers (see Section 6.3). In addition, separate flag registers were provided to communicate packed flag bit vectors back to the scalar unit. There are three flag registers: `vcond` reports compare results, `vovf` holds sticky integer overflow flags, and `vsat` holds sticky fixed-point saturation flags.

3.3 T0 Microarchitecture

Figure 3.1 shows an overall block diagram of T0. T0 integrates an integer scalar unit, a 1 KB instruction cache, a vector unit, an external memory interface, and a serial host interface port. The external memory interface has a 128 bit data bus, and a 28 bit address bus, supporting up to 4 GB of industry standard asynchronous SRAM. The T0 serial interface port (TSIP) has eight bit wide datapaths, and provides DMA into T0 memory as well as access to on-chip scan chains for testing and debug. The scalar unit executes the MIPS-II 32-bit instruction set [Kan89], and contains a hardware integer multiplier and divider. The vector unit contains three vector functional units (VFUs), VMP, VP0, and VP1, which communicate via the central vector register file. VMP is the vector memory unit, which executes scalar memory and vector extract instructions as well as vector memory instructions. VP0 and VP1 are two vector arithmetic units, which are identical except that only VP0 contains a multiplier. All three VFUs contain eight parallel pipelines and can produce up to eight results per cycle.

The vector unit is structured as eight parallel *lanes*, where each lane contains a portion of the vector register file and one pipeline for each VFU. The lane is a key concept in vector microarchitecture. A well-designed vector instruction set constrains most communication between vector instructions to occur locally within a lane. This limits the amount of inter-lane communication required, reducing the cost of scaling a vector unit to large numbers of parallel lanes.

3.3.1 External Memory

The memory system is the most critical part of any vector machine. Our 1.0 μm CMOS technology would not allow much on-chip memory, so we required a large capacity, high bandwidth commodity memory part. Fast page-mode DRAM was the highest speed DRAM readily available at the time we began the design in 1992. With a maximum column cycle rate of 25 MHz, we would have required multiple interleaved

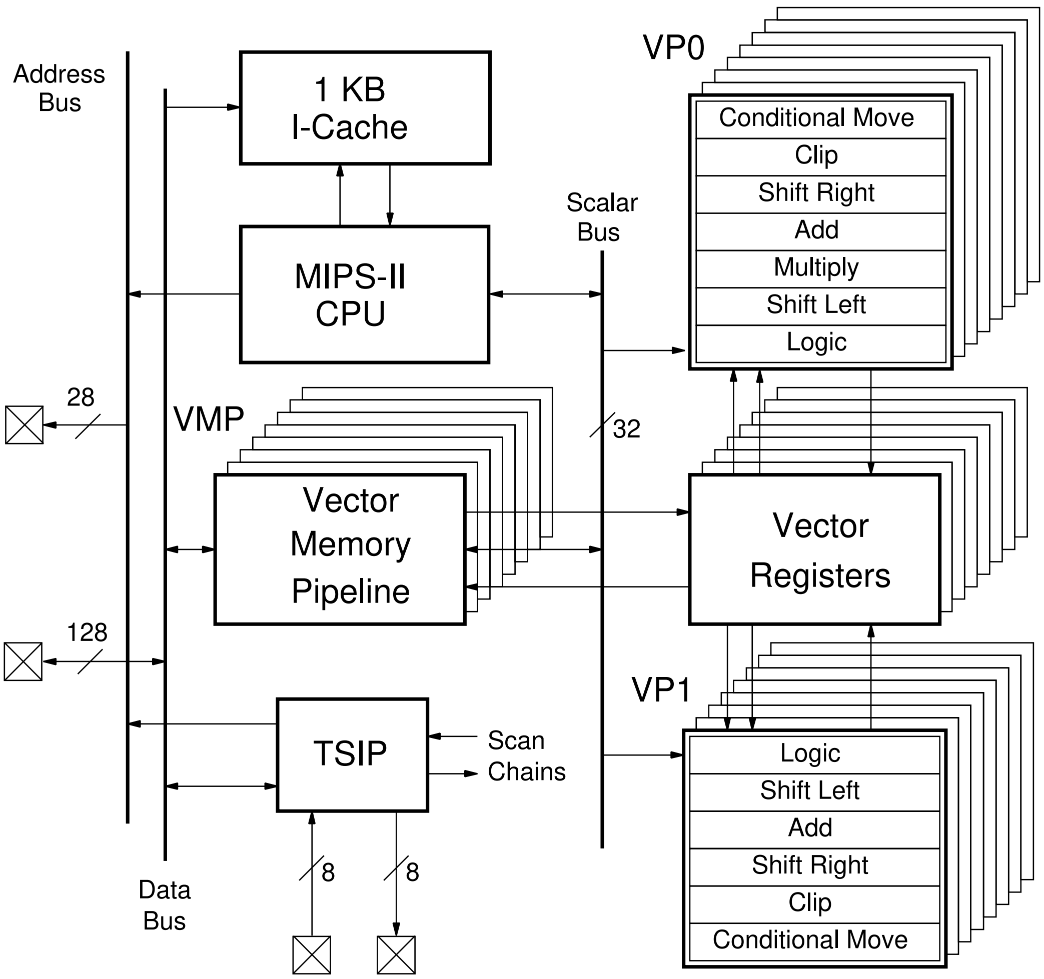

Figure 3.1: T0 block diagram.

DRAM banks to reach our cycle time goal. This would have required external data multiplexing buffers and more complicated control. At the time there was much discussion in the industry about new higher speed DRAM interfaces including EDO, SDRAM, Rambus DRAM, and various graphics RAMs, but it was not clear which of these new options would survive through to the end of our design cycle. The only commodity high-bandwidth memory part we could rely on was SRAM.

Using SRAM would also reduce the design effort required in the memory system, and allow a highly integrated design with few external parts. Several vendors had pre-announced fast 4 Mb SRAM parts which would provide sufficient capacity on a small circuit board. These were pin-compatible with existing 1 Mb parts so that we could prototype early systems with a 2 MB memory systems, and move to an 8 MB memory system as larger parts became available. Vendors had also promised that future 16 Mb SRAMs would also be pin-compatible, allowing a 32 MB memory with the same board design. The external SRAM interface on T0 has a single 28-bit address bus `ma[31:4]`, and a 128-bit wide data bus `md[127:0]`. Each 8-bit byte of the data bus has a separate byte write enable signal, `bwen[15:0]`.

Memory accesses are pipelined over two cycles. During the first cycle, the effective address is calculated and multiplexed out over the external address bus. On the same cycle, a "not-killed" read/write signal (`nkrwb`) indicates if the access could potentially be a write. This signal is used to control the SRAM output enables. When used with conventional asynchronous SRAM, an external address register is necessary to pipeline the address. An external clock generator generates the address register clock, latching the address early to allow wave-pipelining of the asynchronous SRAM access. During the second cycle data is driven out from T0 on a write, or received into on-chip latches for a read. During this cycle the data also crosses the rotate network inside the T0 memory unit.

Asynchronous SRAMs require a write pulse. T0 provides two inputs, `wenb[1:0]`, driven in tandem with the same pulse, that modulate the byte write enables. Two inputs are provided to reduce on-chip wiring delays. The write enable pulse is generated by the external clock phase generator.

3.3.2 TSIP

For fabrication test purposes, we initially adopted the IEEE 1149.1 (JTAG) standard [IEE90] for scan chains. We had successfully used JTAG before on SQUIRT and other test chips. The processor was to be mounted on a board inside a host workstation and so we had to provide interfaces for host control and host data I/O. For host data I/O, one approach would have been to provide a second time-multiplexed port into the external SRAM. This would have provided a high performance path to T0 memory, but would have required multiple external data buffer parts to bridge the host bus to the T0 memory data bus. These external buffers would add electrical load to the memory data bus lines, as well as increase board area and power dissipation. Instead, we expanded the functionality of the JTAG scan chain to allow direct memory access. We kept the same JTAG TAP state machine and protocol, but widened the scan chains from one bit to eight bits to improve I/O bandwidth. We called the new interface TSIP (T0 Serial Interface Port). A shift register is built into the T0 memory system. A parallel path into the shift register can transfer a complete 128-bit block to or from memory in one cycle. The block can then be shifted between T0 and the host at the rate of eight bits per cycle while leaving the memory port free for other accesses. All other chip test, debug, control, and host-T0 synchronization are also performed over TSIP.

3.3.3 Instruction Fetch and Decode

Figure 3.2 shows the pipeline structure of T0. The fetch and decode stages are common to all instructions. The fetch stage (F) is used to access the instruction cache. The 1 KB instruction cache is direct-mapped with 16 byte lines. Although the cache is relatively small, the autonomous fetch stage processing and low refill times reduce the impact of misses. During vector loops, the instruction cache frees the memory port for data accesses. A similar combination of a wide bus to external memory and a small on-chip instruction buffer is used in the HP PA 7100LC design [BKQW95].

During the F stage, the next instruction address is determined and fed to the cache, which returns an instruction by the end of the cycle. If the external memory port is free, the same address is sent off-chip to prefetch the instruction cache line in case there is a cache miss. If there was a prefetch, instruction cache misses take two cycles, otherwise misses take three cycles. The fetch stage operates independently of later stages and so cache miss processing can be overlapped with interlocks and memory pipe stalls. Each cache refill steals only a single cycle from the external memory port, limiting the impact on ongoing vector memory accesses.

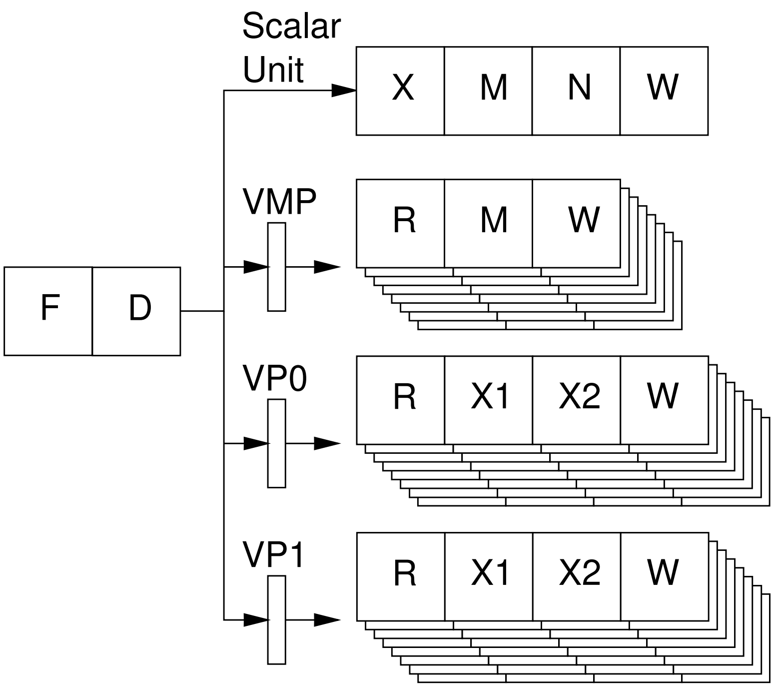

Figure 3.2: T0 pipeline structure. Each of the three vector functional units (VMP, VP0, and VP1) contains eight parallel pipelines.

A more aggressive prefetching scheme might significantly improve instruction fetch performance, especially on scalar code. If the sequentially next cache line was prefetched whenever the memory port was free, there would be no miss penalties on sequential code provided there was at least one memory access slot available every four cycles. Additionally, if the prefetched instructions return in time to avoid stalls, they need not be allocated in the cache, thereby reducing cache evictions [ASPF92]. However, this scheme would have required extra circuitry, some of which might be in critical paths, and was conceived too late in the design cycle to be added to T0.

During the decode stage (D) the scalar register file is read and bypassed and interlocks are checked. If there are no interlocks, the instruction is dispatched to the appropriate unit for execution. If there are interlocks, the instruction is held in the D stage until interlocks clear. All instructions use the scalar pipeline to perform exception checking. Vector instructions may occupy a vector functional unit for multiple cycles.

3.3.4 Scalar Unit

Scalar ALU instructions are executed in the X stage. All integer ALU operations are bypassed with single cycle latency, apart from integer multiplies and divides.

The scalar integer multiply/divide unit contains a separate 34-bit adder, and can operate concurrently with further instruction issue. The multiply/divide unit performs integer multiplies using a 2-bit Booth encoding and takes 18 cycles to complete a 32-bit $\times$ 32-bit $\rightarrow$ 64-bit multiply. Divides produce a single bit per cycle using a non-restoring division algorithm, and take 33 cycles to perform a 32-bit/32-bit $\rightarrow$ 32-bit+32-bit divide. Multiply and divide results are placed in the two special 32-bit registers, `hi` and `lo`.

The logic unit in the scalar datapath evaluates during the end of the D stage, and is used to perform an XOR operation to set up the inputs to the zero comparator used for branch tests. This is one of the critical paths in the processor because the output of the zero comparator is used to control the program counter multiplexor that selects the instruction address to feed to the instruction cache. The zero comparator is implemented as a 32-input precharged NOR gate to speed evaluation.

The fast external memory means that no scalar data cache is required. The external memory is fully pipelined, and returns scalar loads with a 3 cycle latency. A separate address adder is used to generate addresses for all scalar and vector loads and stores. During vector memory operations, the address adder operates concurrently with further instruction execution. Addresses are generated during the first half of the X stage and are sent off chip to an external registered buffer. Load data is returned across the memory system crossbar and into latches within the vector unit by the end of the M stage. The N stage is used to align and sign-extend the data, and to transmit it from the vector memory pipeline datapath to the scalar unit bypass network.

The first half-cycle of the W stage is used to write back scalar results to the register file. These may be read out from the register file on the subsequent half-cycle.

3.3.5 Vector Register File

The T0 vector register file contains 16 vector registers, each holding 32 32-bit elements. Vector register 0 is hardwired to the value zero. The vector register file is split into eight parallel 32-bit wide slices, one for each of the eight parallel lanes in the vector unit. Each lane contains a total of 60 read/write 32-bit registers (15 vector registers with four elements per vector register) and one 32-bit zero register (read or written four times for accesses to vector register zero).

The vector register file provides one read and one write port for VMP, and two reads and a write port for each of VP0 and VP1, making a total of five read ports and three write ports. Each functional unit has dedicated ports, and so there are never any inter-unit vector register access conflicts. To save area, the register file bit cells are time-multiplexed to provide five read accesses when the clock is high, and three write accesses when the clock is low. The time multiplexing scheme adds complexity to the precharge and sensing control logic. To speed reads, it is necessary to precharge bit lines before enabling the row select lines. Also, to avoid spurious writes the row lines must settle to the correct value before the write drivers are turned on. A self-timed circuit controlled by dummy cell rows is used to provide the extra control edges within the single cycle.

The address decoders for the vector register file are also time-multiplexed to reduce area. There are a total of five address decoders. The address decoding is pipelined one clock phase ahead of the bit cell access. When the clock is high, five separate read addresses are decoded and latched into five groups of 61 read-enable pipeline latches. When the clock is low, the decoding is latched into five groups of 61 write-enable pipeline latches. During the write decode phase, pairs of single-ended port decoders are driven with the same address to open a differential write port on the chosen bit row.

3.3.6 Vector Memory Unit

Vector memory instructions are dispatched to the vector memory unit controller at the end of the D stage. An individual vector memory instruction may take many cycles to complete. The controller is a small state machine that generates the pipeline control signals for the remainder of the vector memory pipeline. There are a further three stages: vector register file read R, memory access M, and vector register file write W. The vector memory unit also executes scalar load and store, scalar insert/extract, and vector extract instructions using the same pipeline stages.

In the first half of the R stage, the vector register file read address is determined and passed to the register address decoders. The vector register file is read in the second half of the cycle. The memory rotate network control and byte write enable signals are also calculated during this stage. Scalar store and scalar insert instructions transmit data from the scalar datapath to the vector unit during this cycle. Scalar stores are always performed by lane zero.

During the M stage, store data from different lanes is rotated to the correct byte position by the store alignment crossbar, and then passes out through the pins to the external memory. On a load cycle, data is returned from the external memory, rotated into position by the load alignment crossbar, and then latched in registers within the vector memory pipeline.

During the first half of the W stage, the latched load data is aligned within the datapath, zero- or sign-extended, and written into the vector register file. In the second half of the W stage, scalar data from a scalar load or a scalar extract instruction is transmitted back across the `scbus` to the bypass multiplexors in the scalar datapath. Scalar load data is always returned from lane zero, while scalar extract data is forwarded from the appropriate lane.

The vector memory unit has separate read and write ports into the vector register file. Only a single read or write port is required for most vector memory operations, but a single combined read-write port would not have reduced register file area significantly due to the time-multiplexed vector register file design. The number of bit lines is set by the number of write ports, and the number of address decoders and word lines is set by the number of read ports. Stores and loads access the vector register file at the R and W stages respectively, and separate read and write ports avoid structural hazards that would otherwise occur when switching between back-to-back vector loads and stores. The vector extract instruction uses both the read and the write port simultaneously, and the two ports also simplify vector indexed loads.

The load and store data alignment networks employ a two-stage scheme as described later in Section 8.4 on page 150. All eight 32-bit lanes connect to the 128-bit `md` data bus with a crossbar supporting arbitrary 32-bit word routing. Within each lane, additional rotate networks handle subword accesses. Both the load and store alignment crossbars are composed of three overlaid networks, one each for 8-, 16-, and 32-bit operands. Only the vector extract instruction uses both alignment crossbars simultaneously and then only transfers 32-bit words. This instruction requires that there are separate control lines for load and store alignment of 32-bit words, but for 8-bit and 16-bit operands alignment control lines can be shared between the load and store alignment networks to reduce routing area. The alignment control lines are routed diagonally through the crossbar matrix as in a classic barrel shifter design.

We intended to use an SGI workstation for development and to deploy the completed boards in Sun Sparcstation hosts. Both of these environments are big-endian, and so we adopted a big-endian addressing model for T0. The control lines in the memory alignment crossbars would have to be duplicated to allow bi-endian operation.

Unit-stride loads and stores of 16-bit and 32-bit values transfer 16 bytes per cycle between memory and the vector register file, saturating the external memory bus. Unit-stride loads and stores of 8-bit operands transfer eight elements per cycle limited by the eight available ports into the vector register file, and so only use half the external memory bandwidth. Unit-stride loads and stores may start at any natural alignment of the element type being transferred. The number of cycles taken by the memory pipeline equals the number of 16-byte memory lines that are accessed (8-byte memory lines for 8-bit transfers). Vector unit-stride loads have a two cycle latency, unless the first element is not 16-byte aligned (8-byte aligned for 8-bit transfers), in which case the latency increases to three cycles.

T0 has only a single address port, and so strided and indexed loads and stores move at most a single element per cycle regardless of operand size. The alignment networks rotate data to match each lane with the appropriate bytes of the memory bus.

Vector indexed loads and stores are complicated by the need to transmit indices from the vector register file to the address generator in the scalar datapath. Indexed loads have a start up penalty of three cycles to wait for vector register indices to arrive at the address generator. Indexed stores require both an index and a data value for each element stored and these accesses must be multiplexed over the single VMP vector register read port. The store delay register used for misaligned unit-stride stores is reused as a buffer for vector indices. A group of eight indices is read into the store delay register, adding an extra stall cycle every eight elements. Store indices are passed to the address generator from the store delay register while store data is read directly from the register file. Vector indexed stores are the longest running instructions on T0 and take 38 cycles to complete execution with the maximum vector length of 32.

The vector extract instruction was added primarily to speed reduction operations, but can also used to copy vector registers when the vector memory unit is otherwise idle. Vector extract reads elements from the end of one vector register and writes them to the beginning of a second vector register. The start point for the read is given in a scalar register, while the number of elements to be read is given by the vector length register. Extracts that begin at source elements 0, 8, 16, and 24 do not require inter-lane communication, and are handled with a special sneak bypass path within each lane. These extracts run at the rate of eight elements per cycle. Other extracts move elements at the rate of four elements per cycle across the `md` bus, and are treated as a combination of a vector store and a vector load. The first portion of the memory pipeline places four elements on the internal 128-bit `md`, while the second portion of the memory pipeline reads four elements from `md`. The vector extract instruction allows arbitrary start positions within the source register, and these are handled in the same manner as non-16-byte-aligned unit-stride word loads.

3.3.7 Vector Arithmetic Units

T0 has two vector arithmetic units (VAUs), VP0 and VP1. These are identical except that only VP0 has a multiplier. Allowing VP0 to execute all instructions as well as multiplies did not require a significant area increase, yet provides a large performance improvement on many codes that do not require multiplies but which have a large compute to memory operation ratio. Together, the two VAUs allow T0 to sustain sixteen 32-bit ALU operations per cycle.

Vector arithmetic instructions are dispatched to one of the two VAUs at the end of the D stage. For non-multiply instructions, if both units are free, the dispatch logic issues the instruction to VP1. Otherwise, the dispatch logic issues the instruction to the first available VAU that can execute the instruction.

Vector arithmetic operations continue execution down four further pipeline stages: R, X1, X2, W. The vector register file is read in the second half of the R stage, and written in the first half of the W stage, giving a three cycle latency for dependent arithmetic operations.

The vector arithmetic pipelines operate on 32-bit data, taking two 32-bit operands and producing a 32-bit result plus a single bit flag value. Each VAU has two dedicated vector register file read ports and a single dedicated vector register file write port. Vector arithmetic instructions include a full complement of 32-bit integer arithmetic and logical operations, together with fixed-point arithmetic support. To implement these functions, each arithmetic pipeline contains six functional units: a 32-bit logic unit, a 32-bit left shifter, a 33-bit adder, a 33-bit shift right unit, a 33-bit zero comparator, and a 33-bit clipper. VP0 also contains a 16-bit$\times$16-bit multiplier that produces 32-bit results.

The structure of VP0 is shown in Figure 3.3. Data flows through all functional units for all instructions, with unused functional units set to pass data unchanged. An alternative design would multiplex data around the various functional units, but this would require more area for bypass bussing and multiplexors. A further important advantage of the current scheme is that multiple operations may be performed in a single pass down the vector arithmetic pipeline.

To take advantage of this feature, a set of arithmetic pipeline instructions were defined that allowed multiple operations to be cascaded within a single vector instruction under control of a configuration value held in a scalar register [ABI+96]. This feature was originally added to support fixed-point scaling, rounding, and clipping in one instruction, but was extended to allow any arbitrary combination of the functional unit operations. The arithmetic pipeline instructions have a fourth scalar register argument which contains a configuration value for the arithmetic pipeline. The configuration register contents are decoded to drive the arithmetic pipeline control signals allowing the equivalent of up to 6 basic operations to be performed with a single instruction. By using values held in a scalar register, the cascade of functional units can be dynamically reconfigured on an instruction by instruction basis with no cycle time penalty.

The pipeline can be configured to perform a complete scaled, rounded, and clipped fixed-point operation in a single pass. The pipelines can also be reconfigured to provide other composite operations such as absolute value, max/min, and bit field extract. A complete IEEE single-precision floating-point vector library has been implemented for T0 using the reconfigurable arithmetic pipelines, and this achieves around 14.5 MFLOPS at 40 MHz despite no other hardware support for floating-point arithmetic.

During the first half of the X1 stage, the two inputs to the arithmetic pipeline are multiplexed from the two vector register read ports and the scalar register to give either a vector-vector, vector-scalar, or a scalar-vector operation. The logic unit, left shifter, and multiplier all complete operation during the X1 stage. The logic unit can perform all 16 possible 2-input binary logic functions. The left shifter includes circuitry that shifts in a rounding bit at the 1/2 LSB position. The multiplier is structured as a signed 17-bit$\times$17-bit multiplier producing a 33-bit result. The multiplier inputs are taken from the low 16 bits of each operand and are either sign- or zero-extended to 17 bits. The left shifter is used to add in a rounding bit during multiply instructions. The multiplier produces a sum and carry vector, that is added to the left shifter output using a 3:2 carry save stage. Towards the end of the X1 stage the adder inputs are set up. The adder input circuitry includes sign- or zero-extending logic that extends the 32-bit operands to 33-bit values. The extra bit is used to manage overflow conditions.

At the start of the X2 stage, the precharged adder carry chain evaluates and the adder produces a 33-bit sum value. The adder output is fed to a 33-bit zero comparator. The sign bit and the output of the zero comparator are used to control conditional writes of the result and to control the final clipper unit. The adder output also flows through a 33-bit right shifter. The right shifter can perform either logical or arithmetic shifts. The right shifter also includes sticky bit logic that ORs together all bits that are shifted out. This is

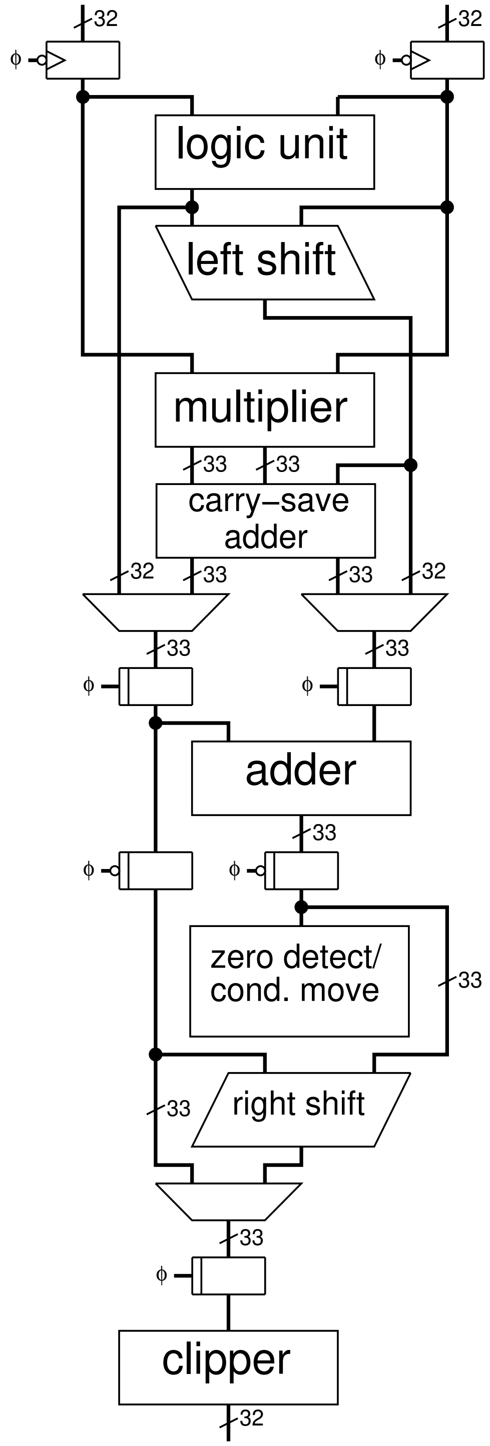

Figure 3.3: Structure of one pipeline in VP0.

used together with a small adder output correction circuit to implement various rounding modes including round-to-nearest-even.

Either the right shifter result, or the logic unit result, is fed to the final clipper unit. The clipper prepares the final 32-bit value that is written to the vector register file, and also produces a single bit flag value that is written to one of the flag registers. The clipper can saturate the 33-bit input to a 32-bit, 16-bit, or 8-bit value. The flag value is set whenever a saturation is actually performed. This flag value is usually OR-ed into the `vsat` flag register. The clipper can also generate boolean values if the input was zero or negative, writing a 0 or 1 to the 32-bit vector register. The corresponding flag values are usually written to the `vcond` register. Finally, the clipper can produce boolean values if the input would overflow a 32-bit signed integer representation. This is used to write the `vovf` flag register during vector signed integer arithmetic operations. The clipper input is set up at the end of the X2 stage, but the wide precharged OR gates in the clipper are evaluated at the beginning of the W stage. The clipper completes evaluation in time to write the vector register file by the end of the first half of the W stage. A write enable signal is generated based on the zero comparator and sign of the adder output. These are used to implement conditional move operations. All eight possible write enable conditions are supported.

3.3.8 Chaining

Vector chaining was first introduced in the Cray-1 [Rus78]. Chaining reduces latencies and allows increased execution overlap by forwarding result values from one vector instruction to another before the first instruction has completed execution. The Cray-1 design had very restricted chaining capabilities because each vector register had only a single port, and so chaining was only possible if the second instruction was issued before results started appearing from the first instruction. Later Cray models added more flexible chaining capability by adding more read and write ports to each vector register.

T0 provides completely independent vector register ports for each VFU and supports chaining for all vector register data hazards: Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW). Chaining RAW hazards reduces latencies and allows greater overlap between dependent instructions. Chaining WAR hazards reduces vector register pressure by allowing a vector register to be reused earlier. Chaining WAW hazards is important to reduce the latency between conditional move instructions that write the same vector register. T0 has no chaining restrictions; chaining may begin at any time after it is first safe to do so. To save area, there are no vector register file bypass multiplexors; chaining occurs through the vector register file storage. The separate read and write ports for each vector functional unit remove vector register access conflicts and allow a single vector register to simultaneously support up to five readers and three writers.

Chaining is controlled by the D stage interlock logic. An instruction is only dispatched when all chaining requirements will be met for the duration of that instruction's execution. If an on-going vector memory instruction stalls due to an instruction cache refill or a host DMA memory access, both VP0 and VP1 are also stalled to preserve the chaining relationships set up at instruction dispatch.

Chaining is supported even between vector instructions that run at different rates. Most T0 vector instructions run at the rate of eight elements per cycle but some vector memory instructions run at slower rates: four elements per cycle for unit-stride word loads and stores and non-4-aligned vector extracts, and one element per cycle for indexed and strided loads and stores. Chaining between instructions that run at different rates is only begun when it is certain that the dependency will be preserved. For example, consider a unit-stride word load which moves four elements per cycle from memory into a vector register and a dependent vector add instruction that reads the elements at the rate of eight elements per cycle. It is safe to start the vector add when it is certain that the last four words will be written by the vector load in the half-cycle before the vector add reads the last eight values. In this case, chaining reduces the latency from nine cycles to six.

3.3.9 Exception Handling

T0 handles several forms of disruption to normal program flow: reset, interrupts (asynchronous exceptions), and synchronous exceptions. Reset and interrupts can be considered to occur *between* instructions. Synchronous exceptions occur *during* execution of a particular instruction. Exceptions are checked at the start of the M stage of the pipeline.

Reset has priority over interrupts which have priority over synchronous exceptions. There are five sources of interrupt, in decreasing order of priority: a host interrupt that can be accessed via a TSIP scan register, the vector unit address error interrupt, the timer interrupt, and two external interrupt pins.

T0 has a 32-bit cycle counter with an associated compare register that can be used to generate timer interrupts. Two input pins, `extintb[1:0]`, provide a fast interrupt path for external hardware. The two pins have separate dedicated interrupt vectors and were intended to support an inter-processor messaging interface.

Synchronous exceptions include address errors, coprocessor unusable errors, reserved instruction errors, syscalls, breakpoints, scalar integer arithmetic overflows, and vector length errors. All synchronous exceptions on T0 are handled in a precise manner, except for address errors on vector memory instructions.

T0 has no memory management unit, and so performs no translation from virtual to physical address spaces. For its intended use as a workstation accelerator, a full MMU would have added little value, but would have added considerable design effort. To help debugging, a simple address protection scheme is implemented which restricts user virtual addresses to the top half of the virtual address space (`0x8000000`–`0xffffffff`). This check only requires inspection of the top bit of the user virtual address, but catches many stray pointer errors including the common case of dereferencing a `NULL` pointer. Address alignment errors in scalar and vector memory operations are also trapped.

Vector address errors can occur many cycles after a vector load or store instruction is dispatched, and after a chained arithmetic operation has started to write results back to the vector register file. T0 doesn't support virtual memory, and so there is no need to restart a process after it experiences a vector memory address error. To simplify the implementation, these synchronous exceptions are not handled precisely but instead cause a fatal asynchronous interrupt to be raised.

The vector length exception is raised if a vector instruction is issued when the contents of the vector length register are larger than the maximum allowed. This trap was added to aid debugging and to support upward compatibility with future implementations that might have longer vector registers.

The remaining synchronous exceptions were added to allow the T0 kernel to provide a standard MIPS programming environment. Instructions for the MIPS standard floating-point coprocessor are trapped and emulated in software on T0. Also, several rarely used MIPS integer instructions (MIPS-I misaligned load/store and MIPS-II trap instructions [Kan89]) were omitted from the T0 implementation and are trapped and emulated by the kernel. Syscall instructions are used to provide a standard Unix OS interface, and the breakpoint instructions are used by the `gdb` debugger.

3.3.10 Hardware Performance Monitoring

T0 has a hardware performance monitoring port with eight pads that bring out internal events so external hardware can provide non-intrusive hardware performance monitoring. The events that are tracked include instruction cache misses, interlocks, memory stalls, exceptions, and the busy status of each vector functional unit.

3.4 T0 Implementation

Figure 3.4 shows a die photo of T0 while Figure 3.5 is a detailed schematic of the chip's internal structure and external signal pads. Figure 3.6 is an annotated die photo showing the location of the major components, while Figure 3.7 shows how virtual processors are mapped to lanes.

These die photos clearly illustrates the advantages of a vector processor over other forms of highly parallel processor. First, only a small amount of control logic is required to control a large array of parallel functional units. Second, vector instructions allow the vector unit datapath logic to be constructed as a collection of distributed parallel lanes. The vector instruction set guarantees that no communication is required between lanes for arithmetic instructions, reducing wiring overhead and inter-instruction latencies. The only communication between lanes is for vector memory instructions and for vector extract instructions, both of which are handled by the vector memory crossbar located in VMP.

3.4.1 Process Technology

As we begun design, we had access through MOSIS to a 1.0 μm two-layer metal CMOS process fabricated by HP (CMOS26B). Our layout followed MOSIS SCMOS design rules. Initially, the reticle limited die size to 14 mm. As the design progressed, it became clear that our original eight lane design would not fit. Fortunately, a larger 17 mm reticle became available, so we could retain the full eight lanes. The process also improved to 0.8 μm process with 3 metal layers (CMOS26G) late in the design phase. The cells we'd already designed in SCMOS rules could not be shrunk automatically to take advantage of the reduced feature size,

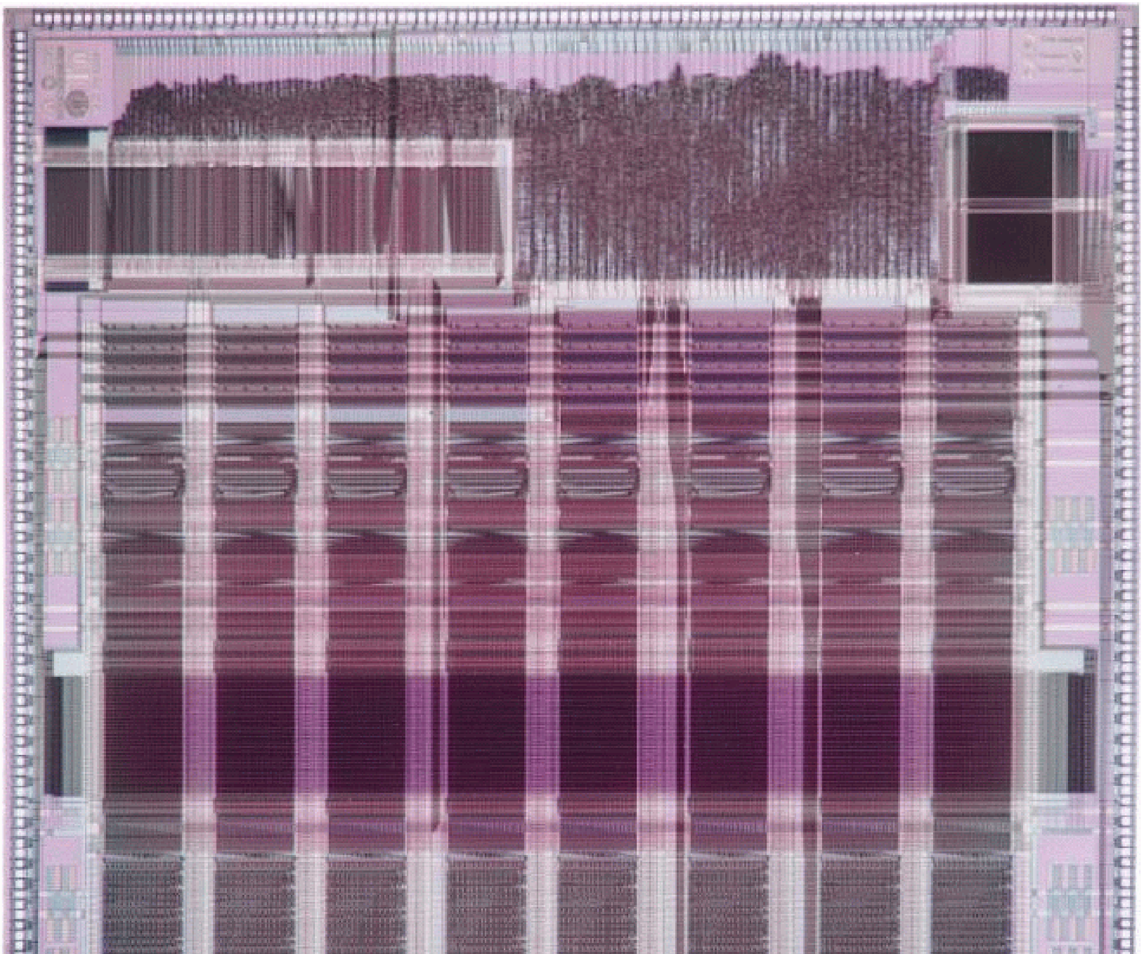

Figure 3.4: T0 die photo.

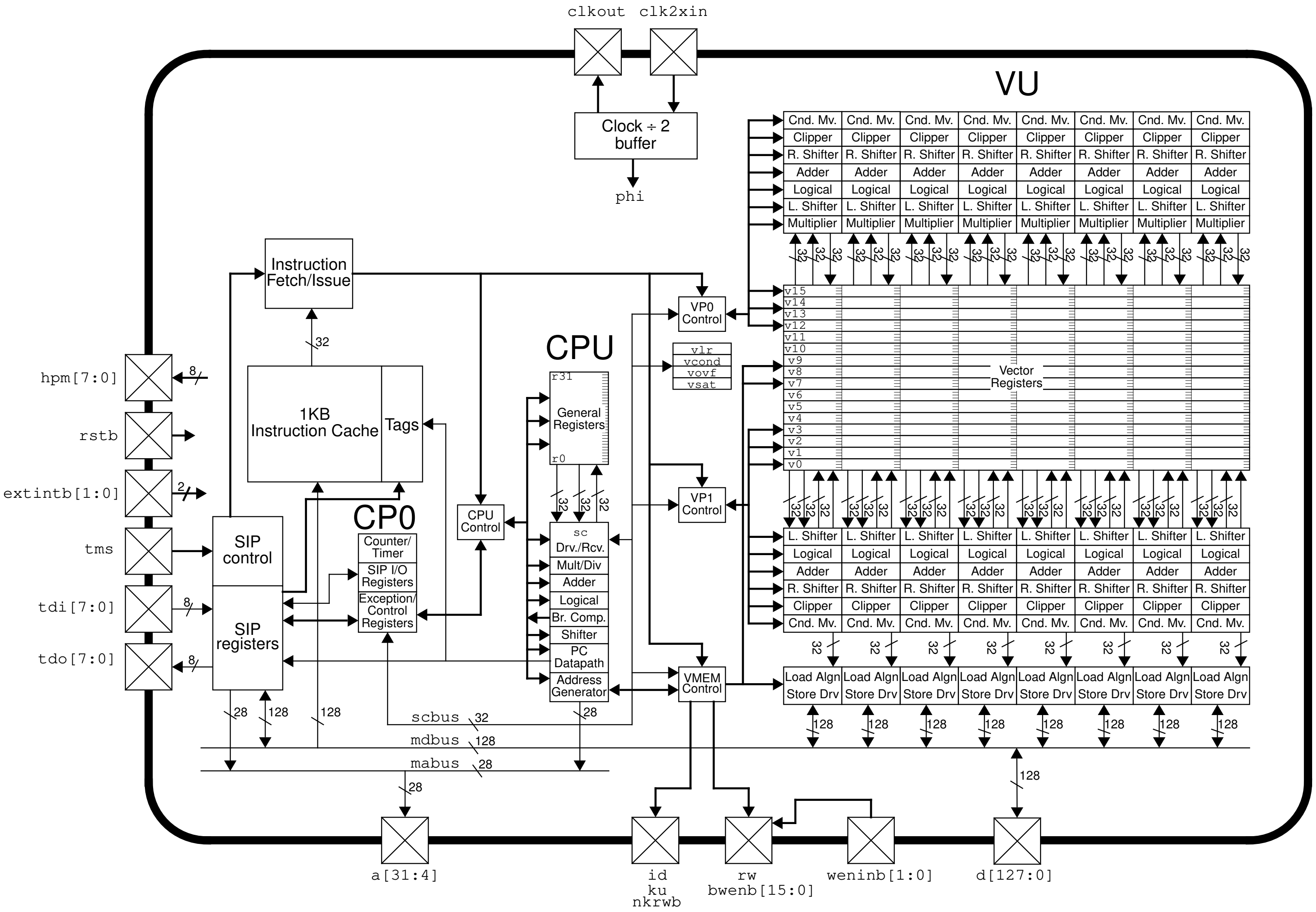

Figure 3.5: Detailed schematic of the T0 implementation.

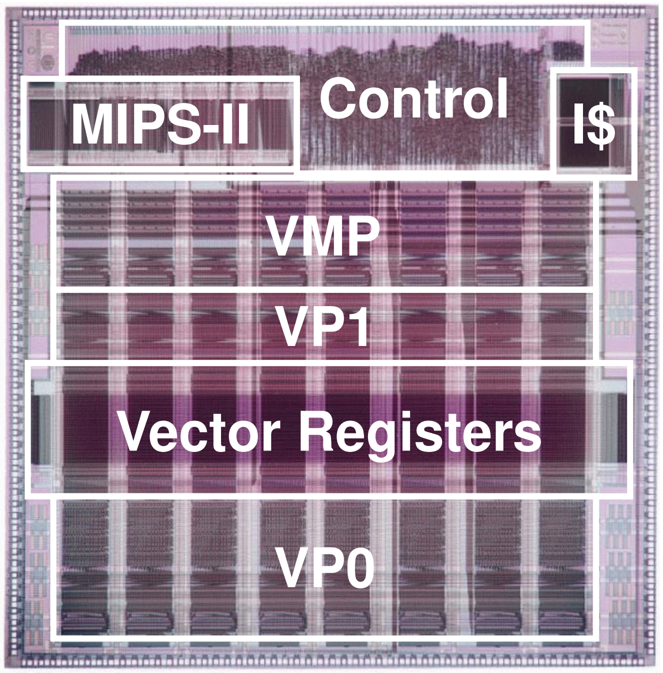

Figure 3.6: Annotated die photo of T0.

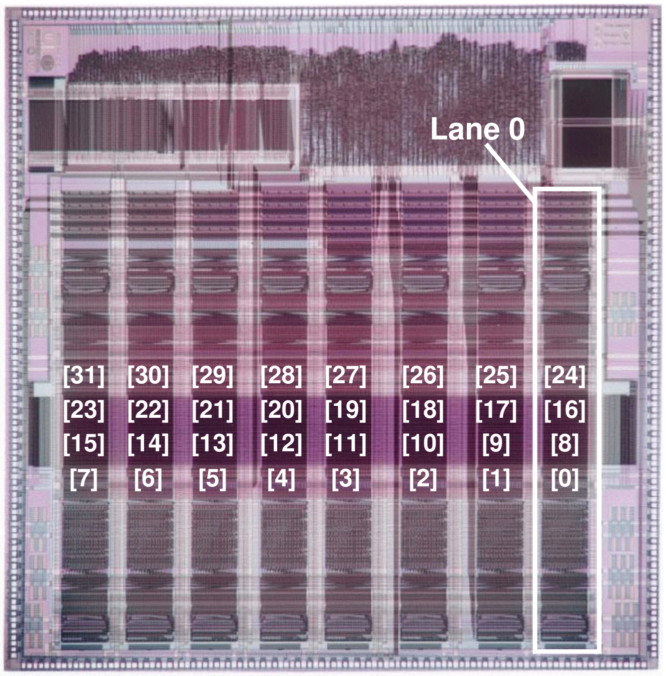

Figure 3.7: Mapping of virtual processors to lanes in the T0 design. The numbers in square brackets show how the 32 virtual processors are striped across the eight lanes.

Component	Area (Mλ^2)	Area (mm^2)	Area (%)
Control logic	135.6	33.9	12.1%
MIPS-II datapath	58.7	14.7	5.2%
Instruction cache	17.1	4.3	1.5%
Vector registers	167.6	41.9	14.9%
VP0	224.9	56.2	20.0%
VP1	106.2	26.5	9.5%
VMP	136.5	34.1	12.2%
(Crossbar in VMP)	(70.6)	(17.6)	(6.29%)
TSIP DMA	12.6	3.2	1.1%
Clock driver	14.9	3.7	1.3%
Large bypass capacitors	22.5	5.6	2.0%
Logos	6.5	1.6	0.6%
Core to pad routing and empty space	133.6	33.4	11.9%
Pad ring	84.5	21.1	7.5%
Total	1121.2	280.5	100.0%
Vector Unit Datapath Total	635.1	158.8	56.6%

Table 3.1: Breakdown of T0 die area. The bypass capacitor figure only includes the separate large bypass capacitors, many other small bypass capacitors are placed underneath power rails around the chip.

and we also had not used a third metal layer. Our only benefit from the new process was somewhat faster transistors, and perhaps also better yield from our now effectively coarser design rules.

3.4.2 Die Statistics

The die occupies $16.75 \times 16.75\,\text{mm}^2$, and contains 730,701 transistors. A breakdown of the die area is given in Table 3.1. The T0 die has a peak clock rate of around 45 MHz, but is run at 40 MHz in the Spert-II system to allow the use of slower and cheaper SRAMs and to provide extra timing margin. The chip dissipates around 12 W worst case from a 5 V supply when running at 40 MHz; typical power dissipation is around 4 W.

3.4.3 Spert-II System

Spert-II [WAK$^+$96] is a double-slot SBus card that packages one T0 together with 8 MB of external SRAM to form an attached processor for Sun-compatible workstations. The integrated design results in a simple circuit board layout that is inexpensive to replicate. A block diagram of the system is shown in Figure 3.8.

T0 is mounted directly to the board using Chip-On-Board packaging. Most of the glue logic is contained in a single Xilinx FPGA that interfaces TSIP to the host SBus, providing a peak DMA I/O transfer rate of around 10 MB/s. An on-board thermometer monitors system temperature and resets the processor if the temperature rises too high. This prevents damage if the board is installed in a workstation with inadequate cooling.

The software environment for the Spert-II system is based on the GNU tools. The `gcc` scalar cross-compiler, `gas` cross-assembler, and `gdb` symbolic debugger were all ported to Spert-II. The assembler was extended to support the new vector instructions.

3.5 T0 Design Methodology

T0 was a completely new custom VLSI design, using full-custom datapath circuitry and standard cell place and route for control logic. The T0 design methodology built on our experiences with previous processor implementations.

3.5.1 RTL Design

The RTL design of the processor was written in C++. The RTL model was the "golden reference" for the whole T0 chip design. The chip was modeled with sufficient accuracy to allow the value in every latch on the chip to be calculated on either phase of any clock cycle. There were two main reasons for writing the model in C++ rather than a hardware description language. First, C++ is a powerful programming language which helped in building our verification environment. Second, high execution rates were achieved, around 1,500 cycles per second on a Sparcstation-20/61, which enabled us to run a large number of cycles for RTL verification.

A T0 ISA simulator was also written which interprets instructions and simulated their effects on architectural state. The ISA simulator could simulate program execution at around 500,000 instructions per second on a Sparcstation-20/61, depending on the mix of scalar and vector code. The ISA simulator was used to verify the RTL design and was also used to enable software development to proceed concurrently with hardware design.

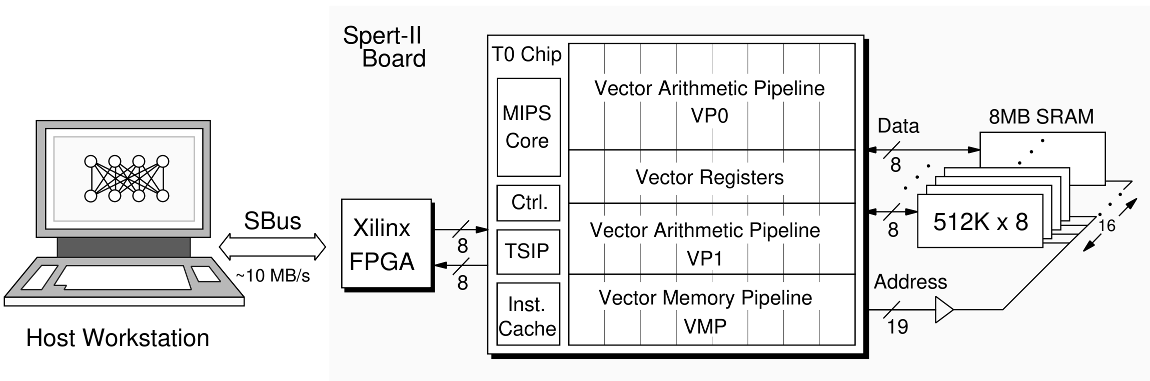

Figure 3.8: Spert-II Block Diagram.

3.5.2 RTL Verification

The verification environment was built on the concept of writing assembler test programs for virtual test machines (VTMs). Each VTM hides differences across different implementations of a VTM by defining: a common set of registers and instructions that are available, which exceptions can be generated, which portions of memory can be accessed, the way the test program starts and stops execution, the way that test data is read, the way that test results are written, and the way that any errors are signaled. The implementation of each VTM on each platform randomizes uninitialized state to improve the chances of exposing errors. Several different VTMs were specified:

- `mips`. This was the first VTM, and only supported the subset of user-mode MIPS-II instructions implemented by the T0 hardware. Programs written for the `mips` VTM could be assembled and run on an SGI workstation as well as the ISA and RTL models. By running the same code on all three platforms and comparing outputs, we could verify that the T0 scalar ISA and RTL was compatible with the real MIPS hardware.

- `t0u`. This VTM added the user-mode vector instructions to the `mips` VTM. The largest set of test programs were written for the `t0u` VTM. The RTL was compared against the ISA model for the `t0u` tests.

- `t0raw`. This VTM exposes most of the kernel state and was used to verify T0 exception handling. Test programs are responsible for initializing the processor after reset and for handling any exceptions. A few parts of the machine state cannot be modeled accurately by the ISA simulator and so were omitted from this model.

- `t0die`. T0 was designed to run test programs from the instruction cache, reporting back results over the TSIP connection. In this way, we could test the die when external RAM was not present, both at wafer sort and after initial bonding to the Spert-II circuit board. A very restricted VTM was defined that would operate out of I-cache with no external SRAM and return signatures over the TSIP port. Test programs could be verified on the ISA and RTL simulators running in a special zero memory mode.

- `t0cyc`. This VTM is like `t0raw` except now all machine state is visible, including elements such as the counter and exception registers whose values depend on exact machine cycle count. Because the ISA simulator does not model execution to cycle accuracy, it could not run these test programs. The RTL was the only implementation available before chip fabrication, and so these test programs were written to be self-checking.

- `t0diecyc`. This VTM is the cycle accurate equivalent to the `t0die` VTM.

The directed tests were designed to cover every piece of logic in isolation, and in total nearly 100,000 lines of hand-written assembler test code were generated for these various VTMs.

Even this large number of directed tests is not enough to cover all the interactions possible in a highly parallel microprocessor such as T0. Complementing the directed tests was an effort to generate focused random test codes for the `mips` and `t0u` VTMs. The main difficulty with generating random test codes is ensuring that the generated code obeys the various constraints of a VTM. The first random test program was named `rantor`[2], and this incrementally built up test programs by tracking values in registers. This approach was inflexible and error prone because of the difficulty in generating some sequences and in validating that the output was valid VTM code. Even so, several bugs were found early in the design process with this tool.

A second test program generator `torture` was written to overcome the problems with `rantor`. `torture` took an alternative approach, providing a central scheduling module that interleaved the output of user-specified instruction sequence generators. Each instruction sequence generator is a C++ object that produces a random instruction thread with some number of registers marked as either visible or invisible. The central module allocating registers to the various threads, interleaves their execution, and ensures that only visible registers are reported in the final output state. The `torture` approach allowed more control over the types of sequences generated, and also dramatically simplified the task of generating complicated instruction sequences that nevertheless obeyed VTM constraints.

Using the random test generators, many billions of RTL verification cycles were run on the network of workstations at ICSI using a parallel make tool. When random tests uncovered RTL bugs, a directed test was written to reproduce the bug and added to the directed test suite. A total of 26 bugs were found by the random testing.

3.5.3 Circuit and Layout Design

T0 uses a single wire two-phase clocking scheme [AS90]. An input clock at twice the operating frequency is divided down to ensure a 50% duty cycle, buffered up through a tree-structured driver chain, then driven across the chip using a single gridded node. A clock output is provided to synchronize external circuitry to the T0 internal clock.

The Berkeley `magic` layout editor was used for leaf-cell editing and to build the final design. A standard set of datapath cells were defined and assembled into datapaths using a custom-written procedural layout generators that produced `magic` layout. The global routing of the whole chip was also performed using procedural layout generators.

Small cells were simulated at the circuit level using HSPICE. Larger circuit level simulations used the CaZM table-driven simulator. The entire vector register file, containing over 250,000 transistors, was simulated at the circuit level using CaZM. Several cycles of operation were simulated to help verify the self-timed read and write circuitry.

A full transistor level schematic was built using ViewLogic. This schematic was simulated with the `irsim` switch level simulator using test vectors generated automatically from the RTL model as it ran test programs. This switch level simulation ran at nearly 2 cycles per second on a Sparcstation-20/61 and took

[2]Written by Phil Kohn.

just over 100 MB of physical memory. A parallel make facility was used to run these switch level simulations with automatic checkpointing and restart. Some switch-level simulation runs took over a week to complete.

The final layout was extracted using `magic` and checked against the full transistor level schematic using the `gemini` LVS tool. The final DRC check was performed using Dracula.

3.5.4 Control Logic Design

The control logic was implemented as two standard cell blocks. A tiny block provided the TSIP test access circuitry, while the rest of the control logic was placed in a single large L-shaped region on the die.

The control logic was assembled using a mixture of hand-entered and synthesized schematics. The control logic was split into around 20 logic blocks, each manually converted from the C++ RTL into the BDS hardware description language. The BDS for the block was converted into `blif` and fed into the `sis` synthesis package. For each block, the `sis` synthesis script was hand-optimized for either area or power. The control schematic stitched the synthesized control blocks together with hand-entered logic and storage elements.

A gate-level net-list generated from the top-level schematic for the entire control block was fed to the TimberWolf tool for standard cell place and route. Extensive massaging of the TimberWolf parameters was required to fit the control logic into the available space on the die. This phase took two months to complete after the rest of the design had been frozen.

The leaf standard cells were mostly hand-picked from an existing library, but modified to provide better power, ground, and clock distribution. New standard cells were designed to provide fast latches and flip-flops.

3.5.5 Timing Analysis

A crude static timing analysis was performed manually across the entire chip. A set of scripts were written to flag signals that might not meet timing constraints. These scripts combined automatically extracted net RC delays and manually-entered block timing data. The timing analysis revealed several critical paths, mainly located in the synthesized control logic and in driving control signal outputs across the datapaths. These paths were reworked by retiming control logic, improving synthesis scripts, increasing control line driver size and/or using thicker metal control wires.

3.5.6 Timeline and Status

The design of the Torrent ISA began in November 1992. The T0 VLSI design was completed by 2 graduate students and 1 staff VLSI engineer. A full time hardware engineer designed the Spert-II board, while a full time software engineer developed the software environment and application programs for the project.

The design taped out on February 14, 1995, and three wafers of the first silicon were received on April 3. Forty good die were obtained from this first batch and the first program was run on a completed

Spert-II board on June 12. No bugs have been discovered in the first-pass silicon. A second run of eight wafers using the same masks was delivered 18 March, 1996, yielding a further 201 good die.

A total of 34 Spert-II boards are currently in operation at 9 sites in Europe and the USA. The boards are in daily production use providing a substantial speedup over commercial workstations (e.g., six times faster than an IBM RS/6000-590) for our speech recognition work [WAK+96].

3.6 Short Chimes

The most distinguishing feature of the T0 implementation versus earlier vector supercomputer designs is that chimes are much shorter. This is made possible by the single chip implementation which reduces intra-CPU latencies and allows higher performance control logic.

Vector supercomputer designers are faced with the dilemma illustrated in Figure 3.9. A single-chip CPU implementation would reduce intra-CPU latencies and provide better cost/performance, but would limit memory bandwidth to that which can cross the pins of a single die. But vector supercomputer customers judge machines primarily on sustainable single CPU memory bandwidth and competitive pressure forces designers to select multi-chip implementations, which increase intra-CPU latencies and give worse cost/performance [Sch97b].

Multi-chip CPU implementations are responsible for the startup penalties in vector supercomputers. As illustrated in Figure 3.10, vector startup latencies have two separate components: functional unit latency and *dead time*. Functional unit latency is the time taken for one operation to propagate down the functional unit pipeline. Dead time is the time required to drain the pipelines of a vector functional unit after it completes a vector instruction before a subsequent vector instruction can begin execution in the same unit.

Functional unit latency is relatively benign, because independent vector instructions can be scheduled to cover this latency and keep the machine's pipelines saturated. This process is similar to instruction scheduling to cover pipeline latencies for a pipelined scalar processor. Dead time has more serious performance implications because instruction scheduling cannot recover the lost pipeline cycles.

Dead time represents a fixed overhead on every instruction startup. Vector supercomputer designers use longer chimes in attempts to amortize this dead time overhead over longer periods of useful work. For example, the Cray C90 [Cra93] has four cycles of dead time inbetween issuing two vector instructions to the same vector functional unit. The Cray C90 vector registers hold up to 128 elements and the vector unit has two lanes, so chimes are 64 clock cycles long. The four cycle dead time overhead limits attainable functional unit efficiency to 94% even with the longest 128-element vectors. Efficiency on short vectors is correspondingly worse. Vectors of length eight can never achieve more than half the peak performance of two elements per cycle from a vector functional unit.

In contrast, the tightly integrated control logic on T0 eliminates dead time. Operations from two different vector instructions can follow each other down a functional unit pipeline with no intervening pipeline bubbles. This allows T0 to saturate a vector functional unit producing eight results per cycle by issuing one

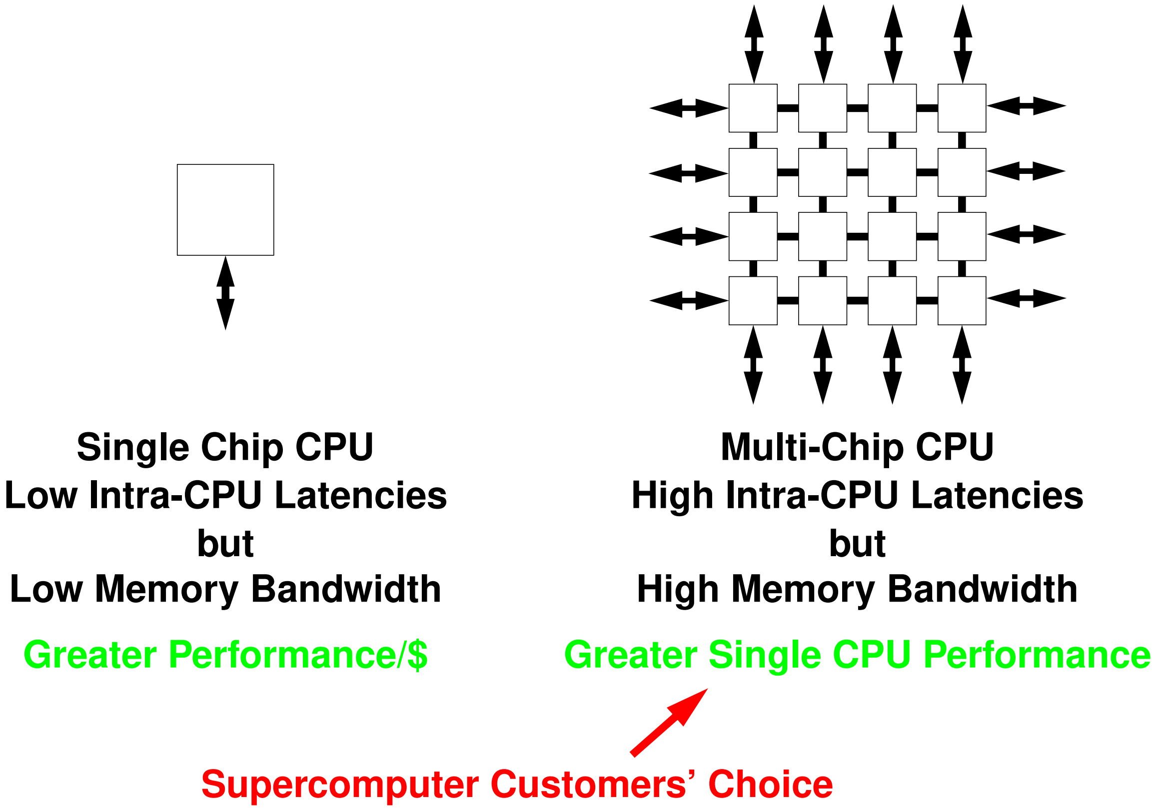

Figure 3.9: The vector supercomputer designer's dilemma. Single chip processors offer lower intra-CPU latencies and better cost/performance, but multi-chip CPUs enable greater absolute memory bandwidth which is one of the main criteria in supercomputer purchase decisions.

eight element vector instruction per cycle.

T0 has three vector functional units (VFUs) each capable of generating eight results per cycle. With the maximum vector length of 32, a vector instruction takes only four clock cycles to execute. This vector length was chosen to be just long enough to allow a single instruction issue per cycle to saturate the three VFUs.

Figure 3.11 illustrates how a segment of code is executed on T0 with the vector length set to the maximum 32 elements. Every four clock cycles, one instruction can be issued to each VFU, leaving a single issue slot available for the scalar unit. In this manner, T0 can sustain 24 operations per cycle while issuing only a single 32-bit instruction every cycle. The diagram also shows how T0 overlaps execution of multiple vector instructions to take advantage of vector instruction-level parallelism, as well as intra-vector data parallelism.

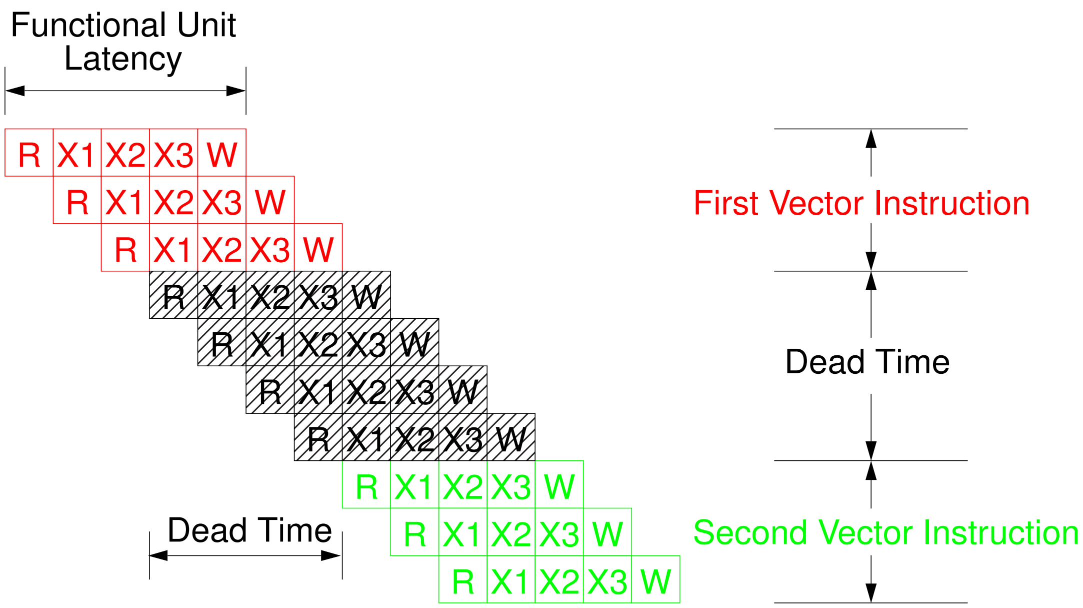

Figure 3.10: Pipeline diagram of one vector functional unit showing the two components of vector startup latency. Functional unit latency is the time taken for the first operation to propagate down the pipeline, while dead time is the time taken to drain the pipelines before starting another vector instruction in the same vector functional unit.

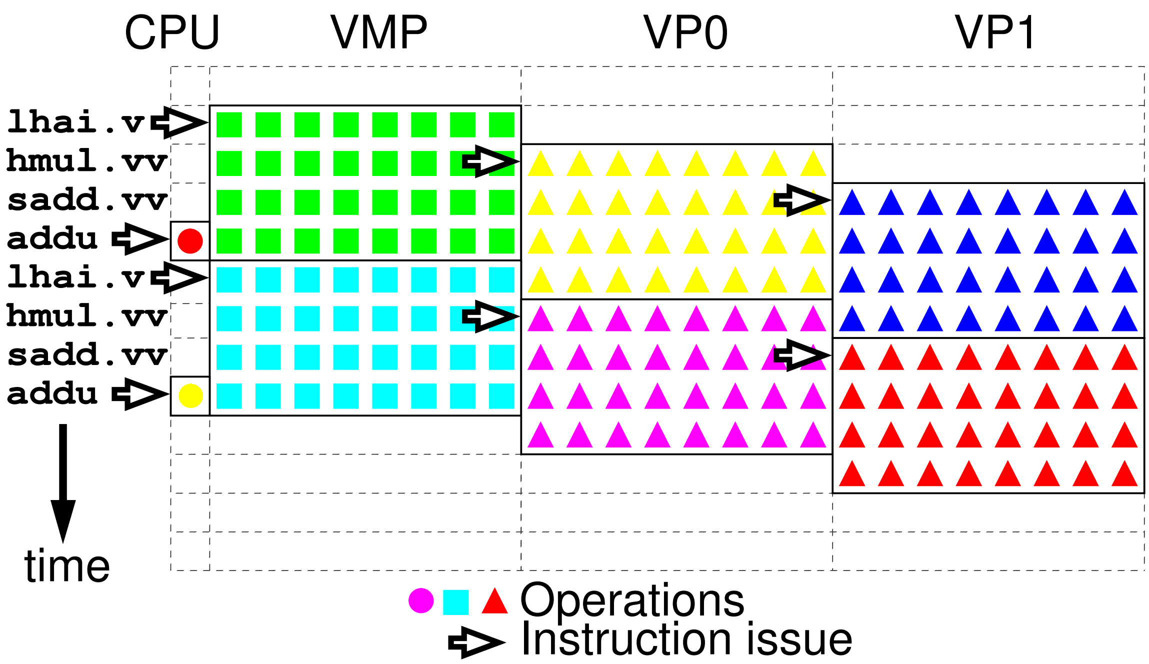

Figure 3.11: Execution of a code fragment on T0 with vector length set to 32 elements. A vector load is issued to VMP on the first cycle, and begins execution of eight operations per cycle for four cycles. A vector multiply is issued on the subsequent cycle to VP0, and the execution of this vector multiply overlaps with the vector load. Similarly, a vector add is issued on the third cycle to VP1, and execution of the add overlaps the load and multiply. On the fourth cycle a scalar instruction is issued. On the fifth cycle the vector memory unit is ready to accept a new vector load instruction and the pattern repeats. In this manner, T0 sustains over 24 operations per cycle while issuing only a single 32-bit instruction per cycle.

Chapter 4

Vector Instruction Execution

Although T0 provides the first example of a complete vector microprocessor, it does not support long latency main memories and lacks virtual memory and floating-point arithmetic. This chapter presents vector instruction execution pipelines suitable for future vector microprocessors. These pipeline organizations can tolerate long memory latencies even with short vector chimes and can provide the exception handling required for virtual memory and floating-point arithmetic.

Section 4.1 first reviews the overall structure of a typical vector microprocessor. Section 4.2 then describes the changes required to a scalar unit when coupled with a vector coprocessor.

Section 4.3 considers various ways to tolerate memory latency and describes how dynamic scheduling is particularly attractive for a vector microprocessor. Section 4.4 reviews decoupled execution [Smi89], a limited form of dynamic scheduling that is particularly appealing for vector machines [EV96]. Section 4.5 examines full out-of-order execution for vector machines [EVS97], describing the problems inherent in vector register renaming and presenting solutions at the instruction set level. One of the main advantages of vector register renaming is that it provides a scheme to provide precise exceptions in a vector machine. Section 4.6 introduces alternative techniques which define architectural exceptions so that vector register renaming is not necessary to provide high vector throughput while supporting virtual memory and IEEE floating-point arithmetic.

Section 4.7 presents the detailed design of a decoupled vector pipeline which supports virtual memory, showing how it can tolerate large memory latency even with short chimes.

Section 4.8 enumerates the few classes of instruction which expose memory latency to software in a dynamically scheduled vector machine, and proposes techniques to reduce the effects of memory latency in those cases.

The chapter concludes in Section 4.9 with a description of the logic required to control interlocking and chaining for the decoupled pipeline. This describes how the control logic is very similar to that for T0, but with the addition of instruction queues and a set of interlocks related to address generation.

4.1 Overall Structure of a Vector Microprocessor

The general structure of a vector processor is shown in Figure 4.1. The major components of a vector processor are the scalar unit, scalar caches, the vector unit, and the memory system.

The scalar unit for a vector machine can be single-issue, superscalar, or VLIW. For the purposes of this thesis, I assume the scalar unit is based on a conventional RISC load/store architecture and code samples are based on the MIPS scalar RISC instruction set. The scalar unit fetches and dispatches instructions to the vector unit. The scalar unit has an instruction cache and a scalar data cache. Hardware keeps the scalar data cache coherent with vector unit memory accesses.

The vector unit contains some number of vector data registers and vector flag registers coupled to some number of vector functional units (VFUs). When a vector instruction is dispatched, it is sent to a particular VFU for execution. If there is more than a single VFU, the microarchitecture can exploit inter-instruction parallelism by overlapping execution of two or more separate vector instructions in separate VFUs. VFUs are divided into three broad categories: vector arithmetic units (VAU), vector memory units (VMU),

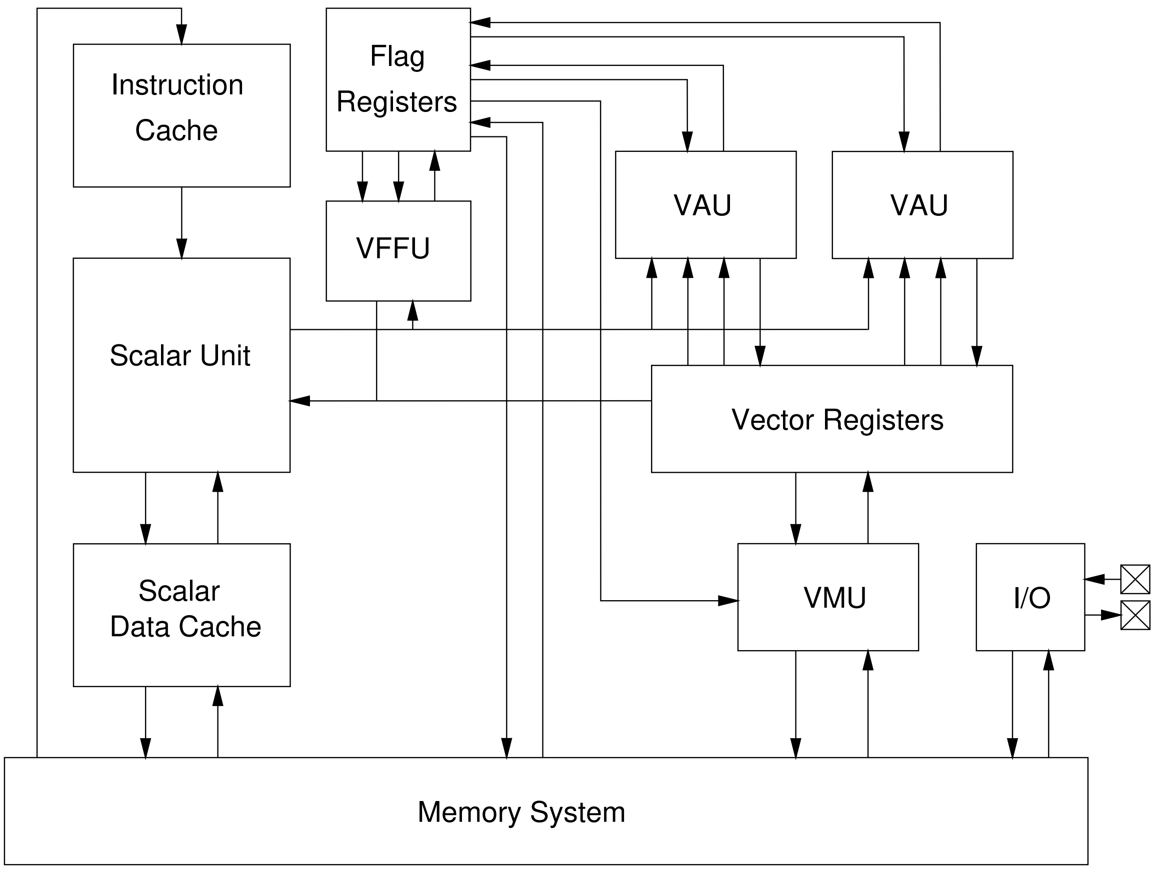

Figure 4.1: Generic vector processor architecture. This machine has two vector arithmetic units (VAUs), one vector memory unit (VMU), and one vector flag functional unit (VFFU).

and vector flag functional units (VFFUs). The VFUs communicate with each other via the vector data registers and the vector flag registers.

4.2 Scalar Instruction Execution in a Vector Processor

The scalar unit in a vector processor is responsible for running non-vectorizable code and for supporting the execution of vector code. Only minor changes to a conventional scalar processor are required to support a vector coprocessor. These modifications require little additional logic and should not affect cycle time.

The instruction cache and instruction fetch stages are largely unaffected by the addition of a vector unit, provided the instruction format of the vector extensions is compatible with that of the base scalar ISA. If there is limited opcode space available for adding the vector extensions, the instruction encoding could become complex and require several levels of logic to decode. It might then be beneficial to predecode instructions during instruction cache refills, storing a decoded version in the cache to speed instruction dispatch. Such predecoding is common in modern superscalar CPU designs [Yea96, Gwe96e, Gwe96a].

The instruction decoder must be modified to parse vector instructions, retrieve any scalar operands they require, and dispatch the decoded instruction and scalar operands to the vector unit. There should be no increase in scalar register ports or interlock and bypass logic provided vector instructions require no more scalar operands than a scalar instruction. Instructions that read state back from the vector unit require synchronization with the vector unit pipeline and a data path back into the scalar unit registers. This path may already exist to support other coprocessors. The scalar unit manages control flow and its pipeline is used to handle exceptions, so there must be a mechanism for synchronizing exception-generating vector instructions with the scalar pipeline. Such mechanisms will usually exist for serializing accesses to non-bypassed system registers.

An in-order non-speculative vector unit can be attached to an aggressive speculative and out-of-order scalar unit. The scalar unit can fetch and dispatch vector instructions to reservation stations to await the arrival of any scalar operands. When all scalar operands arrive and the vector instruction reaches the head of the reorder buffer, it can be issued to the vector unit non-speculatively. The vector instruction can be committed after all possible vector exceptions have been checked. The machine organizations described below allow vector exceptions to be checked in a few cycles regardless of memory latency. This allows vector instructions to be removed from the scalar reorder buffer many cycles before they complete execution, limiting the impact of long vector unit latencies on scalar reorder buffer size.

We should expect the workload for a scalar unit within a vector machine to be somewhat different from that of a vectorless machine. Based on data from Wall's study [Wal91], Lee [Lee92, Chapter 3] found that many applications with high ILP are vectorizable, and that many non-vectorizable applications have low ILP. Smith et. al. [SHH90] suggest that the non-vectorizable portion of the workload might best be handled by a scalar processor that emphasizes low latency rather than parallel instruction issue. Based on his analysis of Cray Y-MP performance, Vajapeyam [Vaj91, Chapter 5] also argues for a less highly pipelined and lower latency scalar unit to handle the non-vectorizable portion of the workload. Free of the burden of providing high throughput on vectorizable code, such a scalar processor should also be simpler to design. Because cycle time might be reduced and because resources are freed for larger caches and lower latency functional units, it is possible that such a scalar processor would have a performance advantage over a wide superscalar processor for tasks with low ILP.

Scalar processors typically employ a hierarchy of data caches to lower average memory latency. Vector memory accesses will usually bypass the upper levels of the cache hierarchy to achieve higher performance on the larger working sets typical of vector code. Extra logic is required to maintaining coherence and consistency between vector and scalar memory accesses, but this will also usually be required for multiprocessor support or for coherent I/O devices.

4.3 Vector Instruction Execution

This section considers instruction execution pipelines for a vector microprocessor.

Unlike high-performance scalar processor designs, where considerable effort is expended on parallel scalar instruction issue, vector instruction issue bandwidth is not a primary concern. A single vector instruction issue per cycle is sufficient to saturate many parallel and pipelined functional units, as shown by the T0 design in Section 3.6.

Memory bandwidth is often the primary constraint on vector performance. To sustain high bandwidth, it is necessary to generate a continuous stream of memory requests. Vector microprocessors may often be coupled with commodity DRAMs which have higher latencies than the SRAMs used in vector supercomputers; memory latencies of 100 cycles or more may be typical for fast microprocessors connected to off-chip DRAM. Sustaining high bandwidth with these high latencies requires the generation of many *independent* memory requests.

There are four main techniques used to tolerate main memory latencies in machines today:

- **Static scheduling** requires the least hardware support. Software schedules independent operations between a load and operations that use the load data. The machine can then overlap processing of a load request to memory with execution of independent operations later in the same instruction stream. Vector instructions are a form of static scheduling, where multiple independent operations are scheduled into one instruction.

- **Prefetching** can be accomplished in software or hardware. In either case, data is requested from memory before the load instruction is issued, which reduces the latency visible to the load. Prefetch schemes require a buffer in which to place prefetched data before it is requested by a load instruction. Software data prefetch [SGH97] requires the least hardware; prefetch instructions are scheduled by software and prefetch data into the caches. Hardware data prefetch techniques [CB94a] attempt to predict future references based on the pattern of previous references.

- **Dynamic scheduling** covers various hardware techniques which execute instructions out of program order. Rather than allowing an instruction to block the issue stage when its operands are unavailable, it is placed in an instruction buffer to await their arrival. The issue stage is then free to find independent instructions later in the same instruction stream.
- **Multithreading** [BH95] finds independent instructions by interleaving the execution of multiple parallel instruction streams.

Traditional vector supercomputers rely on a combination of static instruction scheduling and long-running vector memory instructions to tolerate memory latency. This worked well for early vector machines, where memory latencies were around 10–20 clock cycles and a single vector instruction might execute for 64 clock cycles. For vector microprocessors, latencies could rise to over 100 clock cycles, and chimes might shrink to last only 4–8 clock cycles. More aggressive static vector instruction scheduling can help compensate for short chimes, by loop unrolling and software pipelining stripmine loops for example. Because scheduled instructions tie up architectural registers while waiting for results, the number of available architectural registers places a limit on the amount of static instruction scheduling possible. Software prefetch is really another form of static instruction scheduling whose main advantage is that data is brought into buffers rather than into architectural registers. Because regular load instructions are still required to move data from the prefetch buffers into the architectural registers, software prefetch has the disadvantage of increasing instruction bandwidth.

A limitation common to all forms of static instruction scheduling is that unpredictable program control flow and separate compilation of code reduce the number of instructions the compiler can consider for rescheduling and hence the amount of latency that can be tolerated. This problem is exacerbated by vector execution which can dramatically reduce the number of cycles taken to execute a section of code compared with scalar execution.

Hardware prefetch schemes have the advantage of exploiting the dynamic information in run-time program reference streams and also do not tie up architectural registers or require extra instruction bandwidth for prefetch instructions. The main disadvantages of hardware prefetch schemes are that they are speculative and so can generate excess memory traffic. This is less of a problem for scalar machines which cannot otherwise saturate their memory systems, but is a severe drawback for vector machines which are often constrained by hardware bandwidths. Another problem with hardware prefetch is that it can only react to observed reference patterns and so can have high startup latencies, especially if a prefetch filter [PK94] is employed to reduce useless speculative prefetches. Hardware prefetching is also only effective on predictable strided reference streams.

The great advantage of dynamic instruction scheduling is that it can overlap execution of instructions across unpredictable branches and separately compiled code boundaries. The main disadvantage of dynamic instruction scheduling is the expense of the control logic and the extensive design effort required. A source of much complexity is the desire to maintain precise exception handling despite out-of-order execution.

Multithreaded machines interleave execution of instructions from separate instruction streams to hide latencies. The main disadvantage of multithreaded machines is the cost of providing separate state for each interleaved thread of execution.

All of these techniques can be combined to help tolerate memory latency in vector microprocessors. Hardware and software vector prefetching schemes are not considered further here, because vector memory instructions subsume most of their functionality.

Espasa and Valero [EV97b] explore dynamic scheduling and multithreading for vector machines and documents the benefits of these techniques for binaries compiled for a Convex C3 vector architecture. Dynamic scheduling and multithreading are much simpler to implement for vector instructions than for scalar instructions because the large granularity of vector instructions means they require less instruction bandwidth and much smaller instruction buffers.

For the remainder of this chapter, I concentrate on the design of dynamically scheduled pipelines for vector machines. Espasa's thesis [Esp97b] describes two forms of dynamically scheduling: decoupled architectures [Smi89], which provide a limited form of dynamic scheduling by separating address generation from data computation, and full out-of-order execution with vector register renaming.

4.4 Decoupled Vector Execution

A decoupled vector machine [EV96] splits the vector unit into address generation and data computation components and connects the two with data queues.

All vector arithmetic instructions are sent in-order to the computation instruction queue. Once at the head of the queue and once all operands are ready, they are dispatched to an arithmetic functional unit and begin execution on data in vector registers as in a conventional vector machine.

Vector memory instructions are split into two components: one which generates addresses and moves data between memory and internal vector data queues, and one which moves data between the vector data queues and the vector register file. The first component is dispatched in-order to the address generation instruction queue. Once at the head of the queue and assuming any address operands are available, the address generation component begins generating a stream of memory requests to fill or empty a vector data buffer. In parallel, the second component is sent to to the computation instruction queue, where it is interleaved with vector arithmetic instructions in program order. When this second component is at the head of the queue and its operand is ready, it is dispatched and begins execution, moving data between a data queue and the vector register file. From the perspective of the vector register file, all operations happen in program order.

The advantage of a decoupled pipeline compared to the standard in-order issue pipeline common in vector supercomputers, is that vector arithmetic instructions waiting on vector memory operands do not block the issue stage. The vector arithmetic instruction is sent to an instruction queue, freeing the issue stage to run ahead to find more vector memory instructions later in the instruction stream; these can begin address generation before earlier vector arithmetic instructions have begun execution. The design and operation of a decoupled pipeline is described in greater detail in Section 4.7 below.

4.5 Out-of-Order Vector Execution

Espasa et. al. [EVS97] found significant benefits to full out-of-order vector execution with vector register renaming for binaries compiled for a Convex C3 architecture. The C3 has only eight architectural vector registers and no load chaining. For machines with more vector registers, load chaining, and a compiler capable of aggressive static scheduling, it is unclear how large the benefits would be over simpler decoupled schemes.

Although not addressed by Espasa et. al. [EVS97], conventional vector ISAs such as the Convex, severely complicate the implementation of vector register renaming because of partial vector register updates. Similar problems occur in scalar machines that allow partial scalar register updates. These partial updates can occur in three ways in a vector machine:

- **Reduced vector lengths.** Conventional vector instruction sets are defined such that when vector length is set below maximum, the values in vector register elements above the current vector length setting are left undisturbed. This behavior is desirable as it allows a straightforward implementation of reduction operations (Section 10.1) for vectors that are not multiples of the maximum vector register length. With vector register renaming, the values in the upper elements of the physical destination register will initially be values left over from the previous use of that register. These must be updated to the tail values from the architectural vector register state to preserve the original vector ISA semantics. Unfortunately, this would appear to require that all vector instructions effectively run for the maximum vector length regardless of current vector length setting. In defining a new vector ISA, it would be prudent to specify that destination elements past the current vector length become undefined. Reductions can be implemented by using a mask to control effective vector length (see Section 6.8), with the aid of an instruction that sets the first N bits in a mask register.

- **Masked vector instructions.** Each element of a new physical destination vector register for a masked vector instruction must be written, either with the result of the operation where unmasked or with the original architectural register value where masked. This can imply a third vector register read port for each VFU to read the old value of each destination element. Vector merge instructions, as defined in the Cray instruction set [Cra93], avoid the extra read port because they always write one of the two sources to the destination.

- **Scalar insert into vector register.** One approach to handle inserting a scalar element into a vector register would be to copy over all valid elements into the new physical register, merging the new element into the appropriate position, but this can tie up a VFU for multiple just to insert a single element. An alternative scheme is to reuse the same physical register by updating in place rather than copying data to a newly allocated physical register, but this requires waiting until all previous instructions have committed. Scalar inserts into a vector register are usually performed as part of a partially vectorizable loop where a whole vector of scalar values will be produced. Software can avoid scalar inserts by writing values to a temporary memory buffer instead. The buffer can then be read into the vector unit as one complete unit.

Although these ISA-level solutions reduce the cost of vector register renaming, they also imply a performance penalty for implementations without vector register renaming.

Apart from vector data registers, vector flag registers must also be renamed. As described in Chapter 6, some vector instructions might update multiple flag registers. Because flag registers are relatively small, it is perhaps simplest to rename the entire flag register set. All flag operations then read all flag registers and write all flag registers to a new renamed flag set. The flag register design described in Section 6.11 already provides read and writes of the entire flag set. Flag storage must be increased by an integer factor to hold multiple renamed versions of the flag set.

4.6 Exception Handling for Vector Machines

Handling exceptions can be problematic in vector machines. One solution, adopted by the IBM vector facility [Buc86], is to execute all vector instructions sequentially. This reduces performance compared with schemes that allow multiple vector instructions to overlap in execution. Another approach that allows vector instruction overlap is to provide vector register renaming with a way of rolling back state changes in the event of an exception on an earlier instruction. As described above in Section 4.5, vector register renaming incurs overheads in instruction set design and implementation complexity.

This section explores an alternative approach which relaxes the requirements for precise exception handling to allow simpler implementations to attain high performance. In particular, these techniques are suited to decoupled vector pipeline designs with no vector register renaming.

Note that the only exceptions which require special treatment are synchronous data-dependent exceptions that occur after instruction issue [HP96, Chapter 3]. Asynchronous exceptions, such as device interrupts, can be handled by converting any instruction before the issue stage in the pipeline into a trap pseudo-instruction. Similarly, all synchronous exceptions detected during fetch and decode, such as instruction page faults or illegal instructions, can be handled by converting the faulting instruction into a trap pseudo-instruction. The troublesome exceptions can be divided into two categories: arithmetic traps to support IEEE floating-point arithmetic [IEE85] and data page faults.

4.6.1 Handling Vector Arithmetic Traps

Without some form of revocable register renaming, a vector machine will not be able to provide precise vector arithmetic traps without a large performance loss. Some scalar systems make precise traps a user-selectable mode [WD93, Gwe93]. In this mode, further instruction issue must be delayed until all ongoing operations are known not to require a trap. This scheme can also be used in a vector machine. Whereas a scalar machine must delay after every operation, a vector machine is only delayed once per vector, but this delay can be longer given a large instruction queue between instruction issue and execution.

One improvement is to specify that vector arithmetic instructions cannot themselves cause traps but can only write sticky exception flags. Vector floating-point instructions are defined to write IEEE-mandated default values [IEE85] into the destination data register at element locations where exceptions occur. A separate trap barrier instruction can be inserted by software in the cases where a user-level resumable vector arithmetic trap handler is required. The trap barrier checks the exception flag register for any set exception flags of the appropriate type. This technique allows a simple implementation to continue issuing further instructions after a vector arithmetic instruction without worrying about traps, and only requires that the machine stall issue at the explicit trap barrier instructions. Software must ensure that sufficient machine state is preserved so that a handler can process any traps that occur. This scheme is similar to that employed in the Alpha architecture [Sit92], except that it avoids the problem of *requiring* trap barriers just to implement IEEE compliant default results.

Trap barriers are only used when normal execution has to resume after taking an arithmetic trap. An implementation can also provide support for trap debugging, which allows the location where an exception was raised to be determined accurately without requiring additional trap barriers be inserted in the code. Several levels of exception checking can be provided which trade performance for trap accuracy. In decreasing level of accuracy these could include:

- A precise trap mode which can trap at every vector arithmetic instruction as well as every trap barrier. This would provide the most accurate information at the lowest performance.

- A basic-block level trap mode which can only trap at jumps and branches or at an explicit trap barrier. This allows exceptions to be traced to the basic block responsible, while only checking exception flags once per basic block.

- A subroutine level trap mode which can only trap at a subroutine return or an explicit trap barrier. This allows exceptions to be traced to the subroutine responsible, while only checking exception flags once per subroutine.

- The default execution mode which only traps at an explicit trap barrier.

Section 7.2 on page 130 contains further details on arithmetic trap handling in the context of IEEE floating-point arithmetic. In particular, it discusses how exception flags and conditional operations can be used to support some applications that would otherwise require precise trap handling.

4.6.2 Handling Vector Data Page Faults

Handling data page faults is one of the more difficult design problems in a vector machine. Most modern operating systems support multiprogramming and demand paging. Once one task experiences a page fault, its state is saved, and a second task can then use the CPU while the first task awaits the arrival of the missing page from disk. When the page arrives, the first task can then resume execution after restoring state.

Most vector machines have avoided the problem by requiring that all pages accessed by the vector unit are pinned into physical memory during execution [Rus78, UIT94]. These restrictions have been acceptable in the supercomputing market for two reasons. First, vector supercomputers primarily run large batch jobs which can be swapped rather than demand-paged. Second, their expense warrants tuning application code to fit into real memory with explicit handling of data movement to and from fast solid-state secondary storage. It is likely, however, that demand-paged virtual memory will be a requirement for more general-purpose, low-cost, vector microprocessor systems running popular operating systems. Demand-paging can provide better response in a time-shared interactive environment, and simplifies coding when an application has a data set larger than available real memory.

There have been a few commercial vector machines that provide a demand-paged virtual memory [HT72, DEC90a, Buc86]. Providing precise traps on all vector memory accesses would cause a large performance loss for vector machines without some form of revocable register renaming. Fortunately, precise data page fault traps are not required for demand paging. It is sufficient that enough machine state be saved to allow execution to resume after the operating system has retrieved the missing page from secondary storage. For example, the vector unit in the DEC Vector VAX extensions can save and restore its internal state to memory [DEC90a]. The operating system code only needs to know the maximum size of the vector machine state, not the interpretation of the bits. The following discussion of pipeline structure assumes a data page fault model that saves vector unit internal state, with instruction continuation after the faulting page is reloaded.

4.7 Example Decoupled Vector Pipeline Design

This section develops an in-order, decoupled pipeline structure which can sustain peak pipeline throughput while supporting demand paging in a high latency main memory. This design is used to show the control complexity required and is later used to explain where inter-instruction latencies are exposed to software.

Figure 4.2 shows details of the design. The configuration shown has a single load or store VMU with two VAUs. The VMU is shown with memory data buffers to decouple vector memory instructions from arithmetic instructions [Esp97b]. Vector instructions pass through three queues before being dispatched to the vector functional units.

4.7.1 Instruction Queues

The scalar unit fetches vector instructions and issues these to the pre-address instruction queue (PAIQ). Provided the PAIQ is not full, the scalar processor can continue execution. If the instruction is a scalar unit read of vector unit state, the scalar processor must interlock any use of the destination register until the data is returned by the vector unit. If the instruction might report an arithmetic trap, the scalar unit must delay further issue until the trap condition is resolved.

For each instruction, the PAIQ holds the instruction encoding and program counter value, plus the

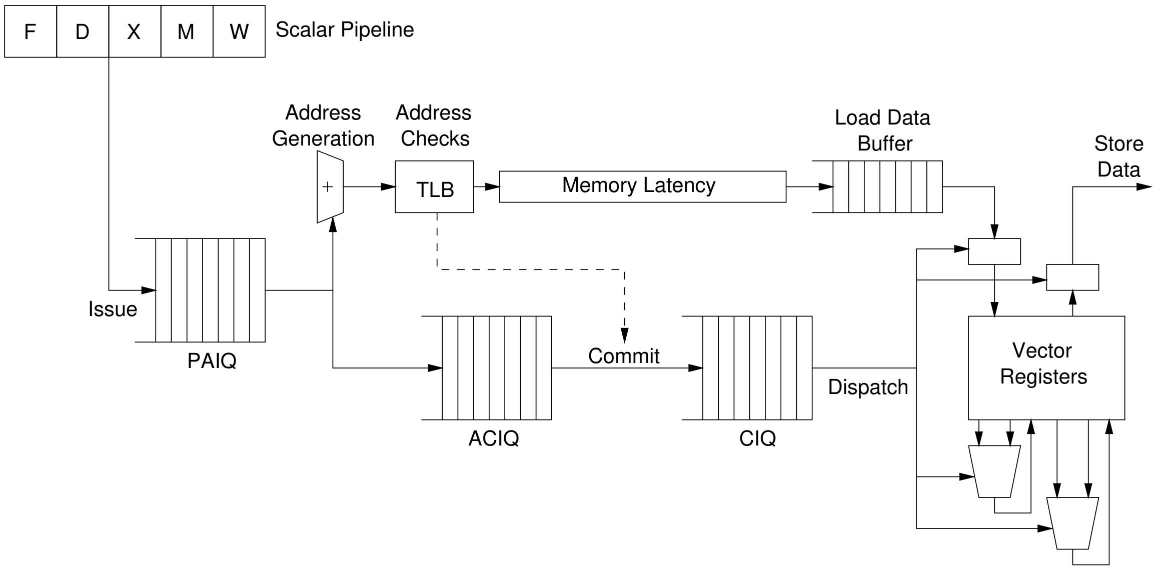

Figure 4.2: Example decoupled pipeline structure. Vector instructions pass through three queues during execution. The scalar unit issues vector instructions to the pre-address check instruction queue (PAIQ). Vector memory instructions are taken from the head of the PAIQ and split into two portions. The first portion begins generating and checking addresses, while the second portion is a register move pseudo-instruction inserted into the address check instruction queue (ACIQ). All other vector instructions are simply passed from the PAIQ to the ACIQ in program order. Once a memory instruction clears address checks, the associated register move portion is passed on to the committed instruction queue (CIQ) along with any following non-memory instructions. From there, instructions are dispatched to the VFUs for execution. If a data page fault is encountered, only instructions in PAIQ and ACIQ need to be saved and restored.

value of the vector length register and any other scalar operands to the vector instruction. Typically, the maximum number of scalar operands is two, for example, base and stride for a strided memory operation. The PAIQ must hold a copy of these values because the scalar unit is free to change vector length and scalar registers after the vector instruction is issued to the PAIQ but before it begins execution.

The vector unit takes instructions in order from the head of the PAIQ. Vector memory instructions begin execution here, while other instructions are simply passed to the second queue, the address check instruction queue (ACIQ). Each vector memory instruction is split into two components, an address generation component that begins execution in the address generators of a VMU at the start of the pipeline, and a linked register move pseudo-instruction which is passed into the ACIQ. The register move pseudo-instruction moves data from the memory data buffers into the vector register file.

The purpose of the ACIQ is to buffer instructions that follow a vector memory instruction until it is known that the vector memory instruction will not generate a data page fault. The wait in the ACIQ represents extra delay for all vector instructions. In particular, the register move for the *first* element of a vector memory operation must wait in the ACIQ until the *last* element address in the vector is checked. But during this waiting period, the address for the first element can proceed down the memory pipeline, allowing the address check latency to be overlapped with memory latency.

Because a vector microprocessor will have short running vector instructions, address checks do not take long. A typical vector microprocessor will have chimes lasting around eight clock cycles, and this is the length of time required to generate and check all the memory addresses in a vector memory instruction assuming there is an address generator and TLB port per lane. This delay is approximately the latency to a large cache and only a small fraction of expected latency to main memory, allowing address check time to be completely hidden behind memory access latencies.

Instructions are committed in-order from the ACIQ to the final committed instruction queue (CIQ). Any non-memory instructions at the head of the ACIQ are simply passed along to the CIQ. The register move component of a vector memory instruction is not dispatched to CIQ until the entire vector of effective addresses has been checked for page faults in the address generators. This ensures that if a vector memory instruction experiences a data page fault, only the state in the PAIQ and ACIQ has to be saved and restored for instruction continuation. Because they occurred earlier in program order than the faulting instruction, any instructions already in the CIQ can complete execution.

On a page fault, one detail is that the register move instruction at the head of the ACIQ corresponding to the faulting vector memory instruction should be dispatched to the CIQ but with a truncated vector length. This truncated register move pseudo-instruction reads out valid data in the memory buffers for element positions before the faulting element. As part of process restart, a register move pseudo-instruction, with a non-zero starting element position, is issued as the first instruction. In the worst case, a single vector memory instruction could experience a separate trap per element.

Also, when the page fault is taken, there may be instructions which write scalar registers (e.g., popcount) buffered in the PAIQ or ACIQ which have created interlocks that would block a read of a scalar register. These interlocks must be cleared when taking the trap to allow the scalar state to be saved. When

Instruction	Issue to PAIQ	Issue to ACIQ	Issue to CIQ	Dispatch from CIQ	Complete
`ld.v`	1	2	14	102	114
`ld.v`	2	14	26	114	126
`fmul.d.vv`	3	15	26	115	127
`ld.v`	4	26	38	126	138
`fadd.d.vv`	5	27	38	127	139
`sd.v`	6	38	50	138	150
`membar`	7	39	50	139	151
`ld.v`	8	50	62	150	162
`ld.v`	9	62	74	162	174
`fadd.d.vv`	10	63	74	163	175
`sd.v`	11	74	86	174	186
`membar`	12	75	86	175	187

Table 4.1: Time line of execution of code example on decoupled pipeline. This machine has a single VMU with a 100 cycle memory latency, and floating-point units with six cycle latency. Note how the memory pipeline is kept busy with a vector length of twelve despite the 100 cycle memory latency. After a vector memory instruction finishes executing, the ACIQ head pointer jumps over multiple instructions to the next vector memory instruction; hence the multiple issues to CIQ in the same cycle.

execution resumes after the page fault, the interlocks can be restored as PAIQ and ACIQ are reloaded.

When a trap is encountered in the scalar pipeline, there may be multiple pending vector instructions held in the PAIQ and ACIQ which could potentially cause vector data page faults. One approach is to save and restore the vector instruction queues as for a data page fault on every scalar trap. Alternatively, because vector data page faults are rare, the hardware can simply allow these queues to drain before servicing the scalar trap. If a data page fault occurs during the draining process, the scalar trap is converted to a vector data page fault. The scalar trap will be handled when execution resumes after servicing the data page fault.

Another scheme, adopted in the DEC Vector VAX [DEC90a], is to associate a process ID with the vector unit state, and to delay saving the state until another process attempts to use the vector unit. If no intervening process attempts to use the vector unit, the vector unit can continue executing instructions from the original process. Any pending vector page fault traps will be handling when the original process is rescheduled, without requiring a queue drain or save of vector unit state.

4.7.2 Execution Example

To illustrate the operation of the decoupled pipeline, consider the example C code in Figure 4.3 and the resulting string of dynamically executed vector instructions.

Table 4.1 shows how the execution of this string of instructions unfolds in time on a single-lane machine with a single VMU with a 100 cycle memory latency, and separate floating-point multiply and add VAUs with six cycle floating-point latencies. Figure 4.4 shows the same information graphically.

Note that the scalar pipeline finishes issuing instructions for the second routine `bar` before the

```
/*
foo(int n, const double *a, const double *b, double *c)
{
    int i;
    for  (i=0; i<n; i++)
    {  c[i] += a[i] * b[i]; }
}

bar(int n, const double *a, double *b)
{
    int i;
    for  (i=0; i<n; i++)
    {  b[i] += a[i]; }
}

static double A[34], B[34], C[34], D[34];

main()
{
    foo(12, A, B, C);
    bar(12, D, C);
}
*/

foo:    ld.v vv1, a1            # Get A values.
        ld.v vv2, a2            # Get B values.
        fmul.d.vv vv3, vv1, vv2 # Calculate product.
        ld.v vv4, a3            # Get C values.
        fadd.d.vv vv5, vv3, vv4 # Calculate sum.
        sd.v vv5, a3            # Store result.
        membar                  # Make store values visible.

bar:    ld.v vv1, a1            # Get A values.
        ld.v vv2, a2            # Get B values.
        fadd.d.vv vv3, vv1, vv2 # Calculate sum.
        sd.v vv3, a2            # Store result.
        membar                  # Make store values visible.
```

Figure 4.3: Sample code to illustrate tolerance to memory latency of skewed pipeline. For brevity only the assembly code for the dynamically executed vector instructions is shown.

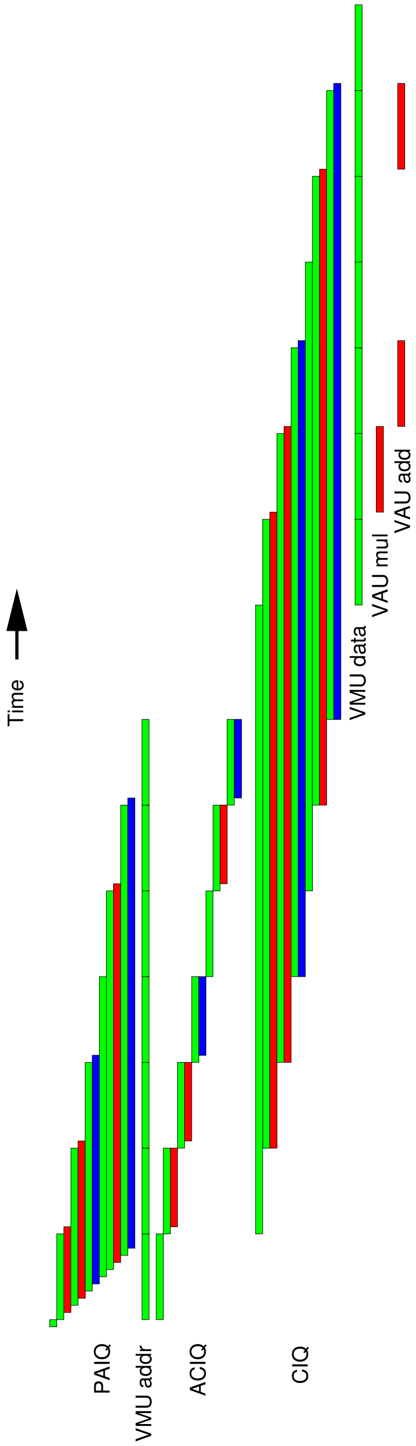

Figure 4.4: Execution of example code on skewed pipeline. Each shaded bar represents the stay of an instruction within the given instruction queue or functional unit. A slice across the trace shows the location of each instruction during each cycle of execution. Note that the scalar unit is free to continue execution after filling the PAIQ in the first few cycles.

second memory instruction in the first routine has starting checking addresses, and long before data for the first load in `foo` has returned from memory. The VMU address and data structures are kept continuously busy during the series of vector memory instructions, although the address and data portions are separated by 100 cycles. Further vector memory instructions could continue to saturate the VMU, even with these relatively short vectors (12 elements) and long memory latency (100 cycles).

4.7.3 Queue Implementation

While the PAIQ, ACIQ and CIQ have been described as if they were three separate structures, in practice they can be implemented as a single circular buffer held in a RAM structure. Four address pointers represent the head of CIQ, the head of the ACIQ, the head of PAIQ, and the tail of PAIQ. Once a vector memory instruction clears address checks, the head of ACIQ can be advanced over multiple instructions in one cycle to point to the next waiting vector memory register move pseudo-instruction (this will be at the head of PAIQ if there is a single VMU). The location of the next enqueued vector memory instruction can be held in a second smaller RAM list.

Figure 4.5 shows the structure of the vector instruction queue. The instruction at the head of CIQ is waiting for register and functional unit hazards to clear before being dispatched. The instruction at the head of ACIQ is the register move instruction for the vector memory instruction currently undergoing address checks. The instruction at the head of PAIQ is the next vector memory instruction waiting to enter the address generation phase.

To sustain throughput of a single vector instruction issued per cycle, the FIFO RAM must support one write of a new instruction, one read from the head of CIQ to dispatch an instruction, and one read from the head of PAIQ to dispatch a memory instruction to the address generators in a VMU. Each entry in these queues requires roughly 29 bytes on a 64-bit processor: eight bytes for program counter, four bytes for instruction, one byte for vector length, and eight bytes for each of two scalar operands. The program counter is used to identify the instruction that caused the trap for debugging purposes, but could possibly be omitted to save area. A queue with the same number of entries as the number of cycles of main memory latency will allow the memory pipeline to remain saturated even with short vectors. With 100 entries total, the queue will require just under 3 KB of storage.

Note that only the instructions in the PAIQ and ACIQ need be saved on a context swap. Address checks take only one short chime and this sets the maximum length of the ACIQ; a typical value might be 8 instructions. The length of the PAIQ might be limited by the hardware, for example, to 16 instructions. Only these 24 instruction slots would then have to saved and restored on context swap. Assuming a high bandwidth path to cache capable of transferring 32 bytes per cycle, saving the state would only require 24 cycles.

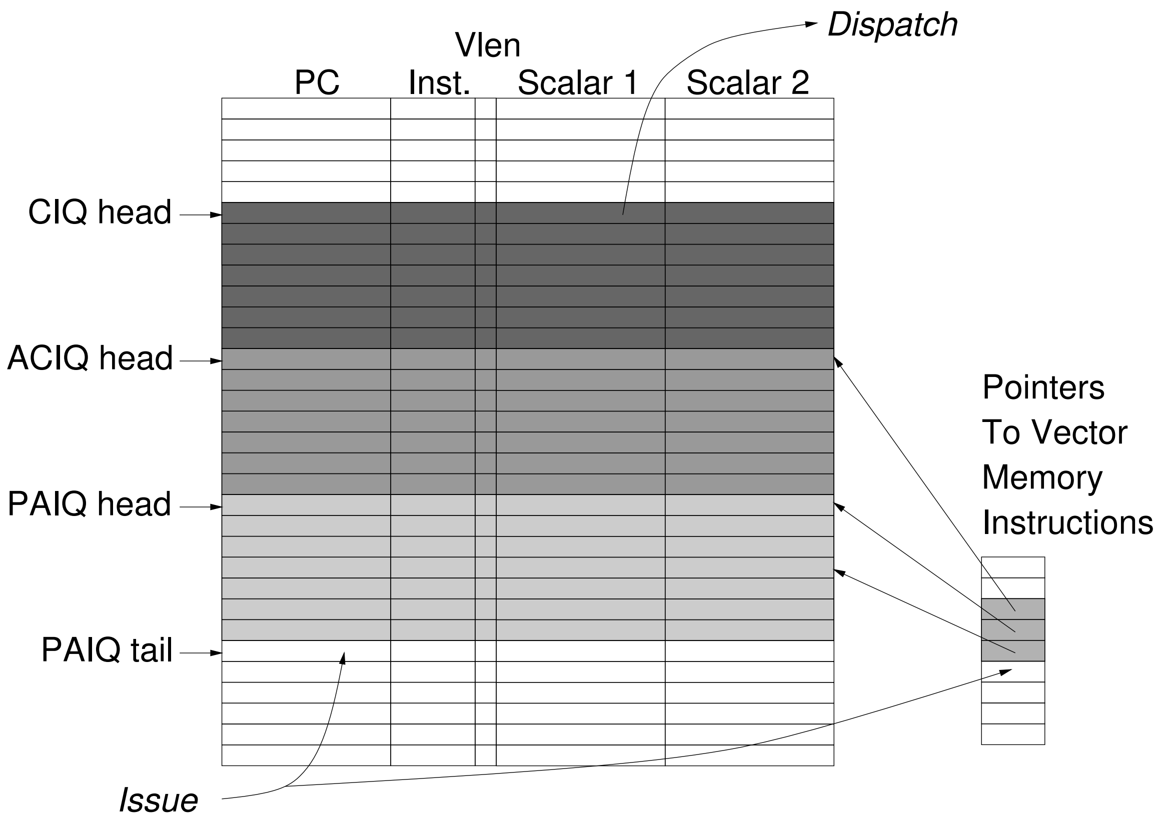

Figure 4.5: The three instruction queue structures can be implemented as a single circular buffer in RAM. Four pointers mark the boundaries between the queues. A second circular buffer holds pointers to the vector memory instructions in the queues.

4.8 Inter-Instruction Latencies

In this section, I consider the cases where memory latency is made visible to vector instructions executing in a dynamically scheduled vector pipeline. Most vector code begins with loads from memory, and so we can expect that the instruction queues will often stretch out to the point where data returns from memory, as shown in Figure 4.6. For most instructions, values are produced and consumed by units executing at the end of the pipeline. This makes these inter-instruction latencies independent of memory latency and dependent primarily on much shorter functional unit latencies. This general behavior occurs either with decoupled or out-of-order vector machines.

There are three classes of instruction that require values produced at the end of the pipeline to be read at the beginning of the pipeline thereby exposing memory latency to software:

- Scalar reads of vector state cannot complete until the value of the vector state is known. The scalar unit can continue execution after issuing the read provided it does not try to use the destination register value before it's received.
- Indexed vector memory instructions cannot begin address generation until the index vector is known.
- Masked vector memory instructions cannot generate page faults until the value of the flag register is known.

Another way in which memory latency is exposed is for scalar operands fed to vector instructions.

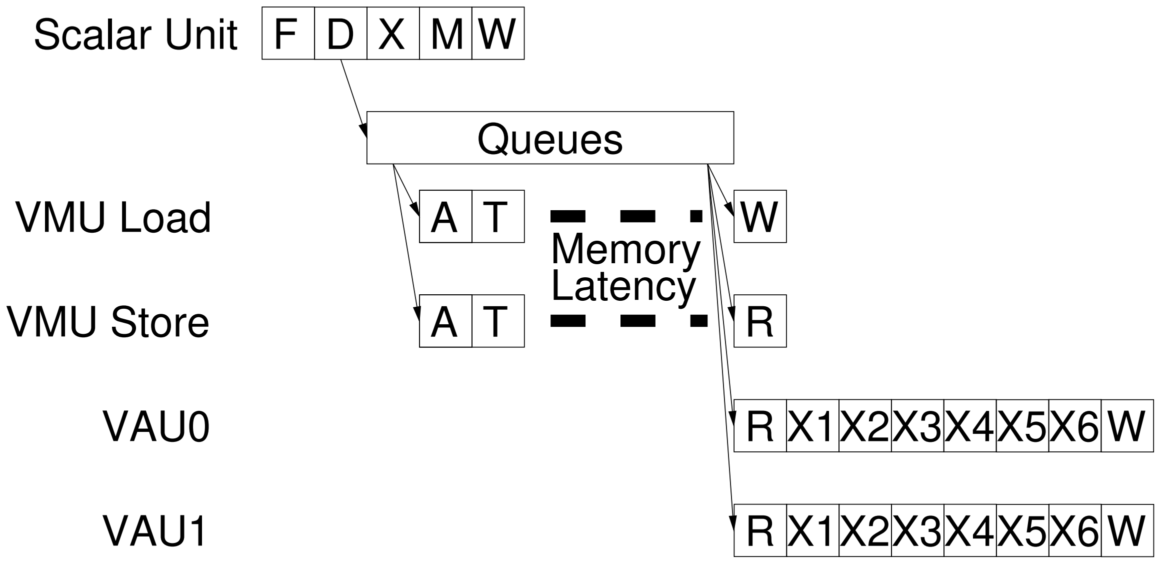

Figure 4.6: In normal operation, the execution of the vector arithmetic units is delayed relative to the scalar unit and vector memory address generation (stage A) and translation (stage T). Most values are produced and consumed amongst functional units executing at the end of the pipeline and are therefore insensitive to memory latency. Only instructions that require values produced at the end of the pipeline to be read at the beginning of the pipeline experience the full memory latency.

Vector codes often include a stream of scalar operands for scalar-vector instructions, which can seriously impact performance if these scalar operands cannot fit in cache and the scalar unit is poor at tolerating main memory latency.

These instructions deserve special attention because they potentially expose long memory latencies to software. Two simple techniques can be used to improve performance. First, where possible, the latencies can be tolerated with longer vectors or by scheduling independent instructions. Second, if the application has a small working set, effective memory latency can be reduced by caching. Some more advanced techniques can further reduce the impact of the long memory latency.

Scalar unit reads of vector unit state are primarily used for two purposes: to help execute partially vectorizable loops, and to affect control flow in loops with data-dependent exits. We can convert more partially vectorizable loops into fully vectorizable loops by adding better support in the ISA for such operations as scatter/gather, compress/expand, and conditional execution. Another approach for handling partially vectorizable loops is to provide a decoupled scalar unit [Smi89] that lives at the end of the memory pipeline in the same way as the vector unit. This approach can also solve the latency problem for scalar operands streams in cases where the scalar address stream does not depend on the scalar values loaded. Where scalar unit reads are used to affect control flow in the scalar unit, branch prediction and speculative scalar execution can be used to help reduce the impact of the read latency. To help hide the latency of the scalar unit reads of vector state in data-dependent-exit loops, Section 6.8 introduces flag priority instruction to make more vector instructions available for static scheduling during the read latency.

Where vector indices are loaded from memory, or calculated based on values loaded form memory, there is a dependency that includes full memory latency; this is the vector form of pointer-chasing. Apart from using longer vectors, scheduling independent instructions, and using caches to reduce memory latency, there is little that can be done to break this dependence.

It is possible to reduce the effects of the latency for masked vector memory operations by speculatively generating and checking addresses. If no page faults are encountered, execution can proceed. If a page fault is encountered, the speculative address generation and checking must wait until flag values are known before taking a trap.

Speculative masked loads bring data values into the load data buffer, and only move these into the register file when the mask values are known later in the pipeline. Speculative masked unit-stride load accesses are unlikely to encounter page faults (assuming the working set fits in memory), and will generate little additional memory traffic assuming some of the words in the unit-stride span are used. Speculative strided load accesses are also unlikely to encounter page faults, but could generate significant excess speculative memory traffic potentially slowing execution due to extra memory system conflicts. Speculative indexed accesses are more likely to encounter page faults (e.g., following NULL pointers), and will also generate significant extra speculative memory traffic.

Speculative masked stores can buffer up the physical addresses in a write address buffer, and only retrieve data values and send these to memory later in the pipeline when the mask values are known. Because they do not access the memory system until after the mask values are known, speculation on masked stores will not generate excess memory traffic.

One possible design point is to speculate on all masked vector memory instructions except strided and indexed load instructions.

The application studies in Chapter 11 report on the frequency of these memory latency-sensitive instructions. The results there show that scalar operands streams are the most common way in which memory latencies could be exposed to vector execution. In all of these cases, however, the scalar address stream was independent of the scalar values loaded and so could be hidden with a decoupled pipeline.

4.9 Interlocking and Chaining

In this section, I consider the design of interlock and chaining control logic for the in-order decoupled pipeline described above. I show that only a small amount of control logic is required to eliminate dead time and provide flexible chaining of all types of hazard, even with a long memory latency.

For correct execution, hardware must preserve data dependencies between vector instructions. The possible hazards between two vector instructions for each vector register operand are: read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW). I assume dependencies on memory operands are expressed explicitly with memory barrier instructions (see Section 8.3 on page 147).

4.9.1 Types of Chaining

Chaining reduces inter-instruction latencies by allowing a vector instruction with a dependency on a vector register operand accessed by an earlier instruction to begin execution before the earlier instruction has completed. The second instruction accesses earlier element positions concurrent with the first instruction's accesses to later element positions.

Chaining is usually used to refer to the case where the second instruction is reading the results of the first instruction, i.e., a vector RAW hazard. In this thesis, I also use chaining to refer to the acceleration of WAR and WAW hazards.

Chaining a WAR hazard (also known as *tailgating* in the literature) involves writing new values into a vector register while a previous instruction is still reading old values. Chaining WAR hazards reduces vector register pressure by allowing vector registers to be reused earlier.

Chaining a WAW hazard involves writing new values into a vector register before a previous instruction has finished writing earlier values. While performance-critical WAW hazards are uncommon in scalar code, they occur frequently in vector code containing conditional updates. For example, two conditional statements representing two sides of a vectorized `if` statement might write different elements within the same vector register, but this would appear as a WAW hazard to interlocking hardware which manages interlocks at the granularity of whole vector registers.

Chaining requires a vector register file capable of supporting multiple concurrent accesses to the same vector register. The element-partitioned vector register files described in Chapter 5 allow flexible chaining of all types of hazard.

4.9.2 Interlock Control Structure

Figure 4.7 shows the structure of the interlock checking hardware for the vector unit shown in Figure 4.2. Interlocks are required in two places. First, at the start of the VMU pipeline. Second, where instructions are dispatched from the CIQ to the VFUs. By splitting these interlocks into two sets, with an instruction queue in between, we avoid reserving vector registers during the long memory latency and similarly avoid tracking most dependencies across this long pipeline delay.

The late pipeline interlock and chaining organization described below is very similar to that used for T0, which has a very short memory latency. The instruction queues and early pipe interlocks are the only additional control logic required to extend T0's interlock and chaining to handle higher memory latencies.

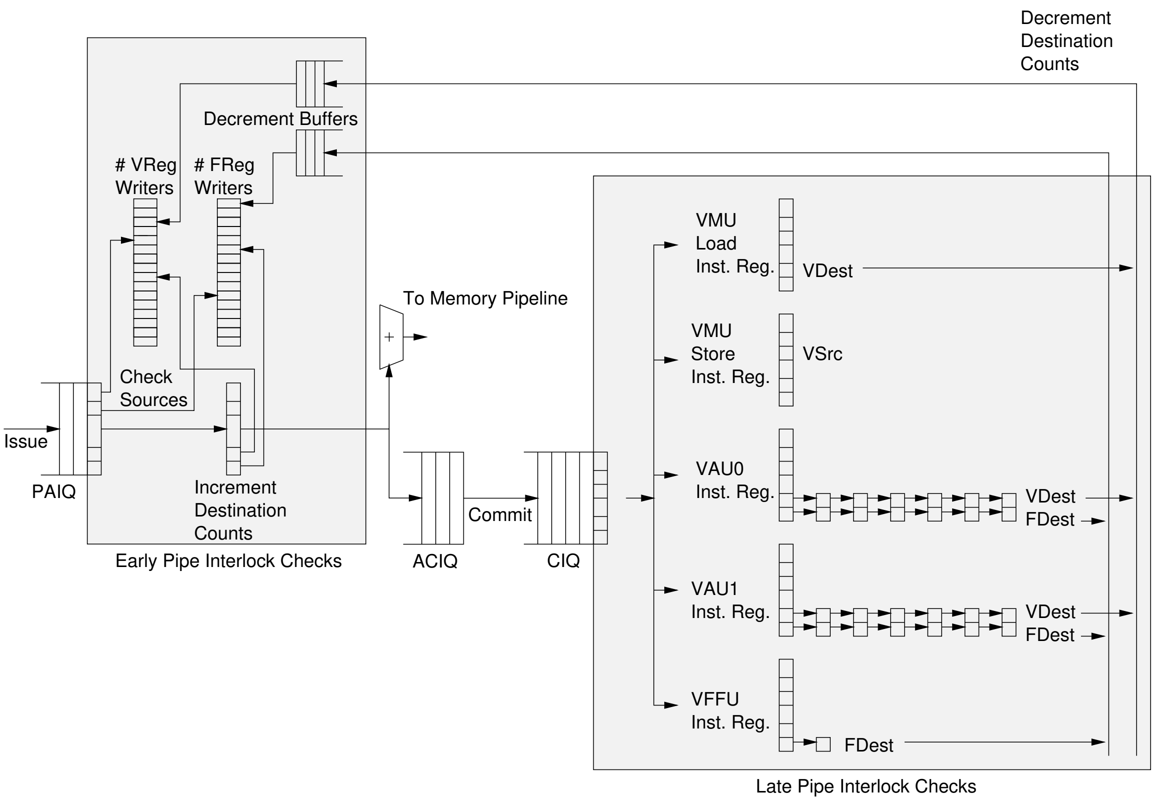

Figure 4.7: Structure of interlock check logic for sample vector pipeline.

4.9.3 Early Pipe Interlocks

The first set of interlocks covers structural hazards on the VMU, as well as a small set of RAW data hazards corresponding to cases where vector state is read early in the pipeline. These include vector indexed memory operations, vector masked operations, and scalar extract instructions that read one element from a vector register. Flag processor operations that return a scalar value are not handled here but in the late pipe interlocks for reasons discussed below in the next section.

To keep track of these RAW hazards, a record of pending writes to vector data and flag registers is kept in a scoreboard as instructions pass from PAIQ to ACIQ. To improve performance, we require that multiple writers of each register can be present in the ACIQ and CIQ. Otherwise, instructions could not enter ACIQ if there was an outstanding write to the same vector register. With long memory latencies, short chimes, and small loop bodies, this would limit loop repeat rate. To allow multiple writers in the queues, the scoreboard keeps track of the *number* of outstanding writes to each register in ACIQ and CIQ. The counts are incremented as instructions enter ACIQ, and decremented at the point where the address generator can chain onto the results produced by the instruction. When a data page fault is encountered, this scoreboard is cleared and instructions in CIQ are allowed to complete. At instruction continuation, the required scoreboard interlocks are restored as the ACIQ is refilled.

The scoreboard can be implemented as two multiported RAMs, one for the vector data registers and one for the vector flag registers. Note that a vector arithmetic instruction can write a vector data register *and* generate exceptional conditions that write vector flag registers. Each scoreboard RAM requires one read port for source interlock checking, one read port and one write port for incrementing destination counts, and one read port and one write port for decrementing destination counts. This gives a total of three read ports and two write ports. The reads can happen in the first phase of the clock cycle, with increments and writes in the second phase. The counter structure is small and so should allow these operations to complete in one cycle.

A check must be made for a simultaneous increment and decrement to the same register, with both writebacks annulled if this is the case. Also, a check must be made that the count is not already at maximum — if so the new instruction cannot be issued to ACIQ but must wait for one of the outstanding writes to decrement the write counter. The number of bits of in each counter should be chosen to limit the performance impact of these write counter overflows.

Even though vector instructions are issued one at a time to the VFUs, multiple instructions might reach the point at which they allow chaining on the same cycle, either because vector lengths changed or because some instructions allow chaining at different points. But the average rate cannot exceed one decrement per cycle, and so small buffers for outstanding decrements can be provided to smooth out bursts in decrement requests. If the buffers could fill, dispatch from CIQ is suspended.

Interlocks of scalar element extract instructions, which read a single element from a vector register, can also be handled in this set of interlocks. It is natural for the scalar extract instruction to use the VMU read port and datapath used for vector indices, since both perform reads in the early part of the pipeline.

Because scalar extracts can randomly access any element from a vector data register, they cannot be chained in the same way as other instructions but should wait until all elements are written. This can be handled in several ways:

- Only decrement the write counters in the scoreboard when an instruction completely finishes execution. This causes a performance loss because chaining is now prevented for indexed and masked vector memory operations.
- Delay scalar extracts after issue for the worst-case number of cycles required to vector the last writer will complete execution. While inexpensive to implement, this penalizes all scalar extracts with added latency.
- Provide a separate set of write counters for scalar extracts which are only decremented as an instruction finishes execution. This incurs additional cost, but gives the best performance.

4.9.4 Late Pipe Interlocks

The second set of interlocks is more complicated. Instructions dispatching from CIQ can alter vector unit state and so all three types of data hazard are possible (RAW, WAR, WAW). There are also structural hazards on the VFUs, on the vector register file access ports, and on the buffer holding write counter decrements.

Flag processor instructions that return a scalar value are also handled in the second set of interlocks. This is because these operations require the use of a VFFU and will typically be chained onto other vector operations, whereas scalar extracts require only a vector register read port and will typically occur in the scalar portion of a partially-vectorized loop when the vector unit is idle.

After instructions are dispatched from the CIQ to the VFU controllers, each instruction is expanded into multiple element groups. Each VFU controller has an instruction register that holds the instruction currently being expanded into element groups. Each active VFU produces one element group's control signals each cycle and these propagate down the VFU control pipelines and execute across the lanes. For the VAUs, the pipeline is several stages long corresponding to the functional unit latency, whereas the VMU register moves are shown here executing in a single cycle.[1]

To remove dead time between different vector instructions executing in the same VFU, it is necessary to provide pipelined control signals so that each stage can be executing a different vector instruction. In addition, it is necessary to keep track of the element group being written in each pipe stage for interlock checking. This requires that each VFU has a shift register that mirrors the datapath pipeline and which holds destination vector data and flag register element group numbers for each pipe stage. In the example, there are six stages in the each VAU pipeline, and so up to six different vector data element groups and/or six different vector flag element groups could have outstanding writes in each VAU pipeline. This machine has a single VFFU, with a single pipe stage.

[1] In practice, it may be desirable to include several cycles of the memory latency in the register move component rather than accounting for all the memory latency before the register move instruction is dispatched. This allows skewing of store vector register reads versus load vector register writes to accommodate differences in the latency for store data versus read data in the memory system.

4.9.5 Chaining Control

Chaining control is simplest for instructions that access operands starting with element group 0 and stepping through one element group per cycle. I term these *regular* vector instructions.

Consider first the case where there are only regular vector instructions and full chaining support. In this case, a new instruction in the example machine would be guaranteed no RAW hazards provided only that there were no outstanding writes to the *first* element group of any of its source operands, either in the VFU instruction registers or in the VFU pipelines. The example machine has no WAR hazards for regular instructions because writes always follow reads across all VFU pipelines. A WAW hazard is possible because the VMU load units and the VFFU have much shorter pipelines than the VAUs. With only regular instructions, a new VMU load or VFFU instruction would be guaranteed no WAW hazards provided there were no writes outstanding to the first element groups of any of its destinations. Once these hazards are cleared, an instruction can be dispatched to a VFU and execute to completion with no further stalls.

Irregular vector instructions are those that do not execute at the same rate and/or do not follow the usual vector register access pattern; these complicate chaining control. Vector memory instructions can experience stalls during execution caused by memory system conflicts or cache misses. If there is only a single VMU, one approach is to stall the entire vector unit whenever the VMU stalls, so preserving the regular structure across all instructions in execution.

An alternative approach is to *regularize* vector memory instructions by using data buffers to decouple vector register file access from the memory system. For vector loads, the register move from the data buffers into the register file is not dispatched from the CIQ until it is known that all elements in the vector will arrive in time to prevent stalling. This can usually be determined early in the pipeline based on effective addresses and so does not necessarily add to memory latency or prohibit chaining. Similarly for vector stores, a write buffer can isolate the drain of data from the vector register file from the write to the memory system.

Some other irregular vector instructions are more difficult to handle. These include the scalar extract described earlier, as well as scalar element insert and vector register permutes (Section 10.3) which access elements with irregular access patterns. Scalar insert can write any element in a vector register according to the element index, and so might stall in dispatch until all outstanding read or write accesses to the target vector register will complete. Vector register permutes can read and write vector element groups in arbitrary patterns.

Some irregular instruction can support some forms of chaining to regular vector instructions. For example, compress can guarantee to read its source at normal rate and so can support RAW or WAR chains on its input. Compress cannot guarantee that the result will be produced at normal rate so cannot be RAW chained on its destination, but can be WAR or WAW chained because it can only produce values *slower* than a regular instruction.

The final important set of irregular instructions are slow arithmetic instructions such as divides and square roots. These can simply reserve destination registers until all values are completed, or until the point at which it is known that a regular instruction could not overtake and violate a dependency.

4.9.6 Interlocking and Chaining Summary

Efficient pipeline control is important for vector microprocessors. By delaying arithmetic instruction execution until data arrives from memory and by eliminating dead time, we remove the main reasons for providing longer vector registers. By hiding memory latency and by providing flexible chaining, we increase the efficiency of executing short vector instructions on parallel lanes, thereby increasing the range of applications which achieve speedups through vectorization.

The control logic required to provide this efficient pipeline control is small, consisting of a few kilobytes of queue information and a few dozen comparators and incrementers, yet allows for sustained execution of dozens of operations per cycle while tolerating a 100 cycle main memory latency.

Chapter 5

Vector Register Files

The vector register file lies at the heart of a vector unit. It provides both temporary storage for intermediate values and the interconnect between VFUs. The *configuration* of a vector register file is the programmer-visible partitioning of the vector element storage into vector registers of a given length. The first part of this chapter discusses the choice of vector register configuration, and in particular, how configurations for vector microprocessors will tend to have more vector registers but with fewer elements per vector register compared to vector supercomputers.

The second part of this chapter reviews several microarchitectural techniques which have been used in commercial systems to reduce the cost of implementing a large high-bandwidth vector register file, and then presents several full-custom VLSI designs suitable for a vector microprocessor. The most area-efficient vector register file design is shown to have several banks with several ports, rather than many banks with few ports as in a traditional vector supercomputer, or one bank with many ports as in superscalar register files.

5.1 Vector Register Configuration

The number and length of vector registers to provide is a key decision in the design of a vector unit. Table 5.1 shows the configuration of vector register files for a variety of existing vector processors. Some machines allow a fixed number of elements to be dynamically repartitioned to form either a large number of short vector registers or fewer longer vector registers.

5.1.1 Vector Register Length

We can place a lower bound on required vector register length if we desire that at least some application codes can keep all of the VFUs busy. This requires that we can issue an instruction to every VFU before the first VFU finishes execution. The peak number of element operations completed per cycle is the product of the number of VFUs and the number of lanes. Equation 5.1 expresses the constraints on minimum

System	Number of vector registers	Vector register length	Element size (bits)	Total storage (KB)	Reference
Vector Supercomputers					
Cray 1, 2, 3, 4	8	64	64	4	[Rus78]
Cray X-MP, Y-MP	8	64	64	4	[ABHS89]
Cray C90, T90	8	128	64	8	[Cra93]
Fujitsu VP-200	8–256	1024–32	64	64	[MU84]
Fujitsu VPP500, VPP300	8–256	2048–64	64	128	[UIT94]
Hitachi S810	32	256	64	64	[LMM85]
NEC SX-2	8+32	256	64	80	[Laz87]
Unisys Sperry ISP	16	32	72^*	4.6	[Cor89]
IBM 3090 Vector Facility	16	128^a	32	8	[Buc86]
Vector Minicomputers					
Cray J90	8	64	64	4	
Convex C1, C2	8	128	64	8	[Jon89]
Convex C4/XA	16	128	64^*	16	[Con94]
DEC Vector VAX	16	64	64	8	[DEC90b]
Ardent Titan	8×32^b	32	64	64	[DHM+88]
Single-chip Vector Coprocessors					
NEC VPP	4+4	64+96	64	5.1	[OHO+91]
Fujitsu μVP	8–64	128–16	64^*	8	[AT93]
CM-5 Vector Units	4–16	16–4	64^*	0.5	[WAC+92]
Vector Microprocessors					
T0	16	32	32	2	Chapter 3

Table 5.1: Vector register file configurations for different vector processors. Several machines allow the element storage to be dynamically reconfigured as a fewer number of longer vector registers or a greater number of shorter vector registers. *The machine can sub-divide these elements into sub-words to allow greater element capacity for operations requiring less precision. ^{a}The IBM 370 vector architecture made the vector length an implementation detail, not visible to the user instruction set architecture. Also, vector registers can be paired to give 64-bit elements. ^{b}The Ardent Titan operating system partitions its register file into 8 process contexts each with 32 vector registers of 32 elements each.

vector register length if we wish to keep all of the VFUs busy:

$$\text{Vector register length} \geq \frac{\text{Number of lanes} \times \text{Number of VFUs}}{\text{Number of vector instructions issued per cycle}} \tag{5.1}$$

In practice, vector code also require some bookkeeping scalar instructions. We can either make the vector registers longer to keep the VFUs busy while the scalar instructions are issued, or supply additional instruction bandwidth to issue scalar instructions in parallel with vector instructions.

There are several advantages to further increases in vector register length:

- Any vector startup overhead is amortized over a greater number of productive cycles. Similarly, energy efficiency can improve because instruction fetch, decode, and dispatch energy dissipation are amortized over more datapath operations.
- Instruction bandwidth requirements are reduced. Also, because each instruction buffers up more work for the vector unit, the scalar unit can get further ahead of the vector unit for a given length of vector instruction queue.
- Longer vector registers will tend to increase the spatial locality within vector memory reference streams, by taking more elements from one array before requesting elements from a different array.
- If the maximum possible iteration count of a vectorizable loop is known to be less than or equal to the maximum vector register length at compile time, then stripmining code can be completely avoided; lengthening the vector registers will increase the probability of this occurring.
- Longer vector registers can reduce memory bandwidth requirements by enabling more scalar operand reuse. For example, in the tiled vector-transpose-by-matrix multiply described in Section 11.1.2, increasing vector register length would linearly reduce the number of scalar memory accesses to the source vector.

We see diminishing returns for further increases in vector register length for two reasons. First, for all sources of fixed overhead per vector (startup cycles, instruction energy consumption, instruction bandwidth, scalar data bandwidth, or memory subsystem address bandwidth) each doubling of vector register length will give ever smaller reductions in the amortized overhead. Second, it is only possible to improve performance for applications with natural vector lengths greater than the existing vector register length and these become fewer as vector register length grows.

5.1.2 Number of Vector Registers

There are several advantages to increasing the number of vector registers:

- Increasing the number of vector registers provides more storage for intermediate values. This decreases the number of vector register spills and thereby reduces memory bandwidth requirements.

Espasa and Valero [EV97a] analyzed vector register spill traffic for several codes taken from the SPECfp92 and PERFECT benchmark suites running on a Convex C3 architecture with 8 vector registers. The three most affected programs were `su2cor` and `tomcatv` from SPECfp92, and `bdna` from PERFECT. For `su2cor`, 20% of all memory traffic was vector register spills, which would mostly be avoided with an increase to 16 vector registers. For `tomcatv`, spills were 40% of memory traffic and could be eliminated with an increase to 32 vector registers. The worst offender was `bdna`, where spills accounted for 70% of all memory traffic. Here the traffic was partly due to vector register save and restore overhead around a large number of calls to vectorized mathematical library routines such as SIN and EXP. Increasing the vector register count might allow a more sophisticated vector function calling convention with both caller-save and callee-save vector registers to reduce spills. Several other codes had spills accounting for 10–15% of total memory traffic.

Chapter 12 reports that increasing the number of vector registers to 32 would improve the performance of several multimedia codes running on T0 by 5–26%, but none of these codes would show significant improvement past 32 vector registers.

In her thesis, Lee [Lee92] found that decreasing the amount of register spill code was not an important reason to increase the number of registers, but her study used a Cray Y-MP vector supercomputer with two load units and one store unit. That the register spill code did not contribute to execution time is an indication that the multiple memory ports were otherwise poorly utilized [Esp97b]. A vector microprocessor will likely operate in an environment with relatively limited memory bandwidth and hence reducing vector register spills is important.

- Matrix operations can make use of more vector registers to hold larger rectangular portions of matrix operands in the vector register file, reducing memory bandwidth requirements by allowing greater operand reuse. For example, in the tiled matrix-by-vector multiply in Section 11.1.2, increasing the number of vector registers would linearly reduce the number of accesses to the source vector.

- A larger number of vector registers makes it easier to perform more aggressive scheduling of independent vector instructions. While register spilling did not contribute significantly to execution time on the Cray Y-MP, Lee [Lee92, Chapter 5] did find that increasing the number of registers improved performance by allowing instruction scheduling to expose a greater degree of inter-instruction parallelism. The independent instructions can then be executed in parallel on multiple VFUs. Similarly, with enough registers, stripmined loops can be unrolled and software pipelined to hide functional unit latencies or a lack of chaining hardware [MSAD91].

For an element storage of fixed capacity, the main disadvantage of increasing the number of vector registers is that more instruction bits are needed to encode the vector register specifier. In Table 5.1, the only machines that have more than 64 user visible vector registers are the Fujitsu supercomputers VP-200, and VPP500/VPP300. The machines without reconfigurable element storage provide between 8 and 32 vector registers. The Torrent ISA [AJ97] demonstrates that a conventional 32-bit RISC instruction format can comfortably support 32 vector registers within the opcode space alloted for a coprocessor.

5.1.3 Reconfigurable Register Files

Having examined the advantages of both longer vector registers and more numerous vector registers, the appeal of a reconfigurable vector register file is clear. Where there are many temporary values, the vector register file can be configured to provide more vector registers to avoid register spills. Where there are few temporary values, the register file can be configured to provide longer vector registers to better amortize overheads. But there are several disadvantages to a reconfigurable vector register file:

- Control logic to manage vector register dependencies, chaining, and vector register file access conflicts is more complex.
- The configuration must either be held as extra state in the processor status word or be encoded in extra instruction bits.
- If the configuration is part of the machine state, a routine must set the configuration before issuing vector instructions (unless it can determine the previous configuration, or requires no more than the minimum possible number of vector registers *and* no more than the minimum possible vector register length).

5.1.4 Implementation-dependent Vector Register Length

One of the goals of ISA design is make an architecture scalable, so as to take advantage of technology improvements over time and to enable both low-cost and high-performance implementations within the same technology generation. From Equation 5.1 on page 85, we see that as we scale up the number of lanes or the number of functional units, we must either increase vector instruction issue bandwidth or increase vector register length to sustain peak throughput. Increasing vector instruction issue rate allows the same vector length to be maintained across a range of implementations, but this can require expensive issue logic. Rather than fix a vector register length for all implementations, we can instead make the vector register length implementation dependent. High-performance many-lane vector units can then use longer vector registers than low-cost few-lane vector units.

For compatibility, an efficient run-time scheme is required to allow the same object code to run on machines with different vector register lengths. A machine's standard ABI (application binary interface) could define a dynamically-linked library holding constants related to machine vector register length. This requires no hardware support, but does require an operating system capable of supporting dynamically-linked libraries. Also, loading this information from memory before using it to make the first control decision for a stripmined loop adds startup overhead.

The IBM 370 vector architecture supported implementation vector register lengths between 8 and 512 elements, and added a "load vector count and update" (VLVCU) instruction to control stripmine loops

[Buc86]. This instruction has a single scalar register operand which specifies the desired vector length. The vector length register is set to the minimum of the desired length and the maximum available vector length, and this value is also subtracted from the scalar register, setting the condition codes to indicate if the stripmine loop should be terminated. This instruction has two problems for a modern RISC pipeline. First, it places a comparison and a subtraction in series which is unlikely to complete in a single machine cycle. Second, it requires several additional instructions to determine the vector length currently being processed; this length is often required in more complex vector code to update address pointers or to check for early exits from speculative vector loops.

An alternative scheme more appropriate for a RISC pipeline is to provide a "saturate to maximum VL" scalar instruction. This "`setvlr` <*dest*> <*src*>" instruction takes a scalar register as the only source operand, and writes a destination scalar register with the minimum of VLMAX and the scalar source value treated as an unsigned integer. The same result value is simultaneously written to VLR. When the saturated vector length is not required in a scalar register, the result can be dumped to the scalar zero register. The use of such an instruction is shown in Figure 5.1. Although a separate subtract instruction is needed with this scheme, the subtract can be scheduled after vector instructions have been dispatched to the vector unit. Note that the ISA should specify that vector instructions perform no operations if VLR is set to zero.

The `setvlr` operation requires only the evaluation of one wide precharged NOR gate before driving multiplexer control lines over a few bits of the datapath width. The precharged NOR gate checks for the case where the scalar value is greater than VLMAX $-$ 1 by looking for any set bits at the top of the scalar operand. The high bits of the result are always zero, so the output multiplexer only needs to drive either the scalar operand or VLMAX onto the low bits of the result bus. The whole operation should evaluate in less than a single processor cycle. The zero comparator can be part of the branch or conditional move zero comparators in the integer datapath.

While writing both VLR and a scalar register with the same result is straightforward for an in-order machine, for machines that perform register renaming it can be troublesome for an instruction to produce two results. The operation can be split into two parts, one that performs the saturation in a scalar register and a conventional move-to-VLR instruction, but these then have a true dependency which adds a cycle of latency before vector instructions can be dispatched.

The `setvlr` instruction can also be used to provide a branch prediction hint for stripmined loops. Loop-closing branches in stripmined vector loops are much more difficult to predict than those in scalar versions of the same loops because there are many fewer taken branches for every not-taken branch. While branch mispredicts might not impose a large penalty in highly vectorized code with long vectors, in mixed scalar and vector code, accurate stripmined loop branch predictions can avoid polluting scalar branch prediction tables with vector loop branches. The next backward branch after a `setvlr` instruction is predicted taken if the scalar source of the `setvlr` instruction was greater than VLMAX.[1] This provides perfect prediction

[1] In addition to the existing comparator that detects that the source was greater than VLMAX $-$ 1, another comparator is required to check if the source is exactly equal to VLMAX. This second comparator requires a wired-OR across the least significant bits of the source.

```
/*
     C source for routine:

     foo(size_t n, const double*A, const double*B, double*C)
     {
         size_t i;
         for (i=0; i<n; i++)
             C[i] = A[i] * B[i];
     }
*/

 ### Register allocation at function entry:
 ### a0 holds n
 ### a1 holds A pointer
 ### a2 holds B pointer
 ### a3 holds C pointer

foo:
        setvlr t0, a0  # Write vlr and t0 with strip length.
        ld.v vv0, a1   # Load A vector.
        sll t1, t0, 3  # Multiply vector length by 8 bytes.
        addu a1, t1    # Increment A pointer.
        ld.v vv1, a2   # Load B vector.
        addu a2, t1    # Increment B pointer.
        fmul.d.vv vv2, vv0, vv1 # Multiply.
        sd.v vv2, a3   # Store C vector.
        addu a3, t1    # Increment C pointer.
        subu a0, t0    # Subtract elements completed this time.
        bnez a0, foo   # Continue loop if any elements left.

        j ra           # Return from subroutine.
```

Figure 5.1: Example of stripmine code showing use of `setvlr` instruction to make object code independent of implementation vector length. Only a single non-branch instruction is required before issuing the first vector instruction in the routine.

for the loop-closing branch in most stripmined loops. As described below in Section 6.7, where there are data-dependent loop exits, forward branches may appear before the loop-closing branch in stripmined code. To handle this case, the `setvlr` branch prediction is ignored after any taken forward branches.

To manage implementation-dependent vector lengths, it is also desirable to provide instructions to read VLMAX and $\log_2(\mathrm{VLMAX})$ into a scalar register. The value $\log_2(\mathrm{VLMAX})$ can be used as a shift amount to perform multiplies and divides by VLMAX. Strides need to be multiplied by the number of elements to update pointers in a stripmined loop, while iteration counts sometimes need to be divided by the number of elements to calculate stripmine loop trip counts. An implementation can simplify instruction decode by converting these instructions that read machine constants into "load immediate" instructions at instruction cache refill time.

In some cases, vector lengths are known at compile time. To avoid stripmining overhead for short constant length vectors, it is desirable to architect a minimum vector register length that must be provided by all implementations.

It is possible to support implementation-dependent vector register length with a reconfigurable vector register file. The configuration would first be set by selecting the number of vector registers, then the above techniques can be used to determine configuration vector lengths.

5.1.5 Context-Switch Overhead

Both lengthening vector registers and adding more vector registers increases the amount of processor state that must be saved and restored on a context switch. Although vector machines have high memory bandwidth to help speed these transfers, they might still represent appreciable overhead. Several techniques can be used to minimize this impact. All of these techniques keep information about the usage of the vector register file in separate hardware state bits.

One technique is to add a hardware valid bit for each vector register. The valid bits would be cleared on process startup, and set whenever a vector instruction writes a vector register. This approach reduces register save and restore overhead for a process that doesn't need to use all the vector registers. A user-level "vector state invalidate" instruction can be provided, whose semantics are to make the value of all vector state undefined. This instruction clears all the valid bits to inform the kernel that the user process no longer requires the state in the vector registers. Programming conventions usually classify vector register state as caller-save, and so software can add a vector state invalidate instruction at the exit of every subroutine.

Further reductions in register save overhead are possible if we also keep track of a hardware dirty bit per vector register which is set whenever a vector register is written. The operating system kernel can check these bits and only save those registers that have been modified since the last context swap. These enhancements are particularly useful for real-time multimedia applications where high-priority processes are invoked repeatedly to process frames of data but do not need vector register state preserved between activations. To support efficient user-level context switching, the dirty and valid bits should be made available to user code. Both valid and dirty bits were part of the IBM 370 vector architecture [Buc86].

The valid and dirty bits work at the granularity of whole vector registers. To also avoid overhead on processes that use only short vectors, we can keep track of the maximum vector register lengths used by a process. We can keep a single maximum length for all vector registers, an individual maximum length for each vector register, or some intermediate grouping of vector registers to maximum length values. To save logic, we may only track the high order bits of the maximum vector length, for example, by only counting the number of element groups. The kernel can then use only the observed maximum length to save and restore a vector register.

5.1.6 Vector Register File Configuration for a Vector Microprocessor

Vector supercomputers have vector register files built from multiple small discrete SRAM components interconnected to the functional units using further discrete parts [Lee92, Chapter5]. The main factors in determining cost are the number of packages and the number of pins per package. With current packaging technologies, the cost of a small SRAM will be limited by the number of input and output pads required, not by the size of the on-chip memory array. Adding more words to a small SRAM array will have a minor effect on cost because each additional address pin doubles the number of storage elements that can be placed on a die. Because longer vector registers increase performance in their designs and because the incremental cost is low, some of the vector supercomputer manufacturers provide very large vector register files holding 64–128 KB (Table 5.1).

For a full-custom VLSI microprocessor, die area dominates cost. As will be shown later, the vector register file area is primarily determined by the number of storage cells and the number of ports for the VFUs. The following arguments suggest that it is best to divide element storage into a fixed large number of vector registers, rather than either fewer longer vector registers, or a reconfigurable vector register file:

- The higher integration of a vector microprocessor makes it easier to eliminate dead time (Section 3.4). This removes the main advantage of providing longer or reconfigurable vector registers because there is now no startup overhead to amortize.

- A fixed vector register configuration simplifies control logic, and avoids the overhead of setting the configuration before use.

- Adding more vector registers decreases the number of vector register spills, and vector microprocessors may have limited memory bandwidth compared to vector supercomputers.

- Increasing the number of vector registers allows the use of more aggressive loop unrolling and software pipelining techniques to expose more independent vector instructions. Future technology scaling will allow more VFUs to be added to exploit this inter-instruction parallelism.

- Short fixed vector register lengths help reduce maximum interrupt latency, which can be important for real-time applications.

- While longer or reconfigurable vector registers can also reduce memory bandwidth requirements with certain patterns of data reuse, this effect can always be synthesized by grouping multiple shorter vector registers in software. In contrast, longer vector registers cannot be used to synthesize the reuse patterns possible with more numerous vector registers (register spills are just one example of this).
- We can unroll a stripmined loop and schedule together multiple vector memory instructions for the same memory stream to get the same memory reference locality benefits as with longer or reconfigurable vector registers.
- While longer vector registers buffer more work per instruction, and thus allow the scalar unit to get further ahead of the vector unit, we can increase vector work buffering less expensively by enlarging the vector instruction queue.

If an application has longer natural vector lengths, then longer or reconfigurable vector registers give greater energy efficiency by executing fewer instructions per operation, but as vector register length grows, the savings per element become less significant. Longer or reconfigurable vector registers are also more effective at avoiding stripmine code when a loop's maximum iteration count is known at compile time. But in practice these tend to be short loops, and for longer loops there is relatively less advantage to removing stripmining overhead.

To summarize, a vector microprocessor ISA should favor a fixed large number of vector registers over fewer longer vector registers or a reconfigurable vector register file. Ideally, the vector register length should be implementation dependent to allow scaling. An implementation should ensure vector register length is sufficient to saturate all VFUs given the available vector instruction issue bandwidth. Further increases in vector length can bring some benefits but these should be weighed against increases in die area, interrupt latency, and context switch overhead.

5.2 Vector Register File Implementation

A vector register file must provide storage for register elements and access ports for VFUs. The number of register elements is determined by the number of vector registers and their length. The number of ports is determined by the mix of functional units. Each VAU usually requires two read ports and one write port. Each load unit requires a write port and each store unit requires a read port. In addition, a vector memory functional unit requires an extra read port to supply indices for indexed loads and stores.

For example, T0 has two VAUs and one VMU that performs both loads and stores. Each VAU has two read ports and one write port. The VMU has separate read and write ports. Both VMU ports are used for indexed loads and vector extract instructions, while the single VMU read port is time multiplexed between indices and data for indexed stores. This gives a total of 5 read ports and 3 write ports.

A straightforward, but expensive, implementation of a vector register file would be as a single multiported memory. This would give each VFU independent access to any element of any vector register at any time. We can reduce implementation costs considerably by using two complementary techniques that take advantage of the regular pattern in which vector instructions access register elements. First, we can split the vector register file storage into multiple banks, with each bank having fewer ports. Second, we can make each bank wider to return more than a single element per access.

5.2.1 Register-Partitioned versus Element-Partitioned Banks

There are two orthogonal ways in which we can divide the vector register storage into banks. The first places elements belonging to the same vector register into the same bank; I term this *register partitioned*. The second places elements with the same element index into the same bank; I term this *element partitioned*. Figure 5.2 illustrates both forms of vector register file partitioning for an example vector register file with 4 vector registers each 8 elements long. Figure 5.2 also shows a hybrid of register-partitioned and element-partitioned. All configurations have been divided into 4 memory banks.

By interleaving vector register storage across multiple banks, we can reduce the number of ports required on each bank. A separate interconnection network connects banks and VFU ports. In effect, all of these bank partitioning schemes reduce the connectivity between element storage and VFU ports from a full single-stage crossbar to a more limited two-stage crossbar. This introduces the possibility of element bank access conflicts between two VFUs.

With a register-partitioned scheme, a VFU reserves a bank port for the duration of an instruction's execution. If another VFU tries to access the same bank using the same port, it must wait for the first instruction to finish execution. Figure 5.3 illustrates this stall for the register-partitioned vector register file shown in Figure 5.2.

Because a single VFU monopolizes a single bank port, opportunities for chaining through the register file are limited to the number of available ports on a single bank (plus any additional bypass paths around the functional units themselves). Also, complications arise in a register-partitioned scheme if there is only a single read port per bank and there is more than one vector register per bank. The hardware will either be unable to execute a vector instruction with both vector register operands in the same bank, in which case software must avoid this condition [Lee92, Chapter 6], or the instruction will run at half speed (due to time-multiplexing of the single read port) which will complicate chaining with other instructions. Similarly, if a bank can only support one read *or* one write, then a vector register cannot be used as both a source operand and a destination operand within a single vector instruction.[2]

With an element-partitioned scheme, a VFU executing a single instruction cycles through all the banks. This allows all VFUs to be performing multiple chained accesses to the same vector register even though each bank has a limited number of ports. Figure 5.4 shows how an element-partitioned scheme avoids the stall experienced by the register-partitioned scheme for the same pair of vector instructions shown in

[2]The Cray-1 implementation had a register-partitioned scheme with only a single read or write port per vector register. Users soon discovered that if a vector register was used for both input and output that the machine would recirculate function unit outputs back into the inputs in a way that could be used to speed reduction operations. Some users were disappointed when the Cray X-MP design added separate read and write ports to each vector register bank, and so lost this capability [ABHS89].

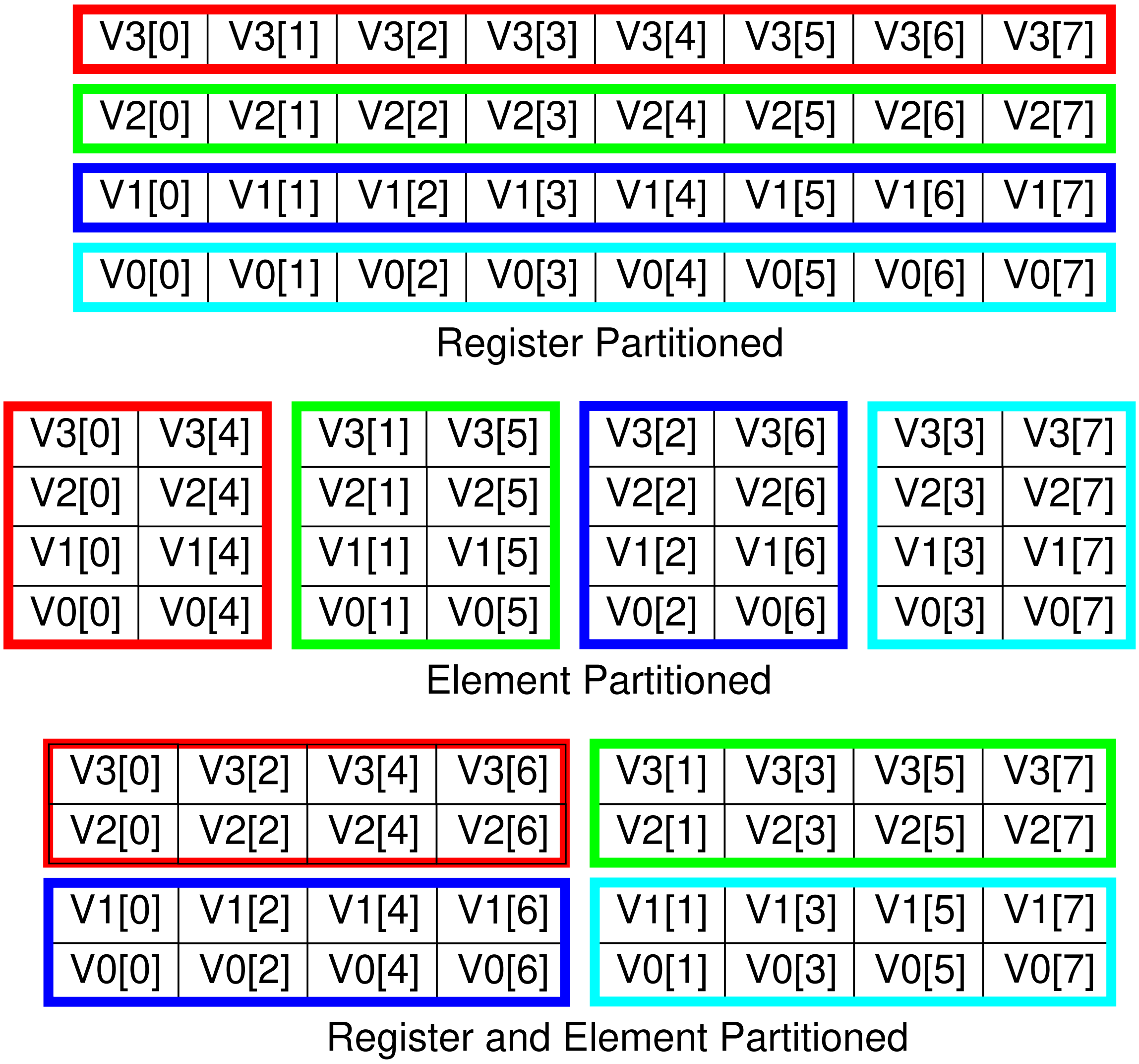

Figure 5.2: Different ways of partitioning the element storage within a vector register file into separate element memory banks. In this example, the vector register file has 4 vector registers, V0–V3, with 8 elements per vector register, 0–7, and there are a total of four element memory banks. The element banks are represented by the thick lines in the diagram. The first scheme is register partitioned, with each vector register assigned to one bank. The second scheme, element partitioned, stripes vector registers across all banks, with corresponding elements assigned to the same bank. The third scheme is a hybrid of the two.

Figure 5.3. This element-partitioned scheme has two read ports per bank, which would also have avoided the conflict in the register-partitioned scheme; however, the element-partitioned scheme would allow further VFUs to read the same register without any additional read ports on each element bank.

If there are more elements than banks, then the VFU will make multiple accesses to each bank. A subsequent vector instruction beginning execution in a second VFU might have to stall while the first VFU performs an access to the first bank, but this stall will only last for one cycle. After this initial stall, both VFUs are now synchronized and can complete further register file accesses without conflicts. An example of such a conflict is shown in Figure 5.5.

One complication with an element-partitioned scheme is that some operations (vector-vector arithmetic and indexed stores) require two elements with the same index but from different vector registers each cycle. There are three possible approaches to solving this problem:

- The simplest approach is to have at least two read ports on each bank as shown in the examples above.
- Another solution is to read element 0 of the first vector operand in one cycle and hold this in a buffer, then read element 0 of the second operand in the next cycle while simultaneously reading element 1 of the first operand from the second bank and into a buffer [IBM87]. This adds one cycle of startup latency if there are no conflicts with other ongoing vector instructions but thereafter the vector accesses complete at the rate of one per cycle.
- If the number of banks is at least equal to the number of vector registers, then a third approach is to place element 0 of each vector register in a different bank [DEC90b]. This stripes vector elements over the banks in a helical barber-pole pattern and allows element 0 of both operands to be obtained from element banks with single read ports, without any additional startup latency, and with subsequent accesses cycling across the banks as before.

The main disadvantage of element partitioning versus register partitioning is that control is more complicated, especially in the presence of VFUs that run at different rates, or that experience different patterns of stall cycles. Stalls are usually only generated for memory accesses by VMUs, but these will also cause any chained VAUs to stall.

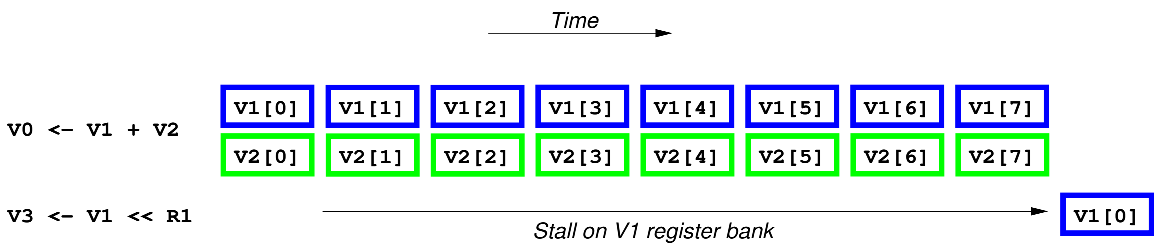

Figure 5.3: Example of vector register file access stall in register-partitioned scheme. If there is only a single read port on the bank holding `V1`, the shift instruction must wait until the end of the add before it can begin execution. Only vector register read accesses are shown.

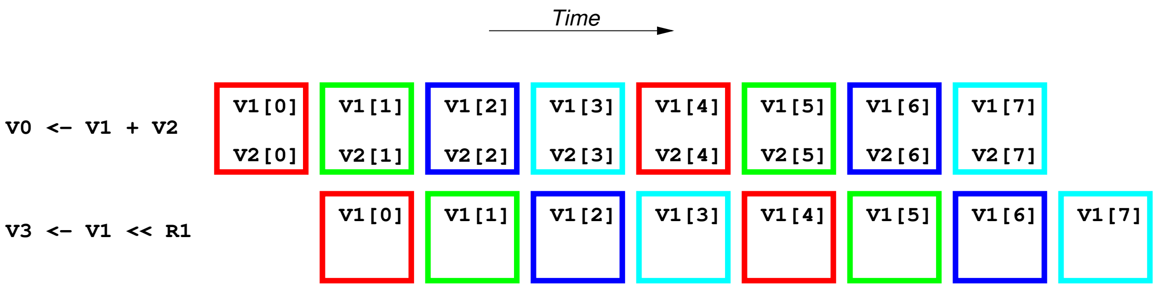

Figure 5.4: The element-partitioned scheme avoids the stall seen by the register-partitioned scheme.

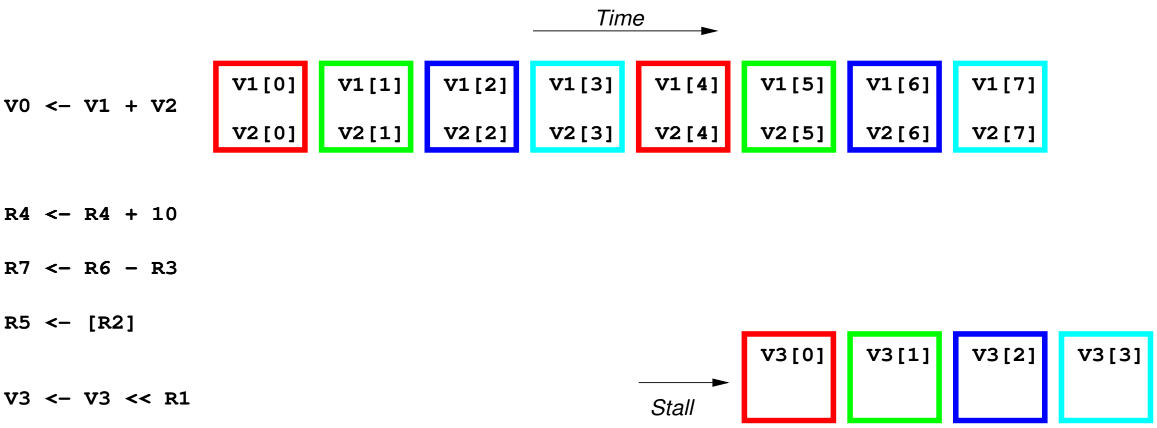

Figure 5.5: This instruction sequence causes one cycle stall with the element-partitioned schemes because the second vector instruction needs to access the element 0 bank on the same cycle as the first vector instruction is returning to the same bank to retrieve element 4. After the single cycle stall, the VFUs are synchronized and experience no further conflicts.

For systems with a single VMU, a simple solution is to stall the whole vector unit on any memory stall thereby preserving the relative access pattern of all VFUs. This allows all potential vector register access conflicts to be resolved once at instruction issue time.

Another simple approach is to use a fixed rotating schedule to connect element banks to VFUs [AT93]. Again, this allows conflicts to be resolved at issue time, and also guarantees good performance when all VFUs are active. The main disadvantage of this approach is that it increases the length of any register access conflicts stalls. The startup latency increases because a VFU has to wait until the schedule rotates around to give it access to the first bank, even if no other VFU was currently active. For example, a system with 4 banks may have up to 3 cycles of wait time until a VFU can access the first bank. This performance impact can be reduced by allocating slots such that two VFUs that are often chained together (e.g., load to arithmetic) perform writes then reads in successive time slots. The time spent waiting for the access slot is then overlapped with the time spent waiting on the dependency. A more significant source of performance loss can occur during execution. If a VMU experiences a stall, it might have to wait for the element bank access port to cycle around again. While it is possible to simply stall the whole vector unit to avoid the long stall for systems with one VMU, for systems with multiple VMUs, it is desirable to allow one VMU to slip ahead of another in the case that both are hitting the same sequence of memory banks.

An alternative approach that still allows all conflicts to be resolved at issue time is to disallow chaining of loads into other operations and provide separate dedicated ports on each bank for VMU accesses. No VMU stall can then affect the operation of other VFUs. Stores can still be chained onto other operations.

The most general solution is to resolve access port conflicts cycle by cycle, giving priority to the oldest instructions for any stall. One problem with this scheme is that a dynamically assigned port access schedule may be suboptimal, i.e., a certain dynamic assignment of VFUs to element banks might prevent a further VFU from beginning execution whereas a different assignment could support all simultaneously. Increasing the number of available bank ports relative to the required number of VFU ports decreases the probability of hitting a bad port access schedule (as does providing separate read and write ports).

5.2.2 Wider Element Bank Ports

Another way of increasing vector register file bandwidth is to make each access port wider, transferring more than one element per cycle. Each VFU read port needs a buffer to hold the multiple words read per cycle until they are needed, and each VFU write port needs a buffer to hold result elements until a whole line of elements can be written to the element storage. An example using this scheme is described below in Section 5.2.4.

The main drawback of this scheme is that it can increase the latency of chaining through the register file, as a vector register read must wait for all elements in a row to be written. Bypass multiplexers can help by allowing a dependent instruction to read a value as it is written to the write buffer. This will only work if the chained instruction issues at the right time, or if the write buffer has only two elements. For example, with four elements per write buffer, if the dependent instruction issues after the first element has been written

System	Number of Vector Registers	Vector Register Length	Register Banks $\times$ Element Banks	Ports per bank	Cycles per access	Elements per access
Cray-1	8	64	8×1	1R or 1W	1	1
Cray X-MP, Y-MP	8	64	8×1	1R and 1W	1	1
Cray-2 patent	8	64	8×4	1R or 1W	2	1
IBM 3838	16	64	1×4	1R or 1W	1	1
Unisys Sperry 3068-0	16	32	1×8	2R or 1W	1	1
Stellar	4×6	32	1×1	1R or 1W	1	16R or 8W
Titan	8×32	32	4×1	1R or 1W	1	1
DEC Patent	16	64	1×16	1R and 1W	1	1
Fujitsu μVP	8–64	128–16	1×4	2R and 1W	1	1
NEC VPP	4+4	64+96	8×1	1R and 1W	1	1
CM-5 VU	4–16	16–4	1×1	5R and 3W	1	1
T0	16	32	1×1	5R and 3W	1	1

Table 5.2: Details of vector register file implementations.

to the write buffer, the first element will already have passed over the write bus and so the second instruction will have to wait three cycles before the write buffer is written to the element bank. With only two elements per port, the dependent instruction can always either read the value from the element bank or catch the value passing over the write bus.

The use of wide access ports can be combined with any bank partitioning scheme. The available read or write bandwidth from a vector register file must be at least equal to the bandwidth required by the attached VFUs. The following inequalities summarize the requirements:

$$\text{total number of banks} \times \frac{\text{read ports}}{\text{bank}} \times \text{width of read ports} \geq \sum_{\text{VFUs}} \text{read ports}$$

$$\text{total number of banks} \times \frac{\text{write ports}}{\text{bank}} \times \text{width of write ports} \geq \sum_{\text{VFUs}} \text{write ports}$$

Table 5.2 gives vector register file implementation details for several vector processors.

5.2.3 Element Bank Partitioning for a Vector Microprocessor

A vector microprocessor is likely to be configured with a large number of vector registers and with vector lengths not much longer than the minimum required to keep the VFUs busy with the available instruction bandwidth. We can rearrange Equation 5.1 on page 85 to determine the minimum number of elements per vector register per lane required to enable a given vector instruction issue rate to saturate the VFUs:

$$\text{Register elements per lane} \quad = \quad \frac{\text{Vector register length}}{\text{Number of lanes}} \qquad (5.2)$$

$$\geq \frac{\text{Number of VFUs}}{\text{Number of vector instructions issued per cycle}} \tag{5.3}$$

A typical single-issue vector microprocessor might have roughly 3–6 VFUs (including VAUs, VMUs, and VFFUs), and so require at least 4–8 elements per lane per vector register. If we assume the machine has 32 vector registers, we get a total of 128–256 register elements per lane. An alternative view is that each lane will hold 4–8 virtual processors, with each virtual processor possessing 32 registers.

As will be shown below, the vector register file within a lane will only contain a few banks (2–4), but a vector microprocessor will have many vector registers (16–32). This represents a disadvantage for a register-partitioned scheme. Multiple vector registers must share the same bank which increases the number of potential register access conflicts. With a register-partitioned scheme, each of these conflicts can last for the entire duration of a vector instruction's execution. Adding more ports to each bank will reduce the severity of these conflicts but is inefficient if the existing port bandwidth would otherwise suffice. While a compiler can be made aware of the vector-register-to-bank mapping and attempt to avoid conflicts, this exposes software to low-level implementation details which will likely change in future designs.

An element-partitioned scheme is better suited to a large number of vector registers with few elements per lane. In effect, the ports on all banks are time-shared across all ongoing vector instructions. While an element-partitioned scheme experiences its own forms of register file access conflict, these are less restrictive and only require at most a few clock cycles to resolve. Again, a compiler could try to schedule around the conflicts present in a given implementation. There is additional control complexity with an element-partitioned scheme, particularly in the presence of load-chaining and multiple VMUs, but this represents relatively little overhead in terms of die area. For these reasons, I assume an element-partitioned scheme in the following.

The width of the access ports is primarily determined by detailed datapath layout considerations. For example, to reduce datapath wiring area the vector register file will likely be pitch-matched to the functional unit datapaths. Multiple small multiported SRAM cells can fit into a typical datapath bit pitch, and this would tend to favor retrieving several words per element bank access.

5.2.4 VLSI Implementation of Vector Register Files

In this section, I compare five designs for a vector register file suitable for a vector microprocessor. The purpose of working through these designs is to provide estimates of the area required for a vector register file in a modern CMOS process, and to show the effects of various design parameters.

The general layout of all the vector register file designs is as shown in Figure 5.6. Vector register file storage is partitioned across all the lanes. The register file address decoders are shared, and drive global decoded word select lines across all lanes. The global word selects are gated locally with a per-lane enable signal to form local word selects. This hierarchical select distribution reduces broadcast delays by minimizing the load on the global word lines, and also saves energy in inactive lanes. Broadcasting decoded global word lines is more energy efficient than broadcasting encoded addresses because there is no duplication of decoding circuitry and because each cycle only a single decoded word line will transition per bank per port. Because the vector register file storage is simply replicated across each lane, I describe just one lane in the following.

Figure 5.7 shows layout for a single lane slice of the vector register file. Each lane's vector register file is split into some number of element banks, with each bank having a few ports. At the periphery of each bank is the circuitry that connects the bank ports to the VFU ports. The read ports from each bank are connected via tristate buffers to the read port busses for each VFU, with each bank having at least one tristate buffer for each read port bus. The write port busses from each VFU are connected via multiplexers to the write ports of the bank, with each bank having one multiplexer input for each write bus. Though not shown in the figure, multiplexers could be added to bypass the register file storage by allowing result data on the write busses to be passed directly into the read ports.

There is a tradeoff between ports per element bank, and element banks per lane. With more ports per bank, larger multiported cells are required but there are fewer banks and hence less overhead circuitry. With fewer ports per bank, each cell is smaller, but more banks are required and hence more bank overhead circuitry. In the following, I compare five alternative designs for the storage within one vector register file lane, varying the number of banks and ports per bank.

Table 5.3 presents details of each design. The design point is for a typical lane configuration with 256 elements (corresponding to 32 vector registers with eight elements per lane), and five read ports and three write ports. This configuration of ports is the same as T0 and supports two VAUs, each with two read ports and one write port, and one VMU with a single read and write port. Alternatively, the VMU ports could

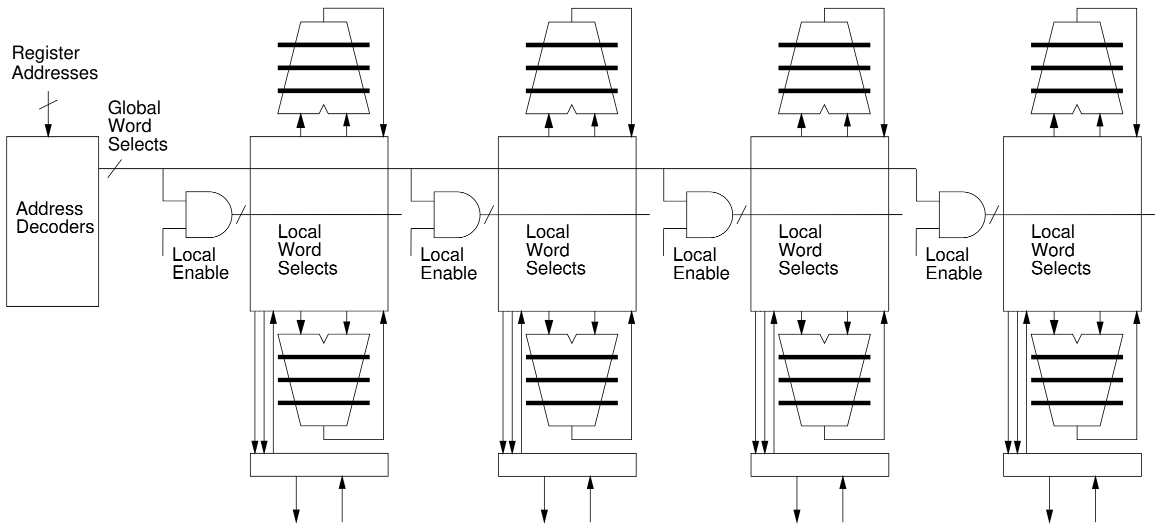

Figure 5.6: General layout of a vector register file. Storage is distributed across lanes, while register address decoders are shared by all lanes. The decoders broadcast global word select lines that are gated with local enable signals, one per lane, to form the per-lane local word select lines. In practice, each lane's storage is partitioned into banks, with each bank having a separate set of decoders. Also, this figure only shows one bank of decoders on the left side of the lanes, whereas the decoders will likely be placed on both sides of the lanes to reduce routing costs.

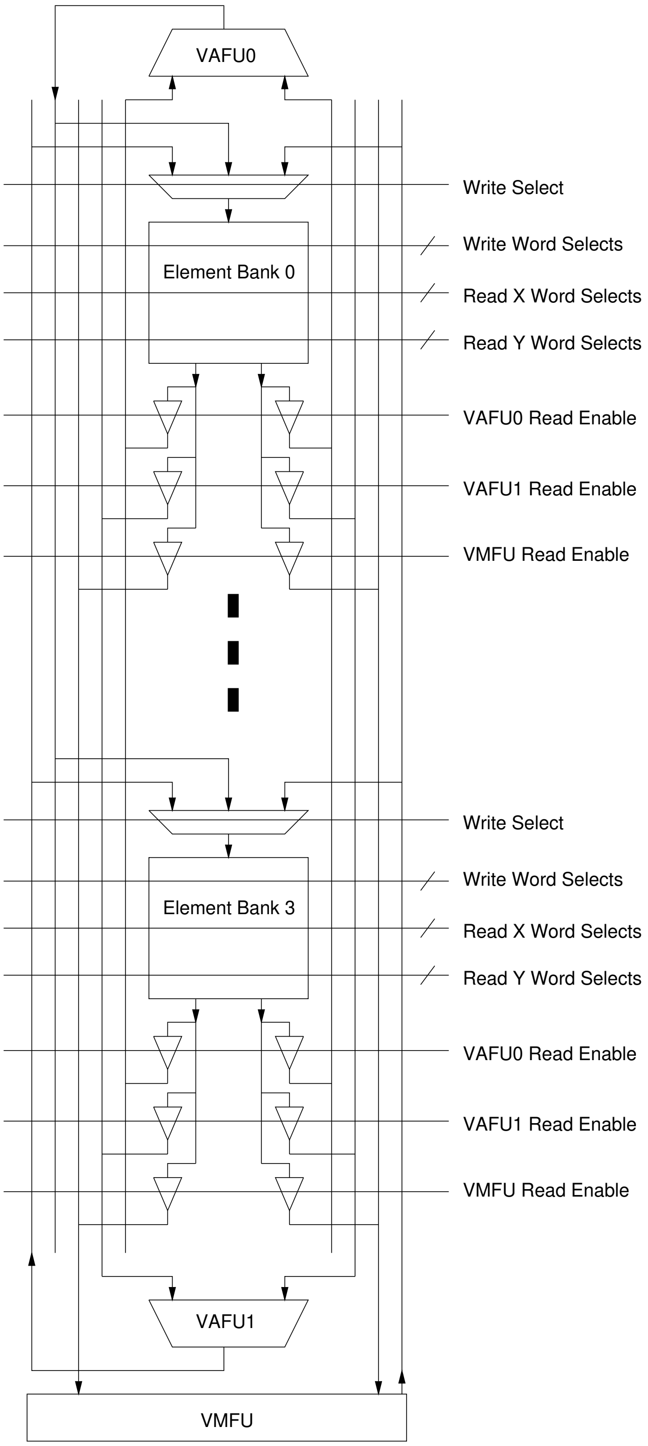

Figure 5.7: Layout of vector register file within one lane. In this example, four banks, each with two read ports and one write port, are used to support two VAUs and one VMU with a total requirement of 6 reads and 3 writes per cycle. With some extra port peripheral logic and more operand busses, the same number of element banks could support up to 8 reads and 4 writes per cycle.

Design	1	2	3	4	5
Banks	1	2	4	2	4
Elements/Bank	256	128	64	128	64
Cell Ports	5R+3W	3R+2W	2R+1W	2R+1W	1R+1W
Words per Port	1	1	1	2	2
Total Bandwidth	5R+3W	6R+4W	8R+4W	8R+4W	8R+8W
Cell Word Lines	5	3	2	2	1
Cell Bit Lines	6	4	2	2	2
Cell Area ($\lambda \times \lambda$)	57×72	41×64	46×28	46×28	40×28
Datapath pitch (λ)	72	64	56	56	56
Cells per Bitline	256	128	32	64	32
Cells per Wordline	64	64	128	128	128
Area per Bank ($\lambda \times \lambda$)	16419×4608	6316×4096	3532×3584	5332×3584	3443×3584
Storage (%)	88.9	83.1	41.7	52.7	35.9
Overhead (%)	11.1	16.9	58.3	47.3	64.1
Total Area ($M\lambda^2$)	75.7	51.7	50.6	40.0	51.2

Table 5.3: Comparison of five different schemes to implement the target 256-element vector register file with five read ports and three write ports.

support separate load and store VMUs. Because an indexed store needs both index and store data, indexed stores take two cycles per element. I consider the impact of providing two read ports for the VMU at the end of this section.

Figure 5.8 shows the layouts of the storage cells used in these designs.[3] The cells use two layers of metal, with the word lines in metal 1 and the bit lines in metal 2. The busses carrying read and write data from the VFUs to the banks will pass over the storage cells in the bitline direction using a third level of metal. In the designs below, I assume that 8 operand buses can be routed over the cell within the datapath pitch. The global word select lines cross over the main word lines in a fourth metal layer. The cells all have single-ended read ports and differential write ports.

I assume in all cases that the word and bit lines are time-multiplexed at twice the clock rate as in the T0 vector register file to provide both a read and a write access in the same cycle. Commercial microprocessors also use this technique to multiport caches, with examples including double cycling for the forthcoming Alpha 21264 [Gwe96a] and triple cycling for the IBM P2SC [Gwe96c]. This technique saves area but requires self-timing with dummy rows to provide the extra timing edges necessary to control read precharge and write drive. Although this adds extra design effort, the vector register file is a large regular array where a relatively small design effort can be rewarded with considerable area savings. Table 5.4 lists the area savings from time-multiplexing word and bit lines.

Design 1 has a single bank, capable of supporting 5 reads and 3 writes per cycle, and is similar to the vector register file design used in T0. This approach is the largest of those considered here, but does have the advantage of avoiding register file access conflicts between VFUs. This design requires one fewer metal

[3] These cells were designed by Brian Kingsbury and Bertrand Irissou.

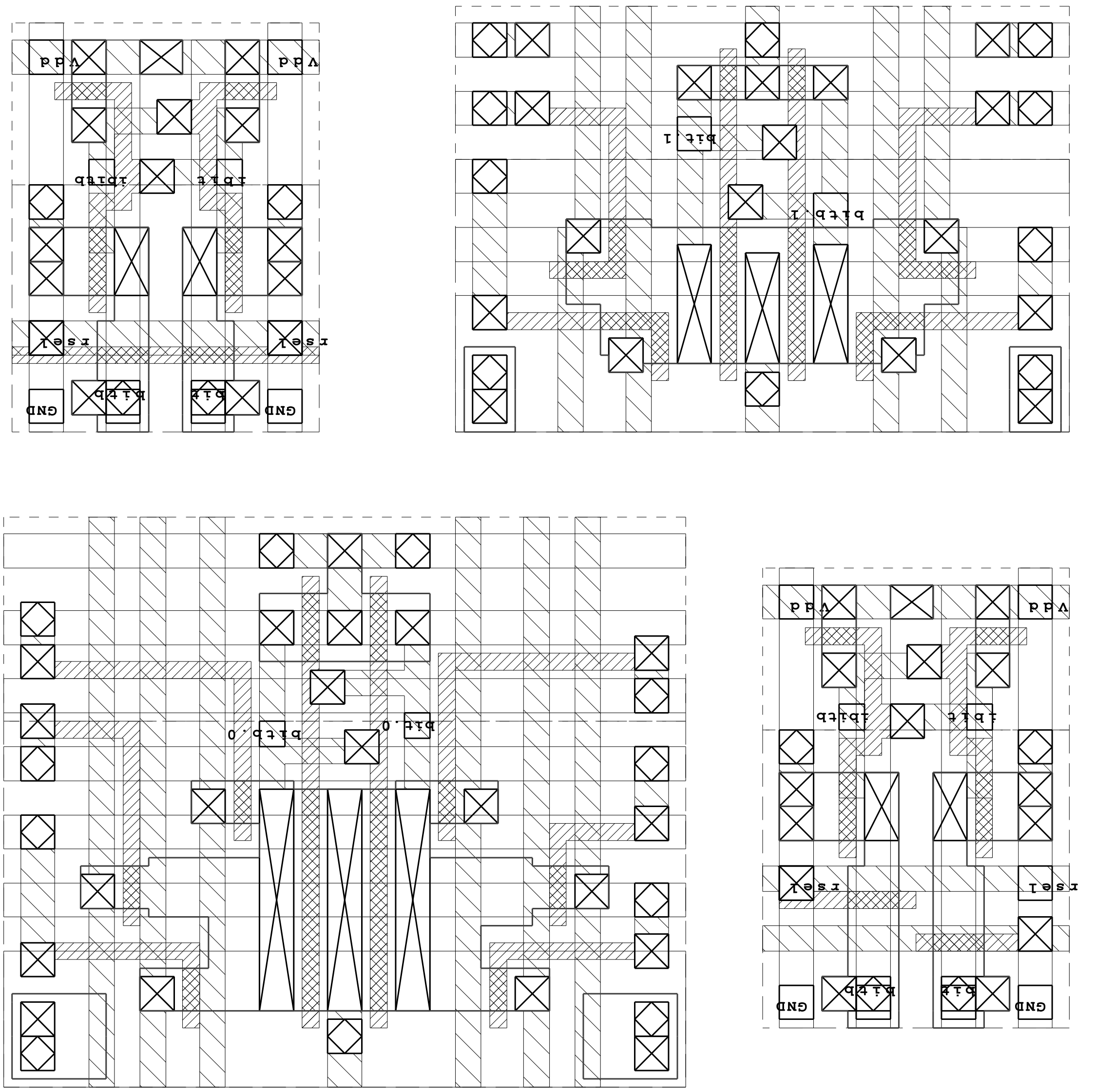

Figure 5.8: Layout for the multiported storage cells used in the vector register file designs. Moving top to bottom and left to right these are: the single-ported instruction cache cell, 2 read port and 1 write port scalar register cell (can also be used as 3 read port or 2 write port cell), 5 read port or 3 write port vector register cell, all from T0, together with the 2 read port or 1 write port cell discussed in the text. These were all designed using MOSIS scalable CMOS design rules with two levels of metal and have been drawn to the same scale.

Ports Required	Area Non-multiplexed ($\lambda \times \lambda$)	Area Multiplexed ($\lambda \times \lambda$)	Savings
2 R + 1 W	41×64	46×28	2.04
3 R + 2 W	57×78	41×64	1.69
5 R + 3 W	79.5×104.5	57×72	2.02

Table 5.4: Area savings by time-multiplexing read and write accesses on same word and bit lines.

layer than the others because all operand busses already run through the cells. The large number of cells connected to each bitline will lengthen the read cycle time.

Design 2 splits the storage into two banks. Each bank must have the same number of ports, and so must provide at least 3 reads and 2 writes per cycle to meet the design goal of 5 reads and 3 writes total. With double cycling, the storage cell requires 4 bit lines and 3 word lines. This scheme also requires crossbar circuitry to connect the two banks to the VFU operand busses. This adds extra overhead to each bank. The design dedicates one read port and one write port on each bank to the VMU. This reduces the amount of multiplexing circuitry required, and also simplifies control logic design because the VMU is usually the only VFU that experiences stalls during execution. The two VAUs alternate between the two banks, each of which has a further two reads and one write port available.

Design 3 has four banks, with each bank capable of supplying 2 reads and 1 write. With double cycling, the storage cell requires 2 word lines and 2 bit lines. Whereas the layout for the preceding cells would occupy most of a conventional datapath bit pitch, this smaller cell can be comfortably packed two per datapath bit slice. This packing does require an extra column multiplexer in the read datapath, and separate write enables for the two words in the write datapath.

Design 4 is a slight variant on Design 3. Because we have two storage cells per datapath bit slice, we can now fetch two elements per access to each bank. This halves the number of banks needed to match the VFU port bandwidth, but requires additional latches and multiplexing circuitry in the periphery of each bank. Because writes must wait for two elements to be produced before writing values into the storage cells, this double wide access port adds one cycle of latency when chaining through the vector register file. This latency can be avoided by providing bypass multiplexers between the VFU write and read busses as described above in Section 5.2.2. This additional bypass area is included in this design.

Design 5 is similar to Design 4, except that it uses a standard SRAM cell with only one word line and two bitlines. With double cycling, this allows one two-element read and one two-element write per bank per cycle. This design needs four banks to provide enough read bandwidth for all the VFUs. This design has the disadvantage of only providing a single read port per bank. For an element-partitioned scheme, this introduces an extra cycle of startup latency because the two zeroth elements live in the same bank. Banks with a single read port also suffer an extra area penalty because the single read port must connect to all VFU read busses. With two read ports per bank, each read port need only connect to one of the two read operand busses for a VFU. For example, Design 4 requires only 10 tristate read buffers, while Design 5 requires 20.

Scaled layout of the five designs is shown in Figure 5.9. The shaded regions represent overhead circuitry. The overhead on each bank includes the write port muxes, write latches for designs that write two elements per cycle, write drivers, dummy rows for the self-timing scheme, read sense amplifiers, read data latches for the schemes that return two elements per port, read data multiplexers for the cases that pack two bits per datapath bit slice, read port tristate drivers, and bypass muxes for the designs with double width access ports. Designs 4 and 5 have extra overhead for bypass circuitry to avoid the extra latency of writing two words at a time. The areas of all of these components were based on cells used in the T0 design.

These area estimates are approximate only. In a real design, layout of cells is likely to change to match the datapath pitch and number of metals available. For some cases, the drivers have not been optimally

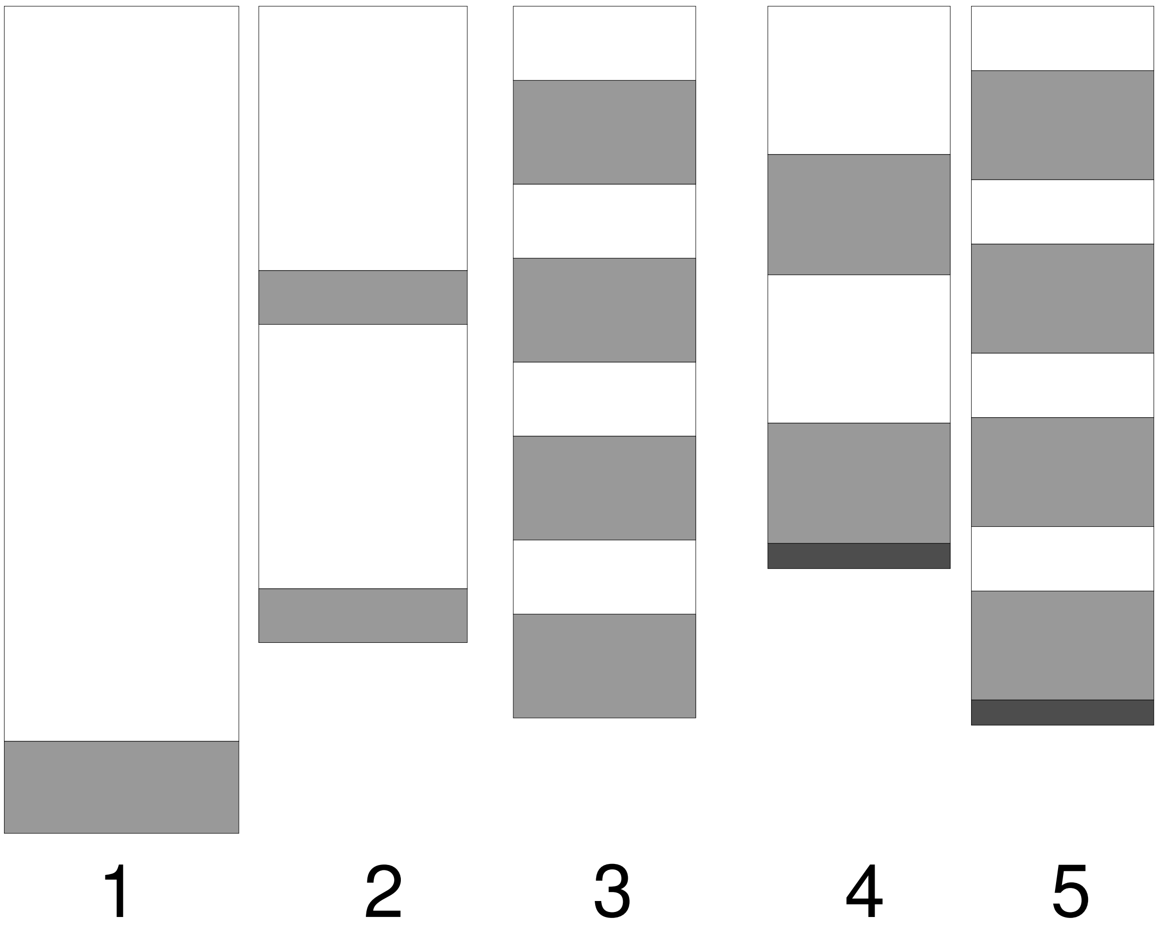

Figure 5.9: Scale comparison of the five vector register file designs, all of which have the same total storage capacity and number of VFU ports. The clear regions represent the storage cells while the shaded regions represent the overhead circuitry in each bank. For the purposes of this figure, all the overhead circuitry for a bank has been lumped together into a single rectangle; in a real design, the overhead circuitry would be split between the top and bottom of each bank. The darker shading at the bottom of Designs 4 and 5 is the area required for the additional bypass multiplexers which remove the extra latency caused by the two element write port width.

sized. While detailed tradeoffs will change with process technology, the overall trends should remain the same.

Among the five designs examined, the most compact was Design 4, with 2 banks using a cell capable of two reads and one write per cycle and with each port returning two words per access. This was almost half the area of the fully multiported design, and less than 80% of the area of the design that used the smallest cells. This suggests that the minimum vector register file area is obtained using storage cells with several ports arranged into a few banks, rather than a single bank with many ports or many banks with few ports.

This design also has only 64 elements per bitline which should still allow reasonable speed with a single-ended read sensing scheme. The available bank bandwidth is eight reads and four writes which provides some slack compared to the peak VFU bandwidth of five reads and three writes. This slack should reduce the occurrence of bank access conflicts.

The two level hierarchy within the vector register file design allows the number of ports and the amount of storage to be scaled independently. For example, a sixth read port can be added to enable indexed stores to run at full speed. Because there is sufficient bank bandwidth, this only requires adding additional read port buffers to each bank and would increase the area of Design 4 by only 3.5%. Adding an additional write port to allow two load VMUs, would only further increase the area by another 1%. This is assuming that the 10 operand buses can still fit over the cells in higher metal layers. Alternatively, doubling the amount of vector element storage would only increase the total size of the vector register file by roughly 50%.

Chapter 6

Vector Flag Processing

Unlike scalar processors, the virtual processors within a vector unit cannot branch and so other forms of conditional execution are required to vectorize code containing conditional statements. In this chapter, I begin by reviewing techniques that have been used to provide conditional execution in vector machines. I then propose a scheme for flag processing in vector machines that extends the traditional vector mask register functions to handle floating-point exceptions and to support vector speculative execution in software. This increases the range of vector processors by allowing loops with data-dependent exits ("while" loops) to be vectorized while preserving correct exception behavior. I then compare various techniques for improving the execution speed of masked vector instructions and conclude with a proposal for a flag processor implementation.

6.1 Forms of Vector Conditional Execution

Conditional execution in vector units has much in common with predicated execution in scalar processors. Virtual processors require conditional execution because they cannot branch, while scalar processors are employing increasingly sophisticated predicated execution schemes to reduce the number of branches.

Mahlke et. al. [MHM+95] compare various levels of support for predicated execution in scalar machines. They outline two levels of support for conditional execution, conditional move and fully predicated execution. Conditional moves are simpler to add to an existing instruction set. Only a few register-register instructions are required which can have the same number and types of operand as regular arithmetic instructions. Predicated execution requires that every instruction have an extra predicate source operand specifier.

The performance of conditional execution schemes can be improved if there is support for speculative execution, i.e., providing a version of an instruction that cannot raise exceptions. Speculative execution considerably reduces the overhead of emulating fully predicated execution with conditional moves, and also improves performance of fully predicated schemes by allowing operations to be scheduled before their predicates are known.

Among vector architectures, an example of partial support is the Cray architecture, which supports conditional execution with various forms of vector merge [Cra93], but does not provide speculative instructions. A vector merge is more powerful than a conditional move, in that it writes a destination vector register with values taken from one or other source operand according to the values in the mask vector. Vector merge is equivalent to the scalar *select* described by Mahlke et. al. [MHM+95]. Another example of partial support is the Torrent instruction set [AJ97] used for T0. Torrent uses vector conditional moves to implement conditional operations but does not support speculative instructions. Full support for masked operations is provided in several commercial vector mainframes, including the Fujitsu VPP series [UIT94], though these do not provide speculative instructions.

Conditional move or merge schemes in vector units share many of the same disadvantages as conditional move schemes in scalar processors [MHM+95]. They execute more instructions than fully masked schemes, especially when there is no support for speculative instructions. Operands to speculatively executed instructions must be modified to ensure no exceptions are generated if the predicate is false [MHM+95].

Figure 6.1 shows an example of this technique, using the vector conditional move operations in the Torrent instruction set [AJ97]. Here, masked vector loads are synthesized by converting unit-stride and strided operations into gathers where masked element addresses have been changed to a known safe locations, such as on the stack.

The routine begins with a load of an index vector which contains the word offset in bytes for each element in a unit-stride memory access. A vector of safe addresses is generated pointing to locations on the stack. A vector of safe addresses is used rather than a single address to avoid bank conflicts when multiple consecutive elements need to access a safe address. The load of the unit-stride index vector and the calculation of the safe stack addresses can be performed outside a stripmined loop. Note how effective address vectors must be calculated explicitly for all vector memory operations to allow the per element substitution of a safe effective address at masked element positions. Aside from the increase in the number of vector instructions executed, performance is impacted by the transformation of unit-stride vector memory operations into indexed vector memory operations. Because this requires much greater address bandwidth (see Section 8.1 on page 140) this is likely to cause a further slowdown over the masked version for which the microarchitecture is aware of the underlying unit-stride access pattern.

For the example shown in Figure 6.1, the compiler may be able to determine through program analysis or programmer annotations that the loop count would not exceed array bounds for the `B`, `D`, and `E` vectors. In this case the compiler can optimize the code as shown in Figure 6.2. Here unmasked unit-stride operations are performed, and unused element positions are simply ignored. The masked store is now performed as an unmasked unit-stride read, conditional move, unmasked unit-stride write sequence. Note that the indexed operation accessing the `C` vector still requires explicit address masking.

Even with this optimization, the conditional move version can generate significant additional memory traffic over that required by the masked version for sparse mask vectors. Another concern is the additional energy consumption. Operations are performed regardless of mask setting, and additional instructions are

```
for (i=0; i<n; i++)
{
    if (A[i] != 0)
        E[i] = D[i] + C[B[i]];
}

# With masked operations.
lw.v vv1, a1                 # Load A vector.
fnez.v vf0, vv1              # Generate mask.
lw.v,m vv2, a2, vf0          # Load B vector.
li t0, 2
sll.vs,m vv2, vv2, t0, vf0   # Multiply indices by 4 bytes.
lw.v,m vv4, a4, vf0          # Load D vector.
lwx.v,m vv3, a3, vv2, vf0    # Indexed load of C.
addu.vv,m vv5, vv3, vv4, vf0 # Do add.
sw.v,m vv5, a5, vf0          # Store E.

# With just conditional moves (Torrent code).
la t0, word_index_vec    # Load index vector containing
lw.v vv10, t0            # 0, 4, 8, ... into vv10.
addiu t0, sp, safev      # Address of safe vector on stack.
addu.vs vv9, vv10, t0    # Get vector of safe addresses.
lw.v vv1, a1             # Load A vector.
snez.v vv8, vv1          # Generate mask.
addu.vs vv12, vv10, a2   # Generate index vector for B.
cmveqz.vv vv12, vv8, vv9 # Write safe address where masked.
lwx.v vv2, zero, vv12    # Gather unmasked B values.
addu.vs vv14, vv10, a4   # Generate index vector for D.
cmveqz.vv vv14, vv8, vv9 # Write safe address where masked.
lwx.v vv4, zero, vv14    # Gather unmasked D values.
li t0, 2
sllv.vs vv2, vv2, t0     # Multiply indices by 4 bytes.
addu.vs vv2, vv2, a3     # Add in C base address.
cmveqz.vv vv2, vv8, vv9  # Write safe address where masked.
lwx.v vv3, zero, vv2     # Gather C values.
addu.vv vv5, vv3, vv4    # Do add.
addu.vs vv15, vv10, a5   # Generate index vector for E.
cmveqz.vv vv15, vv8, vv9 # Write safe address where masked.
swx.v vv5, zero, vv15    # Store values.
```

Figure 6.1: Example of synthesizing masked loads and stores with conditional moves. Stripmining code has been omitted. The unit-stride loads and stores are converted into scatter/gathers with a safe address on the stack substituted for addresses at masked element positions.

required to synthesize masked support. For these reasons, it is likely that a newly defined vector instruction set would include masked operations.

To handle nested conditional statements and to interleave execution of multiple conditional statements, it is desirable to provide several mask registers in a vector ISA. Because these mask registers can be used for purposes other than masking, I use the more general term *flag* registers. Flag registers can be set by vector arithmetic compare instructions. Masked instructions take a flag register as one source operand, and only complete operations at element positions where the corresponding flag bit is set.

There is some overhead to supporting full masking in a vector machine, mostly for masked vector memory instructions. To guard against Read-After-Write hazards on the flag registers used in masked vector memory operations, extra interlock circuitry is required early in the pipeline as described in Section 4.9 on page 76. Speculative address checking and load and store buffering can be employed to reduce the visibility of memory latency on masked vector memory instructions as described in Section 4.8 on page 74.

Another overhead with full masking is the instruction bits required to name the flag vector source. With a new ISA design, this should not be of great concern, because vector ISAs already significantly reduce instruction bandwidth requirements. But where a vector extension is added to an existing fixed-width instruction format, the instruction encoding may not allow an additional arbitrary flag register source to be specified on every instruction. If there are no bits available in the instruction encoding, all instructions can be masked by a single designated flag register. If a single instruction bit is available for mask encoding, it can be used to signal masked/not masked or to switch between two designated flag registers. Additional flag

```
# Torrent code.
# If known that all arrays can be read without exceptions.
la t0, word_index_vec    # Load index vector containing
lw.v vv10, t0            # 0, 4, 8, ... into vv10.
addiu t0, sp, safev      # Address of safe vector on stack.
addu.vs vv9, vv10, t0    # Get vector of safe addresses.
lw.v vv1, a1             # Load A vector.
snez.v vv8, vv1          # Generate mask.
lw.v vv2, a2             # Load B vector.
li t0, 2
sllv.vs vv2, vv2, t0     # Multiply indices by 4 bytes.
addu.vv vv2, vv2, a2     # Generate indices into C.
cmveqz.vv vv2, vv8, vv9  # Write safe address where masked.
lw.v vv4, a4             # Load D vector.
lwx.v vv3, zero, vv2     # Load C vector.
addu.vv vv15, vv3, vv4   # Do add.
lw.v vv5, a5             # Load E.
cmvnez.vv vv5, vv8, vv15 # Merge new values.
sw.v vv5, a5             # Write back E.
```

Figure 6.2: This version of conditional move code assumes that the compiler could determine that any element in the array could be read without causing exceptions.

move instructions can be used to move flag values into a designated flag register before use. These flag move operations can have single cycle latency and can support flexible chaining. Because vector flag functional units (VFFUs) are relatively cheap, more can be added to support the increased number of flag moves if necessary. The extra instruction bandwidth required for this explicit flag register renaming could be hidden with longer chimes.

6.2 Vector Flag Register Organization

Table 6.1 presents the flag register organization used for the examples in this chapter. This machine has 32 flag registers for each virtual processor. The first flag register, `vf0`, is used as the implicit source of mask values for masked instructions. The next seven flag registers, `vf1`–`vf7`, hold sticky exception signals. Five of the sticky exception flags report IEEE floating-point exceptions. Another two are integer sticky exception signals reporting integer overflow and fixed-point saturation. A further eight flag registers, `vf8`–`vf15`, are speculative sticky exception signals. These hold the exception results of speculatively executed vector operations; their use is described below in Section 6.7. The remaining 16 flag registers are available for general use.

6.3 Flag Combining Operations

Complex conditional statements can be handled by combining mask vectors using logical operations on flag registers. Some vector ISAs, including the Cray [Cra93], use scalar instructions to perform flag operations. The scalar unit reads vector mask vectors into scalar registers, performs regular scalar logical operations, then writes results back to the vector mask registers. The main advantage of this approach is that it requires no additional instructions in the ISA and no additional hardware in the implementation. But this approach has three significant disadvantages:

- Scalar reads of vector flag state expose memory latency, as described in Section 4.8 on page 74. Flag values are generated and used late in the vector pipeline. Memory latency is exposed when the scalar unit at the beginning of the pipeline must stall to read the value.

- The mask vector is produced and consumed incrementally by vector instructions, one element group at a time. The scalar unit reads the mask vector in larger chunks, e.g., 64 bits at a time. This wide read cannot be chained to the production of the flag bits. Similarly, if the bit vector is wider than a single scalar word, a dependent masked instruction cannot be issued until the scalar unit has written all of the words in the mask vector.

- If the number of elements in a vector register is greater than the number of bits in a scalar register, then each flag operation must be broken down into multiple scalar instructions. This makes it difficult to scale vector length for future implementations while retaining object-code compatibility.

The Torrent ISA [AJ97] avoids these problems by holding flag values in vector data registers and by using regular vector logical operations to combine flag values. The main disadvantages of this approach are that wide vector data registers are used for single bit predicate values and wide arithmetic datapaths are occupied with single bit predicate calculations.

Flag registers and associated VFFUs are relatively small structures that offload predicate calculations from more expensive vector data registers and VAUs. The VFFUs can operate in the late part of the vector pipeline where they can be chained flexibly to both producers and consumers of flag operands, further reducing inter-instruction latencies. For these reasons, the rest of this section assumes the ISA has multiple vector flag registers, with separate vector flag instructions executed by dedicated VFFUs.

6.4 Masked Flag Generation

Flags must be generated under mask to support nested conditionals. In particular, IEEE floating-point compares [IEE85] are used to generate flags but can also generate invalid operation exceptions which should not be reported at masked element positions.

Flag Register Specifier	Symbolic Name	Function
General Purpose Flags		
`vf16–vf31`		Undedicated
Speculative Exception Flags		
`vf15`	`vfsfpv`	Speculative Floating-point Invalid Operation
`vf14`	`vfsfpz`	Speculative Floating-point Division by Zero
`vf13`	`vfsfpo`	Speculative Floating-point Overflow
`vf12`	`vfsfpu`	Speculative Floating-point Underflow
`vf11`	`vfsfpx`	Speculative Floating-point Inexact
`vf10`	`vfssat`	Speculative Fixed-point Saturation
`vf9`	`vfsovf`	Speculative Integer Overflow
`vf8`	`vfslae`	Speculative Load Address Error
Non-Speculative Exception Flags		
`vf7`	`vffpv`	Floating-point Invalid Operation
`vf6`	`vffpz`	Floating-point Division by Zero
`vf5`	`vffpo`	Floating-point Overflow
`vf4`	`vffpu`	Floating-point Underflow
`vf3`	`vffpx`	Floating-point Inexact
`vf2`	`vfsat`	Fixed-point Saturation
`vf1`	`vfovf`	Integer Overflow
Mask Register		
`vf0`	`vfmask`	Implicit Mask Source

Table 6.1: Flag register assignment in example machine design. `vf0` is the implicit mask source. Seven flag registers (`vf1`–`vf7`) hold non-speculative exception flags. Eight flag registers (`vf8`–`vf15`) hold speculative exception flags. The remaining 16 flag registers are available for general use.

Mask	Compare or Logic Op Result	Unconditional Write (`u.*`)	AND-Write (`and.*`)	OR-Write (`or.*`)
0	0	0	-	-
0	1	0	-	-
1	0	0	0	-
1	1	1	-	1

Table 6.2: Table showing how flags are updated with result of masked compare or masked flag logical operations. Dashes ("-") indicate that the destination flag bit is unchanged.

As with scalar predicate generation [MHM+95], various forms of masked flag generation instruction are possible. Flag results can either be generated by compares on values in vector data registers, or through logical operations on flag registers. There are many possible ways in which the flag results can be combined with mask values to update the destination flag register.

Table 6.2 presents a proposal for masked flag updates which is used for examples in this chapter. This scheme is based on that proposed by Mahlke et. al. [MHM+95], but is somewhat simpler. There are three forms of flag update: masked unconditional flag updates write 0s where the mask is false, masked OR flag updates only write 1s when the mask is true, and masked AND flag updates only write 0s when the mask is true. These flag writes match the semantics of ANSI C logical-AND (`&&`) and logical-OR (`||`). Only one flag is written by an instruction and only one polarity of AND and OR is provided. Additional logical instructions can be used to generate multiple flag values based on the same comparison, and the comparison type can change the compare result polarity.

Figure 6.3 shows an example of using these flag instructions for a complex conditional loop body. This example assumes that only a single bit is used to encode masked/not masked with an implicit mask source of flag register 0 (`vf0`). If the machine had arbitrary mask sources, then the three explicit flag moves into `vf0` could be eliminated. But in a vector machine with one VMU, two VAUs, and two VFFUs, and with sufficiently long vectors, these extra flag operations would not be a bottleneck for this code. For short vectors, the additional cycle of latency caused by the extra flag operations could impact performance.

6.5 Flag Load and Store

A vector ISA must provided some mechanism to save and restore flag registers, if only to support context switching. One approach is to use regular masked instructions to move flag values between flag registers and vector registers, and then use regular vector loads and stores. This has the advantage of requiring no additional instructions or hardware. For context switching, vector arithmetic shift and logical operations can be used to pack multiple flag vectors into vector data register elements. Alternatively, if the flag registers are implemented as described below, then it is relatively inexpensive to provide instructions that move the entire vector flag set to or from vector data registers.

While these instruction speed context swap, they do not provide a mechanism for efficient generation

```
for (i=0; i<n; i++)
{
    if (A[i] > threshold || B[i] < max)
    {
        C[i] += 1;
        if (C[i] > limit)
            C[i] = limit;
        else
            B[i] *= alpha;
    }
}
```

```
ld.v vv1, a1                  # Load A.
u.fle.d.vs vf0, vv1, f0       # Unconditional flag A <= threshold.
u.not.f vf16, vf0             # Invert mask into vf16.
ld.v,m vv2, a2                # Load B under mask.
or.flt.d.vs,m vf16, vv2, f1   # Or in B condition under mask.
u.mov.f vf0, vf16             # Move combined mask to vf0.
ld.v,m vv3, a3                # Load C under mask.
addu.vs,m vv3, vv3, t2        # Increment C under mask.
u.fgt.vs,m vf17, vv3, t3      # Check for C > limit under mask.
u.mov.f vf0, vf17             # Set predicate for nested then.
mov.vs,m vv3, t3              # Set C = limit under mask.
u.mov.f vf0, vf16             # Restore outer predicate.
sd.v,m vv3, a3                # Store C under mask.
u.andnot.ff,m vf0, vf16, vf17 # Set predicate for nested else.
mul.d.vs,m vv2, vv2, f2       # B *= alpha under mask.
sd.v,m vv2, a2                # Store B under mask.
```

Figure 6.3: Example showing vectorization of complex conditional statement. This example assumes a machine with only a single bit to indicate masked or not masked with an implicit mask source of `vf0`. Stripmining and other scalar code has been omitted. Note that the load and compare of `B[i]` must be performed under mask because of the short-circuit evaluation semantics of the logical OR (||) operator in ANSI C. The load and compare of `B[i]` must not signal any exceptions if `A[i]` is greater than the threshold.

of a packed bit vector from a *single* flag register. This operation is important for codes that operate on condition vectors in memory. One approach is to move the flag register into vector data registers, then perform variable shifts and an OR reduction to pack the bits together.

Alternatively, dedicated flag load and store instructions can be provided for this purpose. These can be restricted to unit-stride loads and stores aligned at some bit boundary convenient for the memory subsystem implementation. The bit alignment should be chosen to be no greater than the number of elements in the guaranteed minimum vector length, otherwise it will not be possible to stripmine a loop that produces a contiguous packed bit vector in memory. For example, if a machine specifies the minimum vector register length is 16, the flag load-store instruction should allow the bit vector to be written at any 16-bit-aligned location in memory. For stores, the bit string can padded out with zeros if the vector length is not a multiple of the bit alignment.

6.6 Flag to Scalar Unit Communication

Apart from providing masking of elemental operations, flag registers can be used to report vector conditions to the scalar unit. There are two main operations required: population count and find first bit. Population count ("popcount") returns a count of the number of bits set in a flag register. Popcounts are often used in conjunction with the compress instruction, where the popcount gives the vector length after the compress. Find first bit gives the index of the first set bit in a flag register and is often used to identify the exit iteration for loops with data-dependent loop exits ("while" loops).

Both these instructions read a flag register source and write a scalar register destination. The masked version of both instructions treats masked elements as zero. Because they can build their result incrementally, they can chain to flag-producing instructions to reduce latency. In addition, find first bit can exit as soon as a set flag is encountered. This early-out can significantly speed execution of data-dependent exit loops by reducing the latency of the scalar read. The implementation of these instructions requires that flag values are transmitted from the individual lanes to a central position, likely located close to the scalar unit.

The Cray architecture implements these operations using scalar processor instructions for population count and leading zero count (the mask vector is transferred in bit-reversed order so that the leading zero count gives the same result as a find first bit on the mask register). This scheme has the same drawbacks with using the scalar unit for inter-flag operations as mentioned above in Section 6.3, but in addition prevents the early-out optimization on find first bit. The Torrent ISA provides a packed flag register that can be read by the scalar unit, but no other support for population count or find first bit.

To help handle floating-point flags, a further instruction can be provided that merges the vector floating-point exception flags into the scalar exception flag register. This simplifies retrieval of global flag state and frees the vector exception flags for other uses.

```
foo(size_t n,
    const double* A, double min,
    double *C, double scale)
{
    for (i=0; i<n; i++)
    {
        if (A[i] < min)
            break;
        C[i] *= scale;
    }
}

foo:
    setvlr t0, a0  # Set vector length.
    ld.v vv1, a1             # Get A vector.
    u.flt.d.vs vf0, vv1, f2  # Compare A.
    first.f t1, vf0          # Find any set bits.
    ctvu t1, vlr             # Trim vector length.
    ld.v vv3, a3             # Load C.
    mul.d.vs vv3, f4         # Multiply by scale.
    sd.v vv3, a3             # Store C.
    bne t1, t0, exit         # Break occured.
    sll t2, t0, 3            # Multiply by 8 bytes.
    addu a1, t2              # Bump A pointer.
    addu a3, t2              # Bump C pointer.
    subu a0, t0              # Subtract elements completed.
    bnez a0, foo             # More?
exit:
    j ra                     # Exit subroutine.
```

Figure 6.4: Example of speculative execution past data-dependent loop exit. The find first set flag instruction (`first.f`) will return the current vector length if there are no bits set in the source flag. The load and compare of `A` can generate spurious exceptions.

6.7 Speculative Vector Execution

In this section, I propose an extension of the flag processing model to support speculative execution of vector instructions. Speculative vector execution can be used to improve the scheduling of conditionally executed vector statements by moving instructions above mask dependencies in the same way speculative execution is used to increase ILP for a scalar processor by moving instructions above control dependencies.

Another use for speculative vector execution is to allow vectorization of loops with data-dependent loop exits. Consider the code in Figure 6.4. The vector assembly code speculatively loads and compares values which can be past the exit iteration, and so can generate exceptions that would not be present in a serial execution of the loop iterations.

Figure 6.5 illustrates how vector instructions execute multiple iterations of this loop in parallel, and shows which operations are executed speculatively. In this example, the machine has eight elements per vector register and the loop exits on the iteration `i=5`. The load and compares for iterations `i=6` and `i=7` were also executed and could potentially have caused spurious exceptions.

One limited approach to providing speculative memory loads is to provide a read-ahead buffer area after every memory segment. This read ahead would guarantee that reads to some region after a valid pointer would not cause address errors. The operating system can implement the read-ahead region by mapping in a zeroed page at the end of every allocated memory segment. To allow code that speculates loads to be portable across implementations with different vector lengths, the size of this region can be specified as some integer multiple of the maximum vector length. Although this software technique only provides speculation for unit-stride and small stride loads, it has zero hardware cost and no performance overhead, and can be used in combination with more sophisticated speculation schemes.

Another approach for handling both speculative arithmetic and memory exceptions is define a protocol between compiled code and the operating system that would identify speculative instructions. The operating system could determine that an exception was speculative and nullify the exception signal generated.[1] Speculative vector load exceptions can be nullified, even without precise traps, but reporting exceptions on a subsequent non-speculative use of the value read is more difficult. Speculative arithmetic exceptions might be impossible to undo if the speculative instruction has already modified sticky exception flag bits.

These problems can be solved by adding speculative vector instructions to the vector ISA. This proposal adds two types of speculative vector instructions: speculative vector loads and speculative vector arithmetic instructions. A speculative vector load instruction executes normally, but if there is an address error, a defined value (e.g., 0 or floating-point NaN) is written to the target vector data register element and a 1 is written to the corresponding element in a dedicated speculative load address error exception flag register. The speculative load address error flag is only set on illegal address errors. If a data page fault is encountered during a speculative vector load, it is handled normally with the faulting page brought into memory. This can result in extra paging traffic, but the effect is expected to be minor for most programs. A speculative vector arithmetic instruction executes normally but writes exception flags into speculative sticky exception flags mapped into different flag registers from regular non-speculative sticky flags.

Table 6.1 above shows how the speculative sticky flags are assigned in the example machine. These speculative vector instructions allow operations to be performed before it is known if they are required by

[1] This technique can also be used to tackle read-ahead errors off the end of the machine stack in operating systems that use these errors to automatically extend stack size.

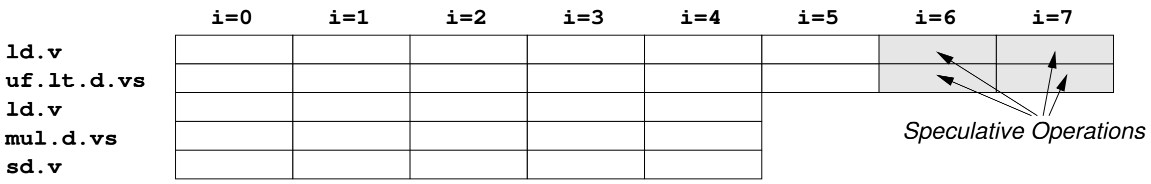

Figure 6.5: Executing the example code on a machine with VLMAX=8, and where the exit condition happens at `i=5`.

the original program control flow. Because no speculative vector stores are allowed, only vector registers can be speculatively modified. Software can allocate vector registers to prevent incorrect speculation from destroying essential machine state.

Once the correct program control flow is known, speculatively generated exceptions should be either reported or discarded as appropriate. In this proposal, speculative vector exceptions are committed with a `vcommit` instruction, which is maskable and controlled by vector length. This instruction executes within a VFFU and can be chained to the exception flag writes of a speculative instruction. The `vcommit` instruction performs two actions:

- Speculative arithmetic exception flags are OR-ed into the corresponding non-speculative arithmetic exception flags. A separate trap barrier instruction must be inserted after the `vcommit` if a precise arithmetic trap is desired when the new value of the non-speculative arithmetic exception flags contains set flags.

- If any speculative load exception bits are set, an address error trap is taken on the `vcommit`. To simplify normal execution, `vcommit` does not provide precise traps by default. Software can insert a separate trap barrier instruction if a precise trap is required for `vcommit` address errors. The speculative load address error flag register can be used to determine which VPs received an address error exception. Alternatively, a precise trap debugging mode can be enabled (see Section 4.6 on page 64) to give precise traps on `vcommit` address errors.

To support complex speculative or conditional flag code, this proposal adds a further set of instructions to move groups of eight flag registers at a time. These instructions read eight flag registers and write them into another group of eight flag registers, supporting the three forms of flag write described in Table 6.2. These instructions are named `u.mov8.f`, `or.mov8.f`, and `and.mov8.f`. To help initialize flags at the start of complex conditional sequences, a `u.clr8.f` instruction is provided which clears a set of eight flag registers. The operands for these flag group instructions must be eight consecutive flag registers starting at `vf0`, `vf8`, `vf16`, or `vf24`. These instructions can also be masked.

Figure 6.6 shows the previous example rewritten to use the vector commit instruction. A more complicated example with nested exit conditions is shown in Figures 6.7–6.8.

6.7.1 Speculative Overshoot

When vector speculative execution is used for data-dependent exit loops, software must balance vector length against "speculative overshoot" [Smi94]. The speculative overshoot of a loop is the number of excess iterations executed by speculative vector instructions. If the speculative overshoot is too large, performance may suffer due to the excess operations performed. If the vector length is too small, performance may suffer because the execution units are not kept busy. For short-chime vector microprocessors, speculative overshoot should be less problematic than in long-chime vector supercomputers, because even full length vector instructions complete in a few cycles.

```
foo:
    setvlr t0, a0             # Set vector length.
    u.clr8.f vf8              # Clear speculative flags.
    !ld.v vv1, a1             # Get A vector speculatively.
    sll t2, t0, 3             # Multiply by 8 bytes.
    addu a1, t2               # Bump A pointer.
    !u.flt.d.vs vf0, vv1, f2  # Compare A speculatively.
    first.f t1, vf0           # Find any set bits.

    addu t2, t1, 1            # Set vector length for
    setvlr zero, t2           # commit operation.
    vcommit

    ctvu t1, vlr              # Trim vector length.
    ld.v vv3, a3              # Load C.
    mul.d.vs vv3, f4          # Multiply by scale.
    sd.v vv3, a3              # Store C.
    bne t1, t0, exit          # Break?
    addu a3, t2               # Bump C pointer.
    subu a0, t0               # Subtract elements completed.
    bnez a0, foo              # Any more?

exit:
    j ra                      # Exit subroutine.
```

Figure 6.6: Using speculative instructions and vector commit to handle the data-dependent loop exit example. Speculative vector instructions are preceded with a `!`. Note that the vector commit instruction `vcommit` must have a vector length one greater than that of instructions following the exit `break`. The second `setvlr` instruction saturates the commit vector length to VLMAX if there were no successful compares.

```
for (i=0; i<n; i++)
{
    if (A[i] < max)
    {
        if (C[i] > B[i])
            break; /* Inner break. */
        else
            C[i] *= scale;
    }
    else
    {
        J[i] += 1;
        break; /* Outer break. */
    }
}
```

Figure 6.7: Source code of complex speculative execution example.

```
loop:
    setvlr t0, a0               # Set vector length.
    u.clr8.f vf8                # Clear speculative flags.
    # Find outer exit iteration.
    !ld.v vv1, a1               # Load A vector.
    !u.fge.d.vs vf0, vv1, f2    # Compare A.
    first.f t2, vf0             # Get outer exit iteration.
    ctvu t2, vlr                # Trim vector length.
    # Find inner exit iteration.
    !ld.v vv3, a3               # Load C vector.
    !ld.v vv2, a2               # Load B vector
    !u.fgt.d.vv vf17, vv3, vv2  # C > B?
    first.f t3, vf17            # Get inner exit iteration.
    ctvu t3, vlr                # Trim vector length.
    mul.d.vs vv3, vv3, f4       # Multiply C by scale.
    sd.v vv3, a3                # Store C.

    addu t5, t3, 1              # Commit length.
    setvlr zero, t5             # Set commit length.
    vcommit                     # Commit exceptions.

    bne t3, t2, exit            # Inner break before outer.
    bne t3, t0, outer           # Outer break.

    sll t1, t0, 3               # Multiply by 8 bytes.
    addu a1, t1                 # Bump A pointer.
    addu a2, t1                 # Bump B pointer.
    addu a3, t1                 # Bump C pointer.
    addu a4, t1                 # Bump J pointer.
    subu a0, t0                 # Subtract elements completed.
    bnez a0, loop               # More?
    b exit                      # No breaks encountered.
outer:
    # Must be outer break before inner break.
    ld.v,m vv4, a4              # Get J[i].
    li t5, 1
    addu.vv,m vv4, vv4, t5      # Increment J[i].
    sd.v,m vv4, a4              # Store[i].
exit:
```

Figure 6.8: Assembly code for complex speculative execution example.

Appropriate vector lengths for vector speculation can be determined by compiler analysis, profiling, or programmer annotation. Alternatively, a dynamic software scheme can adapt speculative vector lengths according to those observed at run-time. For example, software can use the previous exit iteration for a loop to predict the next exit iteration, perhaps rounding up to the next integral number of element groups.

6.8 Flag Priority Instructions

As the example above shows, handling data-dependent exit loops requires frequent scalar reads of the flag state using the find first flag instruction. These reads can be a performance bottleneck because they expose memory latency as described in Section 4.8 on page 74.

The scalar unit reads the find first flag value for two purposes. First, to determine that an exit condition has been encountered and that the stripmine loop should terminate, and second, to reset vector lengths so that subsequent vector instructions can be executed non-speculatively. Because the vector unit must wait for the vector length to be reset, vector execution stalls while the scalar unit reads the exit iteration index.

But note that the vector unit already possesses the information regarding the new vector length in the form of flag values. We can eliminate the round-trip to the scalar unit by adding flag priority instructions that create mask vectors of the required lengths. These mask vectors are then used to trim the effective vector length of subsequent instructions.

This proposal includes three forms of flag priority instruction, all controlled by vector length and mask, and all of which support the various flag update forms described above in Table 6.2 for compares and flag logical operations:

- Flag before first, `*.fbf.f`, reads a source flag register and writes 1s into a destination flag register in all element positions *before* the location of the first set flag in the source flag register, and 0s thereafter. If no flags are set in the source, then 1s are written to all destination element positions.

- Flag including first, `*.fif.f`, is almost identical to `*.fbf.f` except that a 1 is also written to the location corresponding to the first set flag in the source flag register.

- Flag only first, `*.fof.f`, writes a 1 to the location corresponding to the first set flag in the source flag register, and 0s at all other element positions.

Figure 6.9 shows the result of applying these instructions to the code from Figure 6.6. All vector instructions can now be issued without waiting for the scalar unit to receive the vector length from the `first.f` instruction. This value is now only used by the scalar unit to determine whether another stripmine loop is required. Figure 6.10 shows a sample execution of this loop, which can be compared with Figure 6.5.

As a further example, Figure 6.11 shows how the complex nested break statements in Figure 6.7 can be translated using flag priority instructions. Note how the `and.*` form of the flag priority instruction is used to nest the setting of the flag vectors marking the break iteration.

```
foo:
    setvlr t0, a0  # Set vector length.
    u.clr8.f vf8   # Clear speculative exception flags.
    !ld.v vv1, a1            # Get A vector speculatively.
    sll t2, t0, 3            # Multiply by 8 bytes.
    addu a1, t2              # Bump A pointer.
    !u.flt.d.vs vf16, vv1, f2 # Compare A speculatively.
    first.f t1, vf16         # Issue find of set bit.
    u.fif.f vf0, vf16        # Inclusive mask.
    vcommit,m                # Commit under mask.
    u.fbf.f vf0, vf16        # Exclusive mask.
    ld.v,m vv3, a3           # Load C.
    mul.d.vs,m vv3, f4       # Multiply by scale.
    sd.v,m vv3, a3           # Store C.
    bne t1, t0, exit         # Break encountered.

    addu a3, t2              # Bump C pointer.
    subu a0, t0              # Subtract elements completed.
    bnez a0, foo             # More?

exit:
    j ra                     # Exit subroutine.
```

Figure 6.9: Using flag priority instructions to eliminate round-trip latency to the scalar unit to set vector lengths for a loop with data-dependent loop exit. Note how all vector instructions following the `first.f` can be issued before the scalar unit receives the result of the `first.f` instruction.

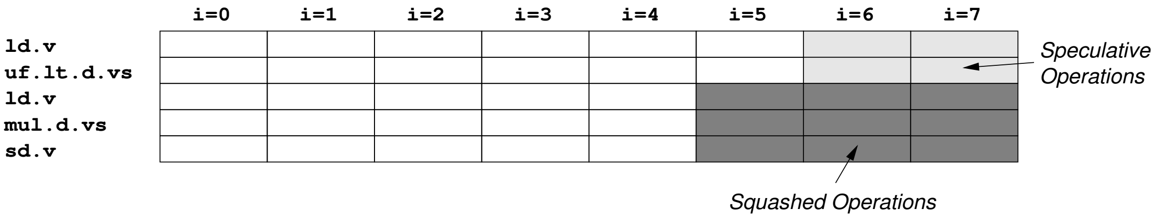

Figure 6.10: Executing the example code on a machine with VLMAX=8, and where the exit condition happens at `i=5`.

```
loop:
    setvlr t0, a0      # Set vector length.
    u.clr8.f vf8       # Clear speculative flags.
    # Find outer exit interation.
    !ld.v vv1, a1                # Load A vector.
    !u.fge.d.vs vf16, vv1, f2    # Compare A.
    first.f t2, vf16             # Get outer exit iteration.
    u.fbf.f vf0, vf16            # Set outer mask.
    # Find inner exit iteration.
    !ld.v,m vv3, a3              # Load C vector.
    !ld.v,m vv2, a2              # Load B vector.
    !u.fgt.d.vv,m vf17, vv3, vv2 # C > B?
    first.f,m t3, vf17           # Get inner exit iteration.
    and.fbf.f,m vf0, vf17        # Set inner mask under outer.
    mul.d.vs,m vv3, vv3, f4      # Multiply C by scale.
    sd.v,m vv3, a3               # Store C.

    u.fif.f vf0, vf16            # Outer commit mask.
    and.fif.f,m vf0, vf17        # Inner commit mask.
    vcommit,m                    # Commit exceptions.

    bne t3, t0, exit             # Inner break before outer.
    bne t2, t0, outer            # Outer break.

    sll t1, t0, 3                # Multiply by 8 bytes.
    addu a1, t1                  # Bump A pointer.
    addu a2, t1                  # Bump B pointer.
    addu a3, t1                  # Bump C pointer.
    addu a4, t1                  # Bump J pointer.
    subu a0, t0                  # Subtract elements completed.
    bnez a0, loop                # More?
outer:
    # Must be outer break before inner break.
    ctvu t2, vlr                 # Length available, so trim.
    u.fof.f vf0, vf16            # Single flag at exit iteration.
    ld.v,m vv4, a4               # Get J[i].
    li t5, 1
    addu.vv,m vv4, vv4, t5       # Increment J[i].
    sd.v,m vv4, a4               # Store J[i].
exit:
```

Figure 6.11: Using flag priority instructions with complex nested exit example.

6.9 Density-Time Implementations of Masked Vector Instructions

Masked vector instructions allow the vectorization of conditional code, but this form of conditional execution can be inefficient if the mask vector is sparse. A conventional *vector-length* [Smi94] implementation of masked vector execution simply nullifies writebacks and exception reporting at masked locations, and so takes a constant amount of time (VL cycles, where VL is the current vector length) to execute a masked vector instruction regardless of the number of set bits in the mask vector.

For sparse mask vectors, alternative conditional execution techniques using compress instructions or scatter/gather can be more efficient [HP96, Appendix B]. But these techniques require either fast inter-lane communication or extra address bandwidth, and also have additional programming overhead which limit their use to very sparse mask vectors.

An alternative approach is to improve the efficiency of masked vector execution through the use of a *density-time* implementations [Smi94]. A full density-time implementation executes only the unmasked operations in a vector instruction by skipping over zeros in the mask vector. In a single-lane implementation, execution time would be reduced from VL cycles to one cycle per set bit in the mask register.

Full density-time is difficult to implement as a design scales to multiple lanes. If each lane is to skip over zeros independently, it requires its own set of vector register file address decoders and its own set of interlock checks, rather than sharing decoders and interlock checks with other lanes (see Figure 5.6 on page 100). Because each lane can now have a different amount of work to do for each instruction, there is the potential for load imbalance across the lanes. In theory, lanes need only resynchronize for instructions that require inter-lane communication, but in practice control complexity will increase rapidly if lanes are not resynchronized after each instruction.

For multi-lane designs, simpler density-time implementations are possible which improve the efficiency of masked vector execution while avoiding the complexities of full density-time. I present two such schemes here: element-group skipping and masked vector length trimming.

- **Element-group skipping** checks the mask vector at the granularity of element groups (see Figure 2.4 on page 15). If all the mask bits in an element group are zero, the element-group is skipped. If one or more mask bits are set in an element group, the element group is dispatched to all lanes, but result writeback and exception reporting is disabled in inactive lanes. Element-group skipping simplifies control logic compared with full density-time and allows sharing of central vector register address generation and interlock control.

- **Vector length trimming** is an even simpler technique; hardware simply tracks the location of the last set bit in a mask register and reduces effective vector length when a masked instruction is dispatched. The main advantage of vector length trimming over element group skipping is that elements are accessed in the usual regular pattern, simplifying chaining.

Both element-group skipping and vector length trimming require that central logic keep track of the contents of the mask registers. The example machine has only one designated mask register (`vf0`) which simplifies implementations, because only writes to `vf0` need to be tracked.

In general, these simpler density-time schemes offer fewer savings than full density-time, but in two common cases their performance is equivalent. The first case is when all the mask bits are clear. This case is important for code that handles rare conditions using inline vector masked instructions. The second case is where the first N bits in a mask register are set. This case is important as it reflects the output of the flag-priority instructions described above.

Some classes of vector instruction may run much more slowly than regular vector instructions, for example, floating-point divides, or scatter/gather instructions in machines with limited address bandwidth. Because these instructions cannot support regular chaining in any case and because they would benefit most from skipping over masked operations, they may be candidates for special case density-time optimizations even if regular masked instructions use a simple vector-length implementation.

6.10 Flag Latency versus Energy Savings

Flag values are not strictly required until the end of the execution pipelines where they can control writeback of architectural state. By only using flags to control writeback, we minimize the latency between flag definition and use. While this increases performance, it incurs an energy cost because the operation is still performed even though its result will be discarded. If the flag value is known sufficiently early, the entire operation pipeline can be switched off to save energy. An energy conscious design might provide an early flag read port that would turn off pipeline stages that were known not be required, perhaps defaulting to speculatively executing operations when flag values are not known in time.

6.11 Vector Flag Implementation

In this section, I present a flag register file implementation which supports the proposed flag processing model. The vector flag register file has to support reads and writes from the VAUs, VMUs, and VFFUs. The target machine used here to describe the implementation has one VMU, two VAUs, and two VFFUs, with eight virtual processors (VPs) per lane. The logical flag organization is taken from Table 6.1.

Because each flag register is only a single bit, an implementation can spread all 32 bits of flag storage for one VP across the least significant half of a 64-bit datapath. All flags for one VP can be read together in a single cycle, and written together in a single cycle. This approach simplifies the implementation of complex flag instructions that require multiple reads and writes of different flag registers. Each VFU requires only a single read port and a single write port regardless of the number of flag registers read and written.

For the target machine, we can implement the flag register storage as two element-partitioned banks (see Section 5.2 on page 92). Each bank holds the 32 flag registers for 4 VPs. To satisfy the total VFU port requirements, each bank needs three read ports and three write ports. One read port and one write port on each bank is dedicated to the VFFUs. Another read port and write port on each bank is dedicated to the VAUs. The remaining ports are used by the VMU. The overall structure of one element bank is shown in Figure 6.12.

The flag read ports for the VAUs and the VMU multiplex one of the 32 flags onto a single bit mask bus fed into the control pipeline for the VFU. Floating-point arithmetic instructions update either the 5 speculative or the 5 non-speculative sticky exception flags using an OR-write, which only writes 1s to the cell by turning on the appropriate pull-down driver. Arithmetic compare instructions combine the compare result with the mask value and update a single flag result according to Table 6.2. Floating-point compares also OR-write the speculative or non-speculative invalid exception flags. OR-writes are also used to set fixed-point saturation, integer overflow, and speculative load address error exception flags.

The VFFUs implement a wide variety of operations, several of which require examining multiple flag registers at once. Basic logical operations among flag registers are handled with flag logic built into the VFFU read and write ports of each bank. This reduces latency and should allow a flag read-operate-write to happen within a single clock cycle. To implement flag logical operations, the VFFU hardware has three bit buses carrying two source flag values and the mask value across every bit position. Each write port has logic to combine these values for writeback. A single flag instruction executing within a VFFU will alternately access the two flag element banks, using the flag logic at each element bank to perform flag operations for VPs within the bank. A second flag instruction will execute within the second logical VFFU, accessing element banks interleaved with the first flag instruction.

As well as providing single bit flag operations, the VFFUs provide some operations on groups of eight flag bits at a time. The `vcommit` instruction reads the eight speculative exception flags and OR-writes them to the seven non-speculative flags. This requires an 8-bit bus running across the data path to carry the values from the speculative execution flags to the non-speculative execution flags. This bus is also used to implement the `*.mov8.f` flag group move instructions. The value of the speculative load address error flag

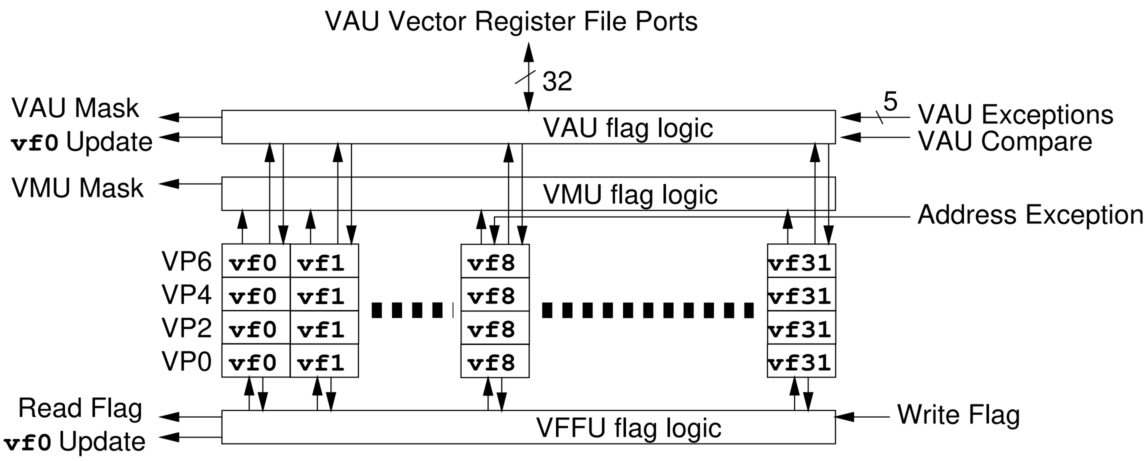

Figure 6.12: Overall structure of example flag register and flag processing hardware for one element bank. This element bank holds all the flag registers for VPs 0, 2, 4, 6. The other element bank would hold flag registers for VPs 1, 3, 5, 7.

is sent to the central instruction issue mechanism to signal load address error traps.

The trap barrier instruction is similar to `vcommit` and uses a VFFU to check for floating-point traps by scanning the five non-speculative floating-point exception flags looking for set bits corresponding to enabled traps.

Assuming the flag registers are arranged across the lane datapath, a path can be provided from the flag unit to the vector register file ports of a vector arithmetic unit. This allows the entire flag register set for a VP to be transferred to a single vector data element in one cycle to speed context switching. All flags can be cleared by transferring zero into the vector data register and then moving this into the flag registers.

There are several flag instructions that require inter-lane communication of flag values. Population count and find first flag instructions use the read port within a VFFU to send single flag bits to a central logic block located by the scalar unit, where bit strings are counted and results passed to the scalar register file. The flag priority instructions require inter-lane communication to determine if a set bit has been encountered. The read port of a VFFU is used to send flag bits to a central logic block which then redistributes the flag priority vector. This inter-lane communication will likely add a cycle or more of latency to the flag priority instructions.

Because at any time a VFFU can only be executing one of either a commit, trap barrier, popcount, find first flag, or flag priority instruction, the same read wire can be used to transmit either exception status or flag values to a central location in the control logic. The density-time optimizations discussed in Section 6.9 above require that all writes to `vf0` are tracked (only `vf0` can be a mask source in the example machine). The VAU and VFFU write ports send the result of any `vf0` write to logic in the issue mechanism that tracks the effective vector length in `vf0`. The VMU can only write the speculative address error flag (`vfslae` or `vf8`) and so VMU writes do not need to be tracked.

Chapter 7

Vector Arithmetic Units

This chapter discusses the design of arithmetic units for a vector microprocessor. Section 7.1 describes how vector arithmetic units can trade increased latency for increased throughput per unit area. Section 7.2 discusses the design of vector floating-point pipelines that can support the IEEE floating-point standard exceptions. Section 7.3 presents a proposal for increasing throughput on lower precision arithmetic in a vector unit by allowing virtual processor widths to vary. Section 7.4 compares vectors to the multimedia instructions which have recently become popular as extensions to commercial microprocessor instruction sets.

7.1 Vector Arithmetic Pipelines

Arithmetic pipelines for a vector processor can be made identical to those for a scalar processor. Indeed, a low-cost design will likely share expensive floating-point units between scalar and vector units.

A higher performance design can provide separate functional units for scalar and vector operations, enabling separate optimization. The scalar functional units can be optimized for low latency, while the vector functional units can be optimized for maximum throughput per unit area. Beginning with a scalar functional unit design, there are two complementary techniques which trade increased latency for improved throughput per unit area. The first saves area by removing latency reduction hardware, increasing the number of cycles required to produce a result at the same clock rate. The second increases clock rate by deepening the pipeline with extra latches.

Current CMOS scalar floating-point units require considerable resources to reduce latencies, to as low as 2 clock cycles in some cases [Yea96, Gwe95b]. If latency is allowed to increase while keeping throughput fixed, the floating-point unit can be reduced in area [UHMB94]. Some possible savings include: using simpler multiplier trees with less wiring; waiting until magnitudes are calculated before normalizing rather than predicting leading-zero positions in adders; and interleaving independent operations to reduce the impact of inter-iteration dependencies in iterative dividers.

Because back-to-back dependent operations are common in scalar workloads, scalar microprocessor clock cycle times are generally fixed by the time to propagate through an integer add and an operand bypassing network. In contrast, vector operations are known to be independent and so functional units can be pipelined more deeply to allow a higher clock rate. One way to reconcile these opposing clock cycle demands in a tightly coupled system is to use a multiple of the scalar clock frequency in the vector unit. For example, the Cray-2 [Sch87], Fujitsu VP-200 [MU84], Hitachi S810 [LMM85] all run the vector unit at twice the scalar clock rate, while the Hitachi S-3800 runs the vector unit at three times the scalar clock rate [KIS+94]. To simplify interfacing with the scalar unit and to allow a relaxed timing specification for the vector unit control logic, a vector functional unit running at twice the scalar clock rate can be treated as if there were twice as many lanes running at the normal clock rate.

Alternatively, the greater degree of pipelining in the vector unit can be used to maintain the same clock rate while the supply voltage is reduced. This approach reduces energy dissipation in the vector unit, while allowing the scalar unit to run with a higher supply voltage to maintain low latency.[1]

7.2 Vector IEEE Floating-Point Support

The IEEE floating-point standard [IEE85] has received wide acceptance in the computing industry. The standard defines a rich set of features to handle exceptional conditions in floating-point calculations. Scalar floating-point units often rely on a trap to software to deal with these exceptional conditions. It is more difficult to use software trap handlers to handle arithmetic exceptions in a vector machine. Although most features of the standard — including infinities and NaNs (Not-a-Number) — are straightforward and inexpensive to provide in hardware in a vector arithmetic unit, there are two features which require special attention: denormalized numbers and trap handling.

7.2.1 Denormalized Arithmetic

Support for arithmetic on subnormals is a requirement for compliance with the IEEE floating-point standard. Subnormals have been a controversial feature of the IEEE standard primarily because of the expense involved in their implementation.

Most existing microprocessors do not support subnormal operands in hardware, but take a machine trap to a handler which performs the required subnormal arithmetic in software. Some architectures, such as the Alpha [Sit92], cannot perform the required trap handling without the insertion of trap barrier instructions, which cause significant slowdowns in current implementations. Enabling subnormal support on the Alpha gives lower performance *even if no subnormals occur*.

Some systems abandon compliance with the standard and flush subnormal values to zero. These include some vector machines such as the Fujitsu VPP series [UIT94] and the IEEE floating-point option for

[1] Suggested by Dan Dobberpuhl at the IRAM retreat, June 1997. Dan also suggested an alternative technique of doubling the number of lanes and *halving* the clock rate to allow a lower vector supply voltage.

the Cray T90. This non-compliant handling of subnormals is also often provided as an optional mode for systems that otherwise support subnormals. This mode is used to reduce the variability of performance for real-time applications, to eliminate the run-time penalty of subnormal trap handling, or to emulate systems that always flush subnormals.

Handling subnormals with software traps is difficult in a vector machine. Without revocable register renaming hardware, handling subnormals with precise traps will entail a large loss in performance. Instead of using precise traps, subnormal values can be serviced with a special handler that is aware of the vector unit microarchitectural state. Unfortunately, subnormal operands may only be discovered when executing arithmetic operations at the end of a long decoupled pipeline. Freezing the vector unit requires stalling a large amount of microarchitectural state both in instruction and data queues and in datapaths across multiple lanes, and driving such a stall signal might adversely affect cycle time. Also, the microarchitecture would need to make all of this state visible to a software handler and providing the necessary access paths could increase area and cycle time.

Alternatively, subnormal values can be handled entirely in hardware, such as in the SuperSPARC processor [Sun92]. There are at least two schemes for handling subnormals in hardware. One scheme reformats floating-point data in registers, so that values are internally handled with a wider exponent range and only converted to and from subnormals on input and output. The disadvantage of this scheme are that either integer operations would incur extra area and latency to undo the reformatting, or loads and stores would need to be typed, requiring more instruction encoding and also complicating context save and restore. This scheme is not considered further here, primarily because vector registers can hold either integer or floating-point values at multiple different data widths (see Section 7.3 below).

The second scheme retains the standard binary representation in machine registers, and handles all subnormal operations within the floating-point arithmetic units. Handling subnormals in addition and subtraction requires little additional hardware. For subnormal source operands, it is sufficient to mask the introduction of the hidden bit into the significand during unpacking, as is already required to handle zeros. If a subnormal result is produced, the final normalizing shift must be modified to align result bits correctly. This requires additional logic between the normalization leading zero count and the shift control, which could add some delay.

Handling subnormal values for multiplication is more difficult. A multiply with normalized source and result values requires at most a single bit shift of the full-width significand product for result normalization, whereas subnormal values require larger shifts. One approach is to take extra cycles for subnormal values to handle these shifts (perhaps even using datapath hardware from the floating-point adder) while retaining reduced latency for normal values. This approach was used in SuperSPARC, where subnormal multiplies were handled in hardware but could take up to 5 extra cycles over normal multiplies [Sun92]. If used in a vector machine, this scheme would require that the whole vector unit stalls while subnormals are handled for an element group, and is therefore unlikely to scale to many lanes and high clock rates.

For a vector machine, perhaps the simplest approach to supporting subnormal multiplies is to provide fully pipelined support in the floating-point multiplier. The penalty is more area required for the extra shifters and sticky bit logic, and additional latency for all multiplies. The area penalty should become increasingly insignificant with advances in fabrication technologies, while the latency impact is ameliorated by the vector execution model. An advantage for real-time applications is that one further source of unpredictable execution time is eliminated.

Hardware for Fully-Pipelined Subnormal Multiplies

There are three cases to consider. The first case is a multiply of two normalized values that produces a subnormal result. This requires that the significand product be shifted right by as much as one significand width before rounding. Apart from the additional significand-width right shifter, this case requires more flexible generation of the sticky bit for rounding.

The second case is a multiply of two subnormal values which, depending on the rounding mode, produces either zero or the smallest subnormal. This case is no more difficult to handle than infinities or NaNs.

The third case is the most complicated and involves the multiplication of one subnormal and one normal value, which can result in either a normal or a subnormal result. This can be pipelined in two ways:

- If the subnormal source significand is shifted left before the significand multiply, the significand product will have no leading zeros. A normal result then requires no special handling. A subnormal result can be handled with a single rounded shift right as above when the product of two normals produces a subnormal.

- Alternatively, the unpacked source significands can be fed directly to the multiplier, which will result in leading zeros in the significand product. The full width multiplier product must be generated, instead of just a sticky bit for the lower half. A normal result will now require that the full width product be left shifted by up to one significand width to normalize the result value. The sticky bit logic must also be able to generate sticky bits for some variable number of bits in the lower half of the product. A subnormal result may require either a left or right shift depending on the source exponent values.

The first scheme has the advantages of a narrower left shifter and only requiring sticky bit logic for right shifts of the result, but it introduces additional latency before the multiplier input to allow for the exponent compare, operand multiplexing, and normalizing left shift.

The second scheme has the advantage of reduced latency because the shift amount can be determined by examining the subnormal significand in parallel with the multiply, but requires the full width significand product, a wider left shifter, and sticky bit logic to handle both left and right shifts at the output. Although the second scheme has an extra area penalty, the additional circuitry might be shared with additional functions. For example, the full width product can be used to provide both high and low words of the result for integer multiplies, while the normalizing left shifter is in the later stages of the pipeline which matches the requirements for floating-point addition; this should allow a multiply VAFU pipeline to implement both multiplies and adds at little additional cost.

7.2.2 User Floating-Point Trap Handlers

The IEEE standard recommends, but does not require, that it be possible to install trap handlers for any of the floating-point exceptions. The purpose of these trap handlers is to allow users to replace the default results produced by an exception before resuming execution.

The proposal presented in Chapter 6 signals exceptions by setting values in designated flag registers, and requires separate trap barriers be inserted to take a trap on asserted exception flags. For vector microprocessors with short chimes (Section 3.6), deep decoupled pipelines (Section 4.7), and some form of density-time mask optimization (Section 6.9), handling exceptional cases using inline masked vector instructions should be more efficient than inserting trap barriers and handling traps with separate user-level trap handlers.

For example, consider the straightforward formula to calculate the length of a two-dimensional vector:

$$l = \sqrt{x^2 + y^2}$$

If the machine does not have an extended precision format, the intermediate result $x^2 + y^2$ may overflow or underflow even though the final result is representable in the destination format. This formula can be evaluated without extended precision arithmetic by scaling with an appropriate constant a when overflow or underflow occurs:

$$l = \frac{1}{a}\sqrt{(ax)^2 + (ay)^2}$$

Figure 7.1 shows example masked code to calculate the value of this expression with appropriate scaling, using the speculative exception flags to signal intermediate overflows or underflows. A machine with some form of density-time implementation for masked instructions can quickly remove the exception-handling instructions from the instruction queue in the case that there were no exceptions signaled.

Floating-point trap handling can also be used for debugging purposes as described in Section 4.6 on page 64.

7.3 Variable Width Virtual Processors

While 64-bit floating-point arithmetic is a requirement for many supercomputing applications, many multimedia and human-machine interface processing tasks can be performed with 32-bit floating-point arithmetic, or even 16-bit or 8-bit fixed-point arithmetic. This reduction in data precision enables a corresponding increase in processing throughput if wide datapaths are subdivided to perform multiple lower precision operations in parallel. The narrower subwords can be handled elegantly with a vector instruction set. For operations with narrower elements, the vector unit can be considered to have more narrower virtual processors (VPs). For example, a vector machine with 32 64-bit VPs could also be treated as 64 32-bit VPs, or 128 16-bit VPs, or 256 8-bit VPs. The maximum vector length now becomes a function of VP width.

This technique was used on some of the earliest vector supercomputers [HT72] to provide higher processing speed for 32-bit versus 64-bit floating-point arithmetic, but has recently experienced a surge in

```
### Calculates z[i] = sqrt(x[i]*x[i] + y[i]*y[i]) using
### scaling if underflow or overflow occurs.
u.clr8.f vf8            # Clear all speculative flags
!mul.d.vv vv3, vv1, vv1 # x^2
!mul.d.vv vv4, vv2, vv2 # y^2
!add.d.vv vv5, vv3, vv4 # x^2+y^2
u.nor.f vf0, vfso, vfsu # Where neither underflow nor overflow,
or.mov8.f,m vf0, vf8    # preserve other exceptions
u.mov.f vf0, vfso     # Move speculative overflow flag into vf0
mov.vs,m vv10, f1     # Small power of 2
mov.vs,m vv20, f3     # Also get corresponding reciprocal (1/a)
u.mov.f vf0, vfsu     # Move speculative underflow flag into vf0
mov.vs,m vv10, f2     # Large power of 2
mov.vs,m vv20, f4     # Also get corresponding reciprocal (1/a)
u.or.ff,m vf0, vfso, vfsu   # Where either underflow or overflow
mul.d.vv,m vv11, vv1, vv10  # Scale x
mul.d.vv,m vv12, vv2, vv10  # Scale y
mul.d.vv,m vv13, vv11, vv11 # Square scaled x
mul.d.vv,m vv14, vv12, vv12 # Square scaled y
add.d.vv,m vv5, vv13, vv14  # Update sum register
sqrt.d.vv vv0, vv5          # Calculate square root
mul.d.vv,m vv0, vv0, vv20   # Multiply out by (1/a)
```

Figure 7.1: Example showing use of masked instructions to handle floating-point overflow and underflow while calculating the length of the hypotenuse. Where there is an intermediate overflow, the inputs are scaled by a tiny power of 2, and where there is an intermediate underflow, the inputs are scaled by a large power of 2.

popularity in the form of multimedia extensions to general purpose scalar architectures [Gwe96d, TONH96, Lee97].

7.3.1 Setting VP Width

VP width can be specified in every instruction or by writing a separate status word. Here, I assume that a separate control register, `vpw`, is written with the desired VP width. The VP width is not expected to change frequently and so VP width updates should not be a performance bottleneck. As described below, changes in VP width will also be treated as vector register file invalidates to prevent the storage layout of VPs becoming visible to software.

Where there are several data types in a single vectorized loop, software should set VP width to that of the widest data type handled in the loop. A VP can load and store all data sizes less than or equal to its width. Only one set of integer and fixed-point arithmetic instructions is required which use the full width of the VP datapath. A full set of single precision floating-point operations should be supported in the 64-bit VPs, to allow correct rounding and exception signaling when both single and double precision operations appear in the same loop.

While at first it would appear useful to switch dynamically between different data sizes so that each arithmetic operation runs at the maximum rate, this would require considerable inter-lane communication to match up data items from the same loop iteration. Software would also have to work around the differences in vector length for each precision.

7.3.2 Narrow VPs and Address Generators

Most vector instructions operate only on same-sized data and so subdividing lanes does not entail any inter-lane communication. One exception is address generation for strided and indexed memory accesses. The width of an address generator is fixed by the virtual address space of the machine, typically either 32 or 64 bits.

As discussed later in Section 8.1 address bandwidth is expensive, and it is likely that there will be at most one address generator per 32-bit or 64-bit lane. Narrower VPs must share address generators and so run strided or indexed memory accesses slower than vector arithmetic instructions. If address generators are located in the lanes then to reduce the amount of inter-lane wiring, narrower VPs should use the address generator located in the datapath slice of which they are part. To help maintain guarantees on element ordering, VPs should execute strided and indexed memory instructions in element order. If there is only one address generator in each 64-bit lane, then this leads to the data layout shown in Figure 7.2, where VPs are always striped across 64-bit datapaths regardless of VP width.

This layout somewhat complicates the memory skew network used for unit-stride loads, which would favor placing VP0 next to VP1 regardless of VP width. For this reason, implementations with one address generator per VMFU, or with centrally located address generators, would favor this alternative layout.

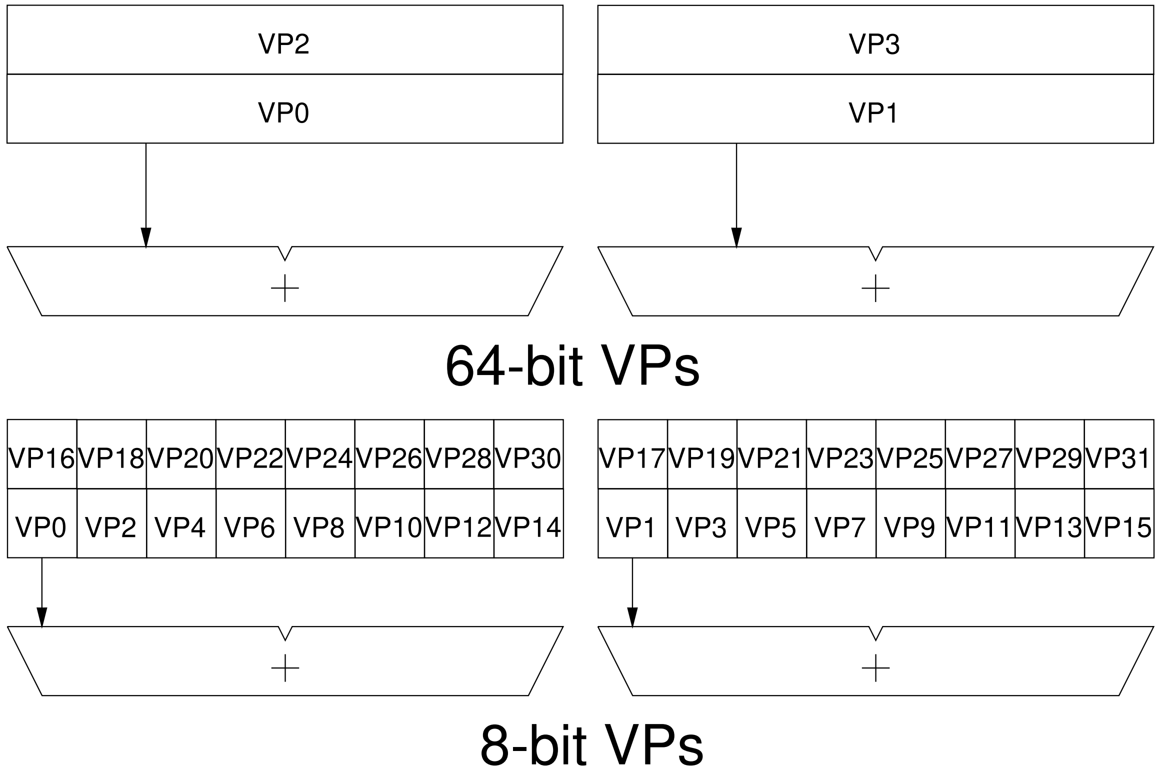

Figure 7.2: Mapping of VPs to two 64-bit datapaths when 64-bit address generators are located one per 64-bit lane. The upper figure shows how 64-bit VPs are mapped to the datapaths, while the lower figure shows 8-bit VPs. Both 64-bit and 8-bit VPs will run strided and indexed operations at the rate of 2 elements per cycle.

To avoid making the implementation-dependent VP storage layout visible to software, changes in VP width can be specified to cause a vector register invalidate operation leaving vector unit state undefined.

7.3.3 Flags and Variable-Width VPs

The flag implementation introduced in Section 6.11 on page 125 places 32 flag registers across 32 bits of a 64-bit lane. The flag storage can be doubled to span all 64 bits, allowing 32-bit wide VPs to have 32 flag registers. Narrow VPs do not support floating-point arithmetic, and so require fewer flag registers. The 16-bit wide VPs can be reduced to 16 flag registers, and the 8-bit wide VPs can be reduced to 8 flag registers. These fewer flag registers still allow support for speculative loads and integer arithmetic instructions, and also still allow all flag state to be transferred to a single vector data element in one cycle.

Flag state could also be specified to become undefined when VP width changes, avoiding the need to define how flag state maps across different VP widths.

7.4 Comparison with Multimedia Extensions

Variable-width VPs provide much the same functionality as the multimedia extensions currently popular in microprocessor architectures. These multimedia extensions add short vector operations to an existing scalar architecture by treating a 32-bit or 64-bit machine register as a vector of packed 8-bit, 16-bit, or 32-bit quantities. They have been shown to provide a significant boost for a few key multimedia applications without requiring much silicon area. But multimedia extensions as implemented so far [Gwe96d, TONH96, Lee97] have several disadvantages compared to a more comprehensive vector approach.

- **The vectors are *too* short!** Although a central theme of thesis has been how microprocessor technology favors vector machines with short chimes, the 64-bit vector length typical for these multimedia extensions is a very small amount of datapath work to represent with each instruction. This forces the use of a superscalar instruction issue mechanism to keep an array of functional units busy. Scaling to higher throughputs will require even greater instruction issue bandwidth. In contrast, the T0 implementation demonstrates how a single-issue vector machine can use longer vectors, specifying 1024 bits of datapath work, to saturate an array of parallel and pipelined functional units.

- **Not enough registers.** The single cycle execution of the multimedia instructions requires software loop unrolling to schedule around long latency instructions such as multiplies. This software loop unrolling *divides* the available register space among the unrolled loop iterations. In contrast, longer vector registers provide hardware loop unrolling which effectively *multiplies* the number of registers available.

- **Data alignment constraints**. Typically, the multimedia extensions cannot load or store data except as naturally-aligned 64-bit words. Unfortunately, it is common for applications to operate on vectors that are not aligned on these wide-word boundaries, and extra code is required to handle this misalignment in software. This restriction is imposed to avoid memory accesses that straddle cache line and page boundaries. Vector memory instruction already have to handle these cases and it is therefore straightforward to support subword vectors starting at any natural subword alignment.

- **Fixed-length vectors.** Not only are these vectors short, but they are also fixed in length. This complicates code when application vector lengths are not exact multiples of the machine vector length.

- **No non-unit stride and indexed memory accesses.** Current multimedia extensions provide no support for non-unit stride or indexed memory accesses.

These limitations seriously impact the performance of these multimedia extensions. Section 11.2 in Chapter 11 compares the performance of T0 against several commercial microprocessors with multimedia extensions on a range of image processing kernels. Even though T0 is a single issue machine, it achieves much higher execution rates than these other systems. It is plausible that future multimedia extensions will evolve into full vector units.

Chapter 8

Vector Memory System

The memory system is the most critical part of any vector machine design. The ability of a vector machine to tolerate latency and to scale to multiple parallel execution pipelines exposes memory bandwidth as the primary performance bottleneck for many vectorizable applications. Vector supercomputer memory systems are usually constructed from thousands of banks of SRAM connected with a hierarchy of crossbar switches [KIS+94]. Such systems provide multiple fast CPUs with high bandwidth access to a large shared memory at moderate latencies but are extremely expensive. The cost is justified in the largest vector supercomputers because absolute performance ("time-to-solution") is the primary design goal [KSF+94]. Vector microprocessor systems will also target markets where low cost and cost/performance are important, and so must be capable of working well with an inexpensive memory system.

This chapter explores the design of inexpensive vector microprocessor memory systems. Section 8.1 discusses the important distinction between address bandwidth and data bandwidth, and explains why address bandwidth is considerably more expensive to provide. I also summarize data from previous studies showing the distribution of unit-stride, strided, and indexed vector memory instructions in various vector supercomputer workloads. In Section 8.2, I take a closer look at vector memory reference streams and identify several higher level access patterns which can be exploited to improve the performance of low cost memory systems.

Vector machines can potentially have hundreds of parallel memory requests outstanding in various pipelines and buffers. Section 8.3 reviews the importance of loose vector memory consistency models which considerably reduce the complexity of implementing such highly parallel memory systems. These loose consistency models also simplify the implementation of various forms of vector cache. Section 8.4 then describes the design and implementation of vector memory units, which provide the connection between the memory system and the vector register file.

Section 8.5 reviews the memory technologies available to use in memory system designs. The single largest cost reduction in a vector memory system design is to replace SRAM with DRAM. DRAM is considerably more compact and less expensive than SRAM, allowing both larger and cheaper memories. New DRAM interfaces support data bandwidths comparable to SRAM, but with lower address bandwidths and with greater latencies. Section 8.6 describes techniques for driving DRAM memories in a vector memory system. A further way to reduce cost is to adopt a cached memory hierarchy; it is much less expensive to provide high bandwidth to a small SRAM cache than to a large DRAM main memory. Section 8.7 describes various forms of cache for a vector microprocessor.

Instead of the flat centralized memory model of the vector supercomputers, many scalar microprocessor-based systems have adopted a cheaper multiprocessor organization where memory modules are distributed amongst the CPUs. Each CPU has full bandwidth and low latency access to its local memory module, but reduced bandwidth and higher latency access to remote memory modules. Caches are often used in such systems to avoid these slower global accesses. Section 8.8 briefly outlines some of the issues surrounding the use of vector processors in such a distributed cache-coherent multiprocessor.

8.1 Address Bandwidth

The bandwidth figures usually quoted for memory systems are for *data bandwidth*, i.e., how many words of data can be transferred per unit time. Often, peak bandwidth numbers are quoted for streams of data held contiguously in memory.

Another important metric is *address bandwidth*, the number of non-contiguous memory requests that can be transferred per unit time. Address bandwidth is much more expensive to provide than data bandwidth for several reasons:

- **Address Busses/Pins.** Communicating the extra addressing information requires additional bandwidth above that required to move just the data.
- **Crossbar Circuitry.** Where there are multiple memory requests per cycle each with separate addresses, there needs to be address and data interconnect to allow each memory requester to connect to each portion of memory. If the memory banks are not fully multiported, this interconnect creates the possibility of conflicts which require additional circuitry to detect and resolve.
- **Cache Coherency Traffic.** In cached systems, each address request may require a separate access to cache tag and directory information, and may generate cache coherence traffic between nodes in a multiprocessor system.
- **TLB Bandwidth.** Each extra address generated per cycle requires another port into the TLB for virtual address translation.
- **Limited DRAM Address Bandwidth.** DRAMs naturally have less address bandwidth than data bandwidth. On every row access, DRAMs fetch kilobits of data into sense amplifier latches and can return data from this block much more rapidly than if intervening row accesses are required.

Because of this expense, designers of low-cost systems attempt to reduce address bandwidth demands by adding caches. Memory hierarchies usually transfer naturally-aligned blocks of data containing some power-of-2 number of bytes so that each data transaction moving multiple data words requires only a single address transaction. A typical microprocessor system memory hierarchy is shown in Table 8.1, which presents address and data bandwidth for the memory hierarchy of the Digital AlphaServer 8400 [FFG+95]. While both peak address and data bandwidth drop as we descend the memory hierarchy[1], address bandwidth drops off much more steeply than data bandwidth. The L1 cache can support one address per data word, while the lower levels of the memory hierarchy only support one address every eight words. The L3 cache on each processor can only support a new address every 21 clock cycles, and the system bus can only support a new address every 12 clock cycles across all 12 processors.

In contrast, vector supercomputers typically allow a unique address for every word transferred throughout their memory system. The 240 MHz Cray C916 vector supercomputer supports up to 6 addresses per cycle per processor[2], and up to 96 addresses per cycle across all 16 processors [Cra93]. This represents over 900 times greater main memory address bandwidth than the AlphaServer whereas main memory data bandwidth is only 115 times greater.[3] This huge address bandwidth and associated crossbar circuitry adds a great deal to the expense of a vector supercomputer's memory system.

Table 8.2 presents data from studies of a variety on scientific codes running on several different platforms, and shows breakdowns of vector memory operations into unit-stride, strided, and indexed access types. Individual program profiles vary widely. Several are completely or almost completely unit-stride. Some perform the majority of their accesses using strides greater than one. Many programs make no indexed accesses; where they occur, indexed accesses are usually a small fraction of the total. One caveat is that some of these figures are taken from source code and compilers that assumed implementations had roughly equal unit-stride and non-unit stride bandwidth. For example, LU decomposition can be performed almost entirely with unit-stride operations, but perhaps this would not perform as well on the Cray C90 hardware as the version reported which has 82% strided accesses. It is also interesting to compare the results for the two PERFECT benchmarks, `arc2d` and `bdna`, traced on both the Alliant FX/8 and the Convex C3. The distribution of unit-stride, strided, and indexed accesses is markedly different between the two machines,

[1] The DRAM bandwidths are actually *greater* than those of the L3 cache. The system bus can transfer one 64-byte line every 12 clock cycles whereas the L3 cache takes 21 cycles to transfer a 64-byte line, but the system DRAM bandwidth is shared by up to 12 processors.

[2] Not including I/O accesses or instruction fetch.

[3] The newer 460 MHz Cray T932 supports over 2,600 times the main memory address bandwidth of the newer AlphaServers with 100 MHz system buses.

Level	Size (KB)	Data Bandwidth (words per cycle)	Address Bandwidth (words per cycle)	Data:Address
L1	8	2	2	1
L2	96	2	0.5	4
L3	4096	0.38	0.048	8
DRAM	Large	0.67	0.083	8

Table 8.1: Peak address and data bandwidths for the memory hierarchy within a Digital AlphaServer 8400 system with 300 MHz Alpha 21164 processors. The final column gives the data:address bandwidth ratios.

Benchmark Name	Unit-Stride (%)	Other Strides (%)	Indexed (%)
NAS Parallel Benchmarks on Cray C90 [HS93]			
CG	73.7	0.0	26.3
SP	74.0	26.0	0.0
LU	18.0	82.0	0.0
MG	99.9	0.1	0.0
FT	91.0	9.0	0.0
IS	73.4	0.0	26.6
BT	78.0	22.0	0.0
Hand-Optimized PERFECT Traces on Cray Y-MP [Vaj91]			
Highly Vectorized Group	96.9		3.1
Moderately Vectorized Group	85.3		14.7
PERFECT code on Alliant FX/8 [FP91]			
`arc2d`	54.4	44.4	1.2
`bdna`	7.1	84.5	8.4
`adm`	22.4	18.3	59.3
`dyfesm`	37.4	22.4	40.2
PERFECT Traces on Convex C3 [Esp97a]			
`arc2d`	80.0	11.0	9.0
`bdna`	78.0	17.0	5.0
`flo52`	72.0	28.0	0.0
`trfd`	68.0	32.0	0.0
SPECfp92 Traces on Convex C3 [Esp97a]			
`tomcatv`	100.0	0.0	0.0
`swm256`	100.0	0.0	0.0
`hydro2d`	99.0	1.0	0.0
`su2cor`	76.0	12.0	12.0
`nasa7`	25.0	75.0	0.0
Ardent Workload on Ardent Titan [GS92]			
`arc3d`	98.6	1.4	
`flo82`	74.5	25.5	
`bmk1`	50.4	49.6	
`bmk11a`	100.0	0.0	N/A
`lapack`	100.0	0.0	
`simple`	55.4	44.6	
`wake`	99.5	0.5	

Table 8.2: Summary of published analyses of vector memory accesses categorized into unit-stride, strided, and indexed. The Cray Y-MP results for the PERFECT traces did not separate unit stride from non-unit stride [Vaj91]. The Ardent workloads do not include scatter/gather accesses as those are treated as scalar references [GS92].

showing how differences in compiler can affect these statistics.

Choosing the balance between address bandwidth and unit-stride data bandwidth is an important design decision in a vector microprocessor as it will have a large impact on final system cost and performance. Vector microprocessors will likely provide less main memory address bandwidth than vector supercomputers.

Unit-stride accesses dominate in vector supercomputer workloads, even though these systems have full address bandwidth. Apart from vector memory instructions measured in these studies there are other demands on the memory system, including instruction and scalar data cache refills and I/O transfers. These are all usually performed as unit-stride bursts.

Although strided and indexed access patterns are a small fraction of the total memory workload, they could dominate execution time if they execute too slowly. One of the key challenges in designing a cost-effective vector microprocessor system is to retain the advantages of vector execution while reducing address bandwidth demands on lower levels of the memory hierarchy. The goal is to convert strided and indexed memory access patterns into unit-stride accesses whenever possible. There are several complementary approaches:

- **Rearrange code to perform unit-stride accesses.** This purely software technique is the cheapest option for hardware. Compilers or programmers can interchange loops to yield unit-stride accesses in the inner loops. This optimization has been commonly performed for vector machines. Even for machines with equal unit-stride and strided bandwidths this technique removes potential intra-vector conflicts and minimizes the impact of inter-vector conflicts. Unfortunately, in some cases, even when loops can be interchanged to give unit-stride accesses in the innermost loop, this might then introduce dependencies that prevent vectorization.

- **Move data within vector registers.** Some forms of strided and indexed memory operations can be replaced with operations that manipulate the contents of vector registers. For example, strided accesses used to implement the butterfly operations within FFT algorithms can be replaced with vector register permute operations. Section 10.3 describes these in more detail.

- **Caches.** Vector machines can use caches to reduce both address and data bandwidth demands. Section 8.7 discusses the addition of caches to vector machines, while Chapter 9 introduces two new forms of cache appropriate for vector machines. One of these is the rake cache, which can convert many occurrences of strided and indexed accesses into unit-stride bursts.

8.2 High-Level Vector Memory Access Patterns

This section identifies higher-level memory access patterns which occur frequently in vectorized programs to help motivate architectural design decisions and to help describe the behavior of applications in Chapter 11. These patterns emerge from the sequences of vector memory instructions used to access one array operand within a vectorized loop nest. Each array operand in a vectorized loop nest could potentially have a different access pattern. The characteristics of interest for each access pattern include spatial locality, temporal locality, and predictability. Spatial locality can reduce address bandwidth demands, temporal locality can reduce both address and data bandwidth demands, while predictability can be used by prefetch schemes to reduce the effects of memory latency.

8.2.1 1D Strided Vectors

Perhaps the simplest common high-level reference pattern is a 1D (one-dimensional) strided vector where every element is touched once before any further accesses are made to the vector. If accessed unit-stride, this operand has considerable spatial locality and requires little address bandwidth. If accessed with a large non-unit stride, there is no spatial locality and a different address is required for every word. A 1D vector can only exhibit temporal locality between separate accesses not within one access, and only then if the whole 1D vector fits in cache. A 1D vector has a highly predictable access pattern, and such access patterns have been the target of prior work with stream buffers for scalar machines [PK94].

8.2.2 Permutations

A permutation access uses indexed vector memory instructions to access every item within an array once. This access pattern has no temporal locality but considerable spatial locality, provided the cache is large enough to hold the entire array. The access pattern is generally difficult to predict.

8.2.3 Lookup Tables

Lookup table accesses use indexed vector memory reads to access elements in a table. These are very common for function approximation. Lookup table accesses can exhibit considerable temporal locality if certain entries are very commonly accessed, even if the whole table cannot fit into the cache. The access pattern is generally difficult to predict.

8.2.4 Neighbor Accesses

The distinguishing feature of a neighbor access pattern is that the same memory operand is accessed multiple times by neighboring virtual processors (VPs). Neighbor accesses are especially common in filtering, convolution, and relaxation codes such as that shown in Figure 8.1. In this example, element `A[3]` would be read by VPs 2, 3, and 4.

The code in Figure 8.1 loads the same data multiple times into the vector register file. Some vector architectures have added instructions that allow vector arithmetic instruction to begin reading elements at arbitrary offsets within vector registers [Con94, WAC+92, DHM+88]. This allows data in the registers of one VP to be reused by another. While this approach would eliminate two out of three of the load instructions, it does not scale well to more lanes. Each pipeline in each VAU would require a crossbar between its inputs and the vector register file slices in all lanes, dramatically increasing the amount of inter-lane wiring, and potentially increasing the latency of all arithmetic instructions.

Rather than provide extra inter-lane communication paths for VAUs, we can instead simply issue multiple load instructions, reusing the existing alignment crossbar in the VMU. A cache can filter out the repeated loads from the underlying memory system. Because all three of these accesses take place in the same iteration of the stripmine loop, even a small cache can capture the resulting temporal locality.

```
/*
    for (i=1; i<N-1; i++)
        B[i] = 0.25 * (A[i-1] + 2*A[i] + A[i+1]);
*/

loop:
    setvlr t0, a0      # Set vector length.
    addu t2, a1, 8     # Start at A[1].
    ld.v vv2, t2       # Load A[i] first.
    add.d.vv vv2, vv2  # 2*A[i]
    ld.v vv1, a1       # Load A[i-1].
    add.d.vv vv2, vv1  # 2*A[i] + A[i-1].
    addu t2, 8
    ld.v vv3, t2       # A[i+1]
    add.d.vv vv2, vv3  # += A[i+1].
    mul.d.vs vv2, f0   # *= 0.25.
    sd.v vv2, a2       # Store B[i].
    sll t1, t0, 3      # Multiply by 8 bytes.
    addu a1, t1        # Bump to next strip.
    subu a0, t0        # Subtract elements done.
    bnez a0, loop      # More?
```

Figure 8.1: Example of a neighbor access pattern in a filtering code.

```
for (j=0; j < rake_depth; j++)
        for (i=0; i < rake_width; i++)
                ...A[i][j]...
```

Figure 8.2: C code showing strided rake access pattern.

8.2.5 Rakes

A *rake* is a particular form of two-dimensional memory access pattern which is common in optimized vector code. The access pattern can be described with two nested loops as in the fragment of C code shown in Figure 8.2. The inner loop moves down a column of matrix A. The outer loop moves over the columns one at a time. Because C stores matrices in row-major order, and assuming that dependencies prohibit interchanging the loop order, this loop nest can only be vectorized using strided memory accesses.

This access pattern does not only occur in two dimensional matrix operations; a rake is so named because these are the access patterns obtained when *loop raking* [ZB91] is used to assign contiguous elements of a one-dimensional input vector to the same virtual processor. In this case, software usually arranges the rake width to match the number of virtual processors.

In general, a rake has the form of multiple interleaved unit-stride streams. When vectorizing a rake access, each unit-stride stream is assigned to one virtual processor. A *strided* rake, as in Figure 8.2, is

```
for (j=0; j < rake_depth; j++)
        for (i=0; i < rake_width; i++)
                ...A[B[i]][j]...
```

Figure 8.3: C code showing indexed rake access pattern.

```
for (j=0; j<N; j+=NVP)
    for (i=0; i<M; i++)
        for (jj=j; jj < min(j+NVP, N); jj++)
            ...A[i][jj]...
```

Figure 8.4: Example of multicolumn access pattern.

where the starts of the unit-stride streams are separated by a constant stride, and so a strided vector memory operation can be used to access the next element in each stream. An *indexed* rake is where the starts of the unit-stride streams are irregularly spaced, and where indexed vector memory operations are required to access the next element in each unit-stride stream. Figure 8.3 gives a fragment of C code showing an indexed rake. The extent of a rake can be characterized with two parameters, *rake width* and *rake depth*. The rake width is the number of unit-stride streams accessed simultaneously, and software often arranges for this to be equal to the number of virtual processors. The rake depth is the number of elements taken from each stream. Strided rakes also have an associated *rake stride* whereas indexed rakes have an associated *rake index vector*.

There are many vectorizable applications where strided rakes appear, including dense matrix multiplication (Sections 11.1.2–11.1.3), data compression (Section 11.6.3), Kohonen networks (Section 11.5.2), and cryptographic applications (Section 11.4). Indexed rakes are not quite as common, but appear in vectorized garbage collection (Section 11.6.5) for example.

Because rakes are a set of unit-stride streams, they possess considerable spatial locality. The *rake cache* introduced in Chapter 9 exploits this spatial locality to considerably reduce the address bandwidth required to support rakes. Rakes are also highly predictable for larger rake depths.

8.2.6 Multi-Column Accesses

Multi-column accesses are another form of two-dimensional access pattern common in matrix arithmetic codes. A multi-column access is shown in Figure 8.4. Here multiple VPs traverse down neighboring columns of a matrix. The pattern consists of a sequence of unit-stride accesses separated by a constant stride. This is the transpose of a strided rake which consists of a sequence of strided accesses separated by a unit stride.

Because the multicolumn access pattern is composed of unit-stride accesses, it has substantial spatial locality. The access pattern is also highly predictable because each unit-stride access is separated from the previous by a constant stride amount.

The Torrent ISA [AJ97] allows an optional address register post-increment to be specified along with unit-stride vector memory accesses. This was originally added to reduce instruction bandwidth requirements, but this post-increment can also be used to give a prefetch hint for the start address of the next unit-stride access in the multicolumn pattern. This prefetch hint would also be effective for accessing unit-stride 1D vectors. A similar scheme was used to predict address streams from post-increment addresses in the HP PA-7200 microprocessor [Gwe94b].

8.3 Vector Memory Consistency Models

A memory consistency model specifies the possible orderings in which concurrent memory accesses can be observed by the agents initiating memory accesses. Consistency can be considered at three possible levels in a vector ISA: between processors (inter-processor consistency), between instructions (inter-instruction consistency), and between elements within an instruction (intra-instruction consistency).

8.3.1 Inter-Processor Consistency

The first level is inter-processor consistency. The inter-processor consistency model defines the possible orderings in which a processor can observe the memory transactions of other processors. Inter-processor consistency has been extensively studied in the context of cache-coherent scalar machines [AG96] and the same techniques can be applied to vector multiprocessors.

8.3.2 Inter-Instruction Consistency

The second level is inter-instruction consistency. The inter-instruction consistency model defines the possible orderings in which different instructions from the *same* instruction stream can observe each other's memory accesses. Scalar processors almost invariably adopt a strict consistency model for instructions in a single instruction stream, requiring that all memory accesses are observed in program order. In contrast, most vector ISAs adopt a very loose inter-instruction memory consistency model. Typically there is no guarantee on the order in which vector loads and stores are performed by a single processor, unless there is an intervening memory barrier instruction of the appropriate type. Figure 8.5 gives example pieces of code, showing how barriers should be inserted to guarantee desired behavior.

The aim of such a loose inter-instruction consistency model is allow a sequence of vector memory instructions to be issued to multiple VMUs, where they can then execute concurrently without the need for hardware to check for collisions between the multiple address streams. Vectorizing compilers have to perform extensive compile time analysis of memory dependencies to detect vectorizable code, and they can pass on the results of this analysis to hardware by omitting memory barriers when unnecessary. This explicit inter-instruction consistency model provides a large cost saving compared to the huge numbers of address comparators required to support the concurrent execution of hundreds of long latency memory operations with a typical scalar ISA.

An important special case where barriers can be omitted is shown in Figure 8.6. A barrier is not required between a load and store to the same location if the store has a true data dependency on the value being read.

Scalar memory instructions are also included in the inter-instruction consistency model. Scalar accesses in a vector machine are generally defined to observe other scalar accesses in program order to simplify scalar coding, or perhaps to remain compatible with a base scalar ISA. But to simplify hardware implementations, the ordering of scalar memory accesses with respect to vector memory accesses can remain undefined except for explicitly inserted barrier instructions. Software can often determine that scalar and vector accesses are to disjoint memory regions and omit unnecessary barriers.

The standard inter-instruction barriers make all accesses globally visible to all virtual processors (VPs). Virtual processor caches described in the next chapter make use of an even weaker form of inter-instruction barrier to improve performance — VP barriers only guarantee that accesses made by one VP are visible to other accesses by that same VP.

Inter-Instruction Memory Barrier Types

There are four basic types of memory access to consider when specifying barriers to maintain inter-instruction consistency: scalar read (SR), scalar write (SW), vector read (VR), and vector write (VW). Machines with VP caches (Chapter 9) also have to consider VP reads (PR) and VP writes (PW). PR and PW can be considered subsets of VR and VW respectively to reduce the total number of potential hazard types. Table 8.3 lists the possible memory hazards that can occur between these six types of memory access.

One possible design for the memory barrier instruction is to have a single instruction with multiple bits, one per type of ordering that must be preserved (12 for Table 8.3). Alternatively, the hazard types can

```
sd.v vv1, a1     # Store data to vector.
ld.v vv2, a1     # Load data from vector.
# vv2 != vv1

# Add barrier to ensure correctness.
sd.v vv1, a1     # Store data to vector.
membar           # Ensure vector reads after writes.
ld.v vv2, a1     # Load data from vector.
# vv2 == vv1
```

Figure 8.5: With a weak inter-instruction memory consistency model, the first code sequence has no guarantee that the values written by the earlier vector store will be visible to the later vector load, even though both are executed in program order on the same processor. For example, the machine may buffer the vector store in one VMU while the load obtains values directly from memory in a second VMU. The second code sequence adds an explicit memory barrier to ensure the load sees the results of the store.

Abbreviation	Hazard Type
SRAVW	Scalar Read After Vector Write
SWAVR	Scalar Write After Vector Read
SWAVW	Scalar Write After Vector Write
VRASW	Vector Read After Scalar Write
VRAVW	Vector Read After Vector Write
VWASR	Vector Write After Scalar Read
VWAVR	Vector Write After Vector Read
VWASW	Vector Write After Scalar Write
VWAVW	Vector Write After Vector Write
PRAPW	VP Read After VP Write
PWAPR	VP Write After VP Read
PWAPW	VP Write After VP Write

Table 8.3: Possible hazards between scalar and vector reads and writes.

be encoded to reduce the size of instruction space devoted to barriers. Some machines have coarser control. For example, the Cray architecture only allows the user to specify that all vector writes must complete before following instructions, or that all vector reads must complete before following instructions [Cra93]. The Fujitsu VP200 [MU84] had a separate bit on load and store instructions that could restrict them to a single pipeline to prevent overlap. The Torrent ISA [AJ97] has only a single `sync` instruction that waits for all hazards to be resolved.

Memory barrier instructions need not stall the issue stage immediately. Rather, they set up hardware interlock flags which prevent a subsequent memory operation from violating the desired constraint. For example, library routines might frequently end with a memory barrier to ensure that memory has the correct values at the end of the routine, but this will not stall issue unless the caller tries to access memory before the consistency conditions are satisfied. In-order machines with a single memory unit used for both scalar and vector memory accesses can simply discard memory barriers when encountered in the instruction stream.

It would be conceivable to implement finer control over memory consistency by identifying the order between individual instructions rather than between classes of instruction. This requires some way of tagging and identifying instructions and is not considered further here.

```
## A[i] = A[i] + 1
ld.v vv1, a1
addu.vs vv1, vv1, t0
sd.v vv1, a1  # No barrier required.
```

Figure 8.6: This piece of code does not need a memory barrier because the write of `A[i]` cannot happen before the read of `A[i]` due to the true RAW dependency through the `addu`.

8.3.3 Intra-Instruction Consistency

The third level is intra-instruction consistency. A single vector instruction specifies multiple parallel memory operations which might be executed concurrently across several lanes. For unit-stride and constant stride stores, the order in which elements are written is usually immaterial because the elements do not overlap except for a few pathological stride values.[4] But there are some algorithms which can only be vectorized if indexed stores write values in element order. For example, the defined ordering allows virtual processor execution to be prioritized.

While applications generally do not require that vector loads are retrieved from memory in element order, some memory barriers can be avoided if we define that values read are chained to other computations in element order. An example is given in Figure 8.7. Here, the order in which load elements are guaranteed to be used in the computation can affect performance. If the load values are used in sequence, then no memory barrier is required, and the load and store can be chained together through the adder and run concurrently. If the load values are not used in sequence, an explicit memory barrier is required between the load and the store, prohibiting chaining between the two.[5] Fortunately, guaranteeing element order should not impact implementation performance. The memory system can perform element loads out-of-order, as long as chaining logic ensures load values are used in-order. For stores, write buffers can drain out-of-order to different memory words, provided that writes to the same word complete in element order.

One situation where relaxing element order simplifies implementations is for the virtual processor caches described in the next chapter. Vector memory instructions with rake cache or histogram cache hint bits set are specified to have no element ordering guarantees to avoid the need for virtual processor cache consistency hardware.

8.4 Vector Memory Unit Implementation

Vector memory units (VMUs) execute vector memory instructions which move data between vector registers and memory. A single vector instruction is converted into some number of separate requests to the memory system. The VMU is responsible for translating virtual addresses into physical addresses, ensuring coherence with any caches, and routing requests between the memory system and individual lanes.

One simple way to reduce memory system cost is to reduce the number of VMUs per CPU. For example, recent Cray Research supercomputers (X-MP, Y-MP, C90, T90) have included two load VMUs and one store VMU per multiply and add unit. This represents a 3:2 memory:compute ratio. In microprocessor designs, it is much cheaper to increase arithmetic performance than main memory bandwidth, and so it is natural to design "unbalanced" machines with memory:compute ratios less than one, for example with one VMU and two VAUs giving a 1:2 memory:compute ratio. As shown in Chapter 11, many kernels and applications have memory:compute ratios that are 1:2 or less. Even where code has larger memory:compute

[4] A stride of zero, or a large power-of-2 stride that wraps around the virtual address space.

[5] If this was part of a stripmined loop, the second code sequence could still overlap the store of one iteration with the load from the next, but this overlap would only occur with longer vectors.

ratios, the excess of inexpensive arithmetic units helps to ensure the expensive memory port is kept fully saturated. A further advantage of a single VMU design is that inter-VMU memory conflicts are eliminated.

The following discussion of VMU implementation is broken into two parts, first unit-stride accesses, then strided and indexed accesses.

8.4.1 Unit-Stride Accesses

Regardless of the number of lanes, a unit-stride access can always be handled with only a single address transaction per cycle. To simplify TLB and cache tag lookups, a unit-stride access can always request naturally-aligned blocks from memory. For example, a system with four 64-bit lanes could always request data in aligned 256-bit blocks from memory, as shown in Figure 9.1. Only a single TLB access is required per cycle because the block can never straddle a page boundary. Only a single cache tag access is required to maintain coherency provided the cache lines are no smaller than the block size (32 bytes for the example). Further reductions in TLB bandwidth and access energy are possible if a single translation is reused for the multiple cycles within a single unit-stride access.

Because a unit-stride vector can start at any element in memory, alignment circuitry is required to align the first element in memory with lane 0, the second with lane 1, and so on. Misaligned accesses can proceed at full bandwidth by using a combination of an element rotator, a delay register, and a per-lane multiplexer. The rotator is set to a given rotate amount by the start address of the vector and stays at the same rotation for the duration of the unit-stride instruction. The delay register holds the output of the rotator

```
## If load elements used in element order.
## A[i] = A[i+1] + 1
addu a2, a1, 8
ld.v vv1, a2 # A[i+1]
addu.vs vv1, vv1, t0
sd.v vv1, a1

## If load elements used in undefined order.
** A[i] = A[i+1] + 1
addu a2, a1, 8
ld.v vv1, a2 # A[i+1]
addu.vs vv1, vv1, t0
membar,vwavr # Barrier required.
sd.v vv1, a1
```

Figure 8.7: Example where order in which elements are read affects execution. If load elements are guaranteed to be used in element order, then because `A[3]` must be read before `A[2]` can be written, the first code sequence is guaranteed that `A[2]` will be read by the load before it is written by the store. This code sequence could dispatch the load to one VMU and the store to a second VMU with both running in parallel. If the load element use order is not guaranteed, an explicit barrier is required to prevent `A[2]` being written before it is read, as shown in the second code sequence. In this case, the load must complete in the first VMU before the store can begin execution in the second VMU.

network for one cycle, to enable words from two neighboring blocks in memory to be merged to form an element group. The final multiplexer in each lane selects the next element either from the block arriving from memory or from the previous memory block held in the delay register. The multiplexers are also set once at the start of the memory instruction according to the start address. The output of the multiplexers is an element group that can be written to one row of the vector register file. Figure 8.8 shows the operation of the rotate and skew unit for a vector load. A similar network can be built to transmit store data from the vector register file to a vector in memory.

8.4.2 Strided and Indexed Accesses

With little additional cost, strided and indexed accesses can run at the rate of one element per cycle using the hardware described previously for unit-stride accesses. On every cycle, the memory block holding the desired element is fetched from memory and the rotate network is used to move the element to the correct lane. Indexed vector store instructions require a read port on the vector register file to communicate indices to the address generator as well as a read port for data. But if the implementation has multiple lanes and only supports a single indexed memory operation per cycle, a single read port can be multiplexed between data and index reads with little loss in performance by buffering multiple read values across the lanes.

Some small strides can be handled faster with special case circuitry at little additional expense. For the example machine, stride-2 accesses, which are the most common stride after unit-stride, can proceed at the rate of two elements per cycle with only a single address transaction per cycle. The rotator needs to be extended to allow the word permutations required for stride-2 accesses. Stride-3 accesses could also potentially be accelerated to an average rate of 4/3 words per cycle.

Increasing the performance of larger strides or indexed accesses requires considerable additional circuitry. To run at the same rate as unit-stride, an address generator, TLB port, and cache tag port must be provided for each lane. This address hardware can either be distributed across the lanes or centrally located. The memory system must be capable of supporting multiple independent requests per cycle, either with multiported storage or with interleaved memory banks. Similarly, the extra bandwidth to TLB and cache tags can be provided either by interleaving or multiporting the TLB or cache storage. For smaller strides, TLB bandwidth requirements could potentially be reduced by reusing translations when successive elements lie on the same page. Hardware must detect any conflicts between multiple accesses on the same cycle and stall requests until conflicts are resolved.

While this more complicated addressing hardware could also support unit-stride accesses, dedicated unit-stride hardware as described above might save considerable energy by minimizing the number of TLB translations and cache tag lookups, and by eliminating conflict resolution.

Providing full address bandwidth out to main memory is a large part of the expense of a vector supercomputer memory system, and it is likely that a more cost-oriented vector microprocessor will have reduced address bandwidth to main memory. This will tend to dilute the benefit of providing greater address bandwidth in the processor core. The virtual processor caches described in Chapter 9 aim to reduce the

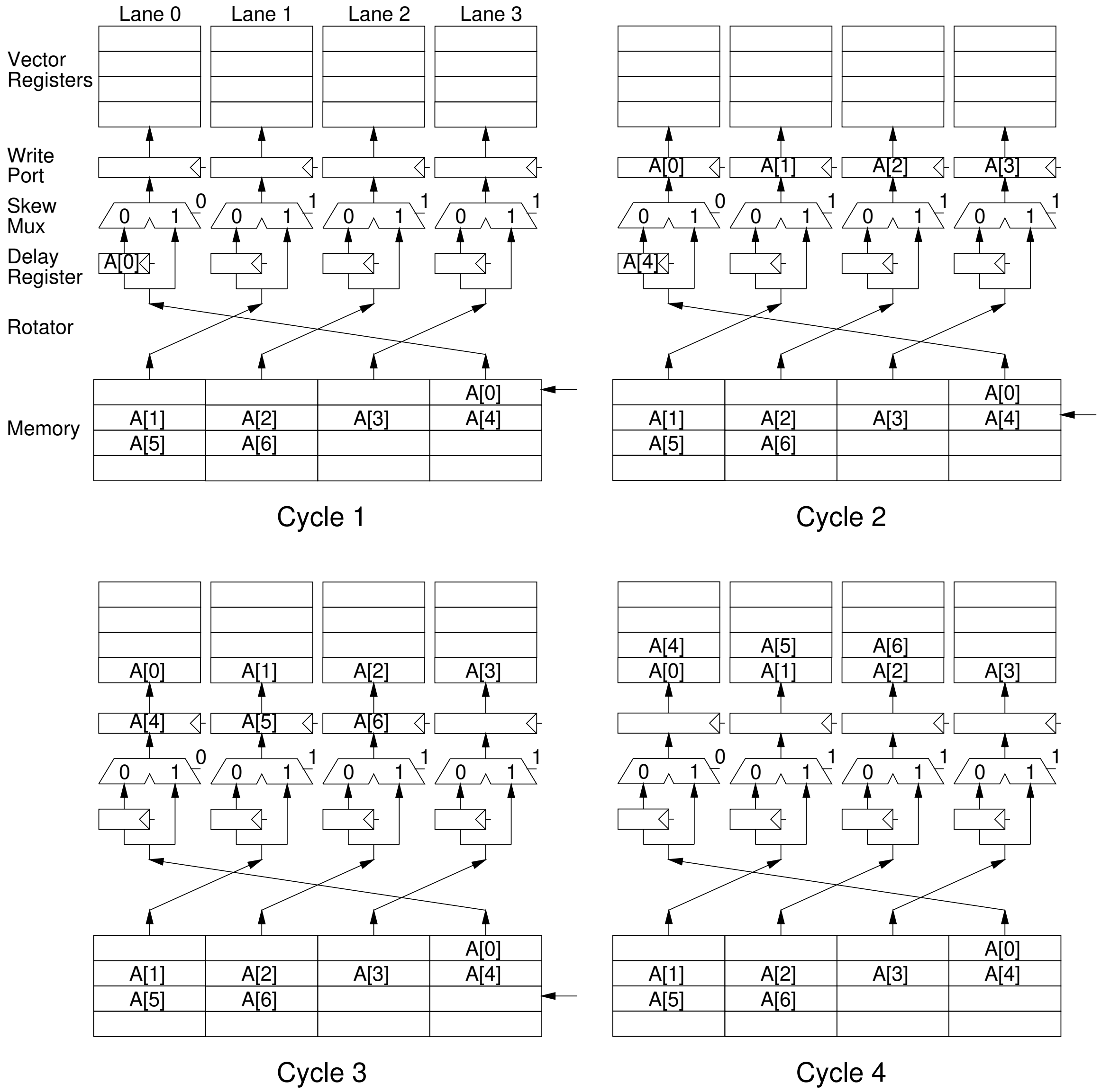

Figure 8.8: Operation of a rotate and skew network used to implement unit-stride loads with only a single address transaction per cycle. The example is loading a memory vector A[0]–A[6] into one vector register of a four lane vector unit. The memory vector is aligned starting at the last word of the four-word aligned memory system. On the first cycle, A[0] is read from memory into the delay register in lane 0. On the second cycle, elements A[1]–A[3] are merged with the delayed A[0] value to prepare an element group to be written into the vector register file. At the same time, element A[4] is clocked into the delay register. The third cycle merges A[5]–A[6] from the last memory block with A[4] from the delay register, while the previous element group is written to the vector register file. The fourth cycle completes the write of the second element group into the vector register file.

address bandwidth demands on lower levels of the memory hierarchy by converting certain forms of strided and indexed accesses into unit-stride bursts.

8.4.3 Subword Accesses

So far, the discussion of memory systems has assumed that each VP is always fetching or storing a complete 64-bit word. In practice, many algorithms, especially those outside of scientific and engineering computing, operate on a range of different data sizes. A 64-bit wide virtual processor might also need to access 8-bit, 16-bit, or 32-bit data. This requires richer connectivity in the memory interconnect. To reduce complexity, a two-level interconnect can be used. The first level is a crossbar between the lanes and the memory system which can perform every permutation of 64-bit words. The second level is within each lane, and includes a byte rotator that can insert or extra any subword from the 64-bit memory word. Section 7.3 on page 133 describes how to subdivide lanes into smaller lanes if all arithmetic can be performed with a narrower word width.

8.5 Memory Storage Technologies

In this section, I review the characteristics of current memory technologies which could be used in a vector microprocessor memory system. The storage components can be divided into four broad categories: on-chip SRAM, off-chip SRAM, on-chip DRAM, and off-chip DRAM.

8.5.1 On-chip SRAM

On-chip SRAM is the highest performance but also the most expensive form of storage, and is typically used to build the highest levels of the memory hierarchy. The amount of SRAM on microprocessor dies has been increasing rapidly, with current microprocessors having capacities up to around 128 KB [BAB$^+$95, Gwe96c, Gwe95a, Yea96], while the HP PA-8500 with 1.5 MB of on-chip SRAM has been announced [LH97]. For current designs, access latency of on-chip SRAM is in the range 2–6 processor cycles, with 3 cycles being typical. On-chip SRAM is typically pipelined to provide one access per cycle regardless of addressing pattern. Some more aggressive designs access the SRAM twice [Gwe96a], or even three times [Gwe96c], in one clock cycle to get greater bandwidth without adding extra physical ports or interleaving banks. Several designs have moved to two levels of on-chip cache [BAB$^+$95, Cas96, Gwe96b] to provide short latencies to a small primary cache for high frequency scalar processor cores, while providing a larger secondary cache to avoid off-chip accesses. This style of design is attractive for a vector microprocessor, where the vector unit bypasses the scalar primary cache and accesses the on-chip secondary cache directly.

8.5.2 Off-chip SRAM

Off-chip SRAM is typically used to build large external caches for microprocessor systems. Off-chip SRAM has moderate access latency, of around 3–12 processor clock cycles, with 7 clock cycles being a typical value from load instruction to use. Because of pipeline conflicts and bus turnarounds, these SRAMs often require one or two dead cycles between off-chip reads and writes. Current off-chip SRAM parts cycle at the same clock rate or one-half the clock rate of current microprocessors. Newer fast burst SRAM designs increase bandwidth by returning data on both edges of the SRAM clock, doubling the data bandwidth but with the same address bandwidth as before.

8.5.3 Off-chip DRAM

Off-chip DRAM provides main memory for most computing systems today, as it has the lowest cost per bit amongst high-speed mutable semiconductor memories. Recent developments in high-speed DRAM interfaces have dramatically increased the bandwidth available from commodity DRAM. A single synchronous DRAM (SDRAM) offers data rates of up to 250 MB/s. A single Rambus DRAM (RDRAM) can supply a peak data bandwidth of 600 MB/s per part. Although the peak data bandwidth of these new high-speed DRAMs is competitive with that of off-chip SRAM, they have much greater latency and considerably less address bandwidth. These properties are inherent in the structure of DRAM.

DRAMs represent data as charge in storage capacitors. The storage cells are arranged in large arrays with sense amplifier latches located at one edge of the array. Once a row of data (typically equal to the square root of the memory size, or several kilobits) is captured in the sense-amp latches, it can be accessed as fast as SRAM. For a typical SDRAM, a new piece of data can be returned from this sense-amp row buffer every clock cycle (8–12ns). Future double-data rate (DDR) SDRAMs will increase data bandwidth by transferring data on both edges of the clock. RDRAMs have much the same internal structure as an SDRAM, but can transfer data from the row buffers at a much greater rate.

Random accesses are much slower. To save area, many DRAM cells share the same wordline. On a row access, each cell develops only a small potential on the bitline and sensing is a delicate and slow process. The long word lines and slow sensing contribute to the long latency of random access for DRAM. First, the bank must be precharged to equalize the bitlines in preparation for sensing. This precharging destroys the values in the sense-amp row buffers. Next, the row address is decoded and used to drive a wordline, and the new row of data bits is captured in the sense-amp buffers. Finally, a column address is decoded and used to select data from the row buffer. DRAM readout is destructive and so a restore period is required after data is latched into the sense amps. During restore, the bitlines are driven full rail by the sense amps to restore the charge on the active storage cells. The sum of precharge, row access, and restore times fixes the bank cycle time.

For a typical SDRAM running with a 3 cycle CAS latency, precharge takes 3 SDRAM cycles, row access takes 3 cycles, and column accesses can be pipelined every cycle but with a 3 cycle latency [IBM96]. Precharge and row access cannot be pipelined, and random accesses to a bank have 8 times less bandwidth than column accesses. To help improve random access bandwidth, current SDRAMs have two independent banks. Precharge and row access in one bank can be overlapped with column accesses to the other. Future SDRAMs will increase this number of banks to four or more.

RDRAMs have much the same performance characteristics as SDRAMs, except with much reduced pin requirements due to the higher speed multiplexed interface. A single 600 MHz 8-bit Rambus channel is roughly equivalent to a 75 MHz 64-bit SDRAM bus, but requires only 31 pins on a memory controller including power and ground, whereas the SDRAM bus would require around 110 pins. Because RDRAMs can provide full channel bandwidth from a single part, the minimum memory increment is also much smaller.

Column access latencies for DRAMs are around 30 ns. Random access latencies are around 50 ns. If precharge cannot be hidden, then row access latencies increase to around 80 ns. Currently specialty DRAMs are available which sacrifice density for lower latencies, and these latencies will continue to drop slowly in future DRAM generations.

These latencies are only for single DRAM parts. In large commercial computing systems, many DRAMs are required to provide bulk main memory. Caches, memory controllers, and memory interconnect add to memory access latency in large systems. For example, in the AlphaServer 8400 system with 300 MHz Alpha 21164 processors and a 75 MHz system bus, the minimum read latency is around 240 ns with 60 ns access DRAM parts [FFG+95]. Workstations have somewhat lower overhead. For example, the AlphaStation 600 workstation has a memory latency of 180 ns with the same 60 ns DRAMs [ZML95]. Newer designs are reducing the latency overhead. For example, the Digital Personal Workstation has a highly integrated memory controller [Sch97a] and with no tertiary cache achieves memory access latencies as low as 75 ns on a hit to an active DRAM row, or 105 ns to an inactive row.

Technology trends will continue to reduce this overhead latency. First, main memory controllers, which are separate chips in most microprocessor-based systems, will be integrated on-chip as transistor count grows. This integration reduces the latency of responses to cache misses, and allows for more intelligent handling of DRAM request streams. Second, especially in smaller systems, the number of DRAMs required to provide all of main memory is falling [PAC+97]. This reduces the size of the memory interconnection network, allowing DRAM to be directly connected to the processor or even embedded on-chip as described below. The combination of an on-chip memory controller and directly connected DRAM allows access latencies to approach the raw DRAM access latencies. Even large server-class systems are moving to distribute DRAM amongst processor nodes, with nodes connected via a high-speed interconnect [LL97]. The access latency of each node to its local DRAM should also fall as the memory controller is integrated and the number of local DRAM parts falls. Even with these improvements, processors running at 1 GHz can expect local DRAM latencies of up to 60–80 cycles.

DRAM bandwidth, on the other hand, will be limited more by system cost than by technological barriers. In the next few years, packaging technologies will soon allow 32–64 GB/s off-chip data bandwidth using 256–512 I/Os signaling at 1–2 Gb/s per pin. This bandwidth can be supplied by 8–16 next-generation 64 Mb RDRAM parts [CDM+97], with a total minimum memory increment of 64–128 MB. Lower-cost systems will rely on on-chip memory hierarchies to reduce off-chip bandwidth demands. As described next, for some systems, *all* of main memory can be integrated on-chip.

8.5.4 On-chip DRAM: IRAM

With each DRAM generation, capacity per chip increases by a factor of 4 while cost per bit decreases by only a factor of 2. This trend has led to a drop in the number of DRAM parts in a given class of system over time. As the number of DRAMs required for a system drops below one, it is natural to consider integrating processor and DRAM memory on the same die yielding "intelligent RAMs" (IRAM) [PAC+97].

This integration brings several benefits. First, latency drops substantially because bus and control overheads are reduced. Random access latencies of 25 ns and below are possible. Second, both data and address bandwidth increase dramatically. For electrical reasons, DRAMs are divided into many smaller arrays of around 1–2 Mb each, but these are grouped together and treated as single banks in commodity DRAMs. If the memory controller is on the same die as the DRAM arrays, it can control each array individually. Data bandwidth is increased because wide parallel on-chip busses can be use to retrieve hundreds of bits per array per column cycle, and multiple such arrays can be active on a cycle. Address bandwidth is increased because independent row accesses can be distributed over many arrays. Finally, energy consumption is reduced because there are no off-chip capacitative busses to drive [FPC+97]. Data bandwidths are expected to be well over 100 GB/s for a single die in the 1 Gb DRAM technology.

There would appear to be a natural synergy between IRAM and vector architectures [KPP+97]. IRAM can provide high address and data bandwidth with low energy consumption, while vector architectures offer a compact processor design that can nevertheless convert high memory bandwidths into application speedups.

8.6 Driving DRAM in a Vector Memory System

SRAMs can accept a new address every cycle regardless of addressing pattern. DRAMs present extra complications due to their two-dimensional structure. Accesses to the same row in a bank can occur rapidly in a pipelined fashion, while accesses that require a change in row address take longer and require a sequence of actions to be performed.

8.6.1 DRAM Control Pipeline

DRAM control can be simplified if all accesses are performed with a uniform pipeline as shown in Figure 8.9. For this example, I use timings for an off-chip synchronous DRAM array [IBM96] operating with a 3-cycle CAS latency. Similar pipeline designs can be used for Rambus DRAM or embedded DRAM. For every separate bank in the DRAM, the memory controller keeps track of the row that is currently present in the sense-amps. At the start of the DRAM pipeline the memory controller determines if the access is a hit or miss to the open row in the addressed bank. For a row miss, the memory controller initiates a precharge and row access in the pipeline to bring the new row into the sense-amps in time to read out values during the column access stages. If the row miss is to an otherwise idle bank, then precharge and row access can be overlapped with column accesses to active banks. One detail to note is that overlapped row and column accesses may both attempt to use a shared DRAM address and control bus on the same cycle, in which case one access will have to stall. If the row miss is to an active bank, the precharge cannot begin until any previous column accesses and restores to the same bank have completed because the precharge will destroy the values in the sense-amps. In this case, the memory controller must stall the new access until this bank busy time is over. For a row hit, the precharge and row access stages do nothing but simply delay column accesses until the appropriate stage in the pipeline. This fixed length pipeline trades increased latency of row hits in order to simplify control and to increase bandwidth in the presence of row misses. With a current processor cycle time of around 2 ns and current 8 ns SDRAMs, the SDRAM component of memory latency would be around 36 cycles.

Unit-stride accesses dominate vector workloads, but non-unit stride and indexed accesses will tend to incur more row misses, and row accesses are around eight times slower than column accesses for an SDRAM memory system. For scientific codes studied by Hsu and Smith in [HS93], miss rates from a four bank system

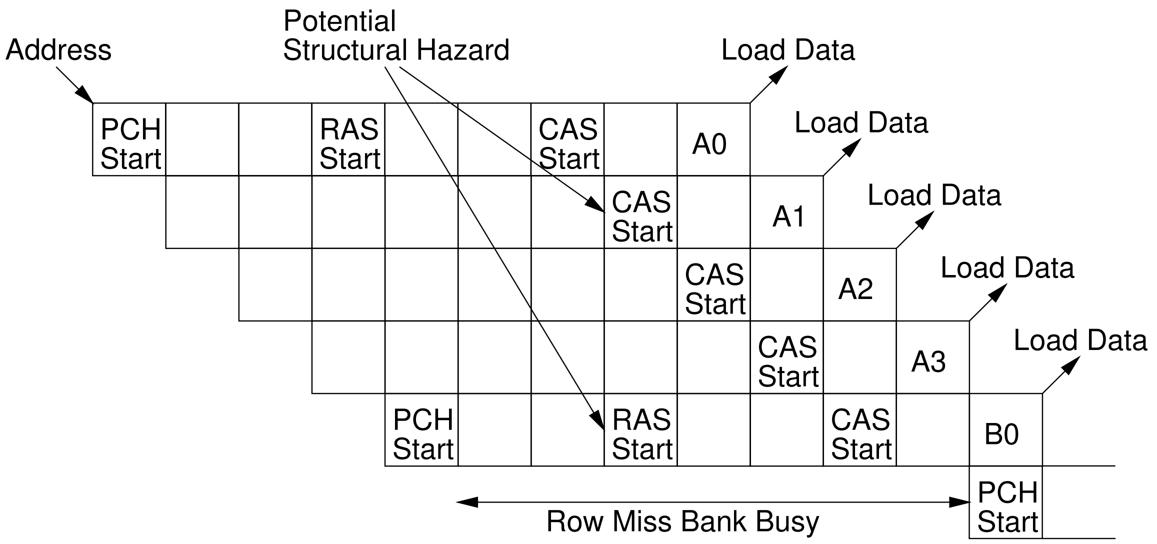

Figure 8.9: Fixed length pipeline used to communicate with SDRAM memory system. The figure assumes an SDRAM with a three cycle CAS latency. The pipeline includes sufficient time for a precharge and row access before each column access. Each of these three phases has a latency of three clock cycles. The column accesses can be pipelined, one per cycle, but the other two phases reserve the bank for their whole duration. The figure shows a sequence of four read accesses to the same bank (A0–A3), followed by a read access to a second bank (B0), followed by another read to a different row in the second bank. The first five reads return at the rate of one per cycle, with the precharge and row access to the second bank overlapped with accesses to the first bank. Depending on the manner in which column accesses are performed (burst length) and whether both SDRAM banks share memory control lines, there could be a structural hazard between the the CAS command for the A1 access and the RAS command for the B0 access. The final access shows the stall that is required when another access to the B bank is required. The column access for B0 must complete before beginning the precharge of the new access.

with 2 KB per open row were found to be around 10%, suggesting that even a small number of open rows can capture significant locality even in the presence of strided and indexed operations. In the absence of bank busy conflicts and with the pipeline described above, these miss penalties could be overlapped. Increasing the number of banks, either by using DRAMs with more banks per chip, or by using parallel arrays of DRAM chips, decreases the probability of bank busy conflicts. Current SDRAMs and Rambus DRAMs integrate two banks on each chip, with future parts planning four or more. Designs with DRAM on-chip can have dozens to hundreds of banks.

8.7 Vector Caches

Current commercial vector supercomputers do not employ vector data caches, although they were included in some earlier designs [Buc86, Jon89]. Vector data caches would not improve performance of these machines significantly, because they provide full address and data bandwidth out to main memory and because many vector programs are tolerant of memory latency [KSF+94].

If main memory bandwidth is reduced, however, several studies [GS92, KSF+94, FP91] have shown that vector data caches can improve performance significantly. These studies also demonstrate that caching strategies used for scalar processors are not optimal for vector data.

Capturing spatial locality to reduce address bandwidth is less important than for a scalar data cache, because vector unit-stride or strided instructions already accurately encode the presence of spatial locality. This reduces the benefit of the long cache lines and speculative prefetching used to exploit spatial locality in scalar caches. But vector data caches can help capture spatial locality in some other higher level patterns of vector access, such as permutations or rakes.

The main benefits of a vector data cache arise from capturing temporal locality. Reusing vector data from cache reduces data bandwidth demands on the lower levels of the memory hierarchy. Large capacity and high associativity, or other techniques to avoid conflicts [Yan93], help to capture temporal locality.

Vector data caches can also improve performance by reducing memory latency, though this is less important than in scalar processors due to the inherent latency tolerance of vector instructions. In particular, speculative prefetching to reduce latency may be beneficial for latency-sensitive scalar processors but fetching additional speculative data can slow down a latency-tolerant vector processor [KSF+94].

Because only some vector data will benefit from caching, it is desirable to select those vector accesses that should be cached. Some possible schemes for distinguishing accesses that should be cache allocated include a “cache” hint bit on vector data instructions, “cacheable” bits on page table entries, or some form of dynamic cacheability predictor. Other no-allocate vector accesses will still operate in cache if there is a hit, but do not allocate data in cache if there is a miss.

8.7.1 Using Scalar Secondary Cache to Cache Vector Data

All current high-performance microprocessors rely on large caches to boost scalar performance when running from DRAM memory systems. Often these caches are arranged in two or more levels, with typical outermost cache sizes of between 256 KB–4 MB. Some designs [Cas96, LH97] are integrating these large caches on-chip to increase bandwidth while reducing latency, power, and system-level costs.

One possible configuration for a vector microprocessor is to provide private primary caches for the scalar processor, and then to share a larger secondary cache between vector and scalar units. Attaching the vector unit to the secondary cache will require extra interconnect that can potentially increase the latency of scalar accesses. The scalar primary caches decouple the scalar unit from the longer latency secondary cache, and allow scalar and vector accesses to occur in parallel.

An important function of the shared secondary cache is to provide communication between scalar and vector units for partially vectorizable loops. Communicating blocks of vector data via the cache can be faster than using scalar insert and extract instructions to access vector registers directly. Scalar insert and extract instructions interfere with vector processing by tying up vector register file access ports and by inhibiting chaining. They can also interfere with vector register renaming (Section 4.5). Scalar accesses to cached vector data can lower average vector element access latency by fetching multiple words at once using long primary cache lines.

8.7.2 Scalar-Vector Coherence

The simplest hardware scheme for managing coherence between scalar and vector accesses is to make the scalar primary data cache have a write-through policy and to invalidate scalar primary cache lines on vector stores. The scalar write-through traffic will be caught by the secondary cache. Vector stores need to check scalar primary cache tags to invalidate any matching lines. This tag port can either be shared with the scalar unit or implemented as a separate second port. A second port removes the vector tag accesses from the scalar tag lookup critical path, and also allows scalar accesses to proceed in parallel with vector stores. The second tag port can also be used to maintain interprocessor and I/O coherence. If the vector unit can generate multiple store addresses per cycle, multiple ports into the cache tags will be required.

The secondary cache will likely be managed with a write back policy to reduce main memory traffic. All vector loads and stores must check the secondary cache tags for hits. Most scalar accesses will be filtered by the primary caches before reaching the secondary cache, and so the scalar unit can share a secondary cache tag port with the vector unit.

8.7.3 Non-blocking Secondary Cache

A single vector memory instruction can touch many cache lines and main memory latencies can be very long, and so the secondary cache should be non-blocking to allow multiple outstanding cache misses to be in progress simultaneously. Hardware should allow multiple vector accesses to the same cache line to be merged as is common with merging scalar accesses in current scalar processors [ERB+95].

Consider for example the neighbor access pattern shown in Figure 8.1. The accesses to the A matrix can be marked cacheable because it is known each element will be used three times. Assume that the A vector is initially uncached. The first vector load to A encounters cache misses and begins fetching a sequence of cache lines, with status information held in a set of miss status registers. The first load completes execution after all the cache misses are initiated. Assuming a decoupled architecture, the arithmetic instructions can issue to the instruction queues where they will wait for data to arrive from memory. With a long memory latency, the second and third vector loads will issue before the first cache line for the first load has returned. The second and third vector loads will touch the same set of cache lines as the first, plus one additional line. These misses can be merged with those of the first. With a decoupled pipeline, each vector load will reserve a separate load data buffer. As cache lines arrive from memory, each piece of data will be written into three different load data buffers, effectively amplifying main memory bandwidth by a factor of three.

8.7.4 Transient Vector Caches

Note that in the previous example of a neighbor access, most reuse happens while data is still in flight through the memory system. The outstanding cache miss entries act as a fully associative cache when snooped by new loads. Instead of marking the neighbor access as cacheable, we can rely on this memory access filtering to reduce traffic. This prevents the neighbor access from sweeping through and evicting more useful data from the secondary cache.

The idea of treating buffers holding data en route to and from the memory system as a fully-associative cache has been suggested in several earlier proposals. A similar idea was used in the HP-PA 7200 microprocessor [Gwe94b] where an *assist* cache was used to hold newly accessed cache lines until it was determined they should be moved to main cache. Espasa [Esp97b] noted that snooping on memory store queues can reduce vector memory traffic. While the original motive in his work was to reduce vector register spill traffic, some non-spill traffic was also avoided.

Another example use is for 1D unit-stride vector access. The vector memory accesses within the 1D vector access will usually be misaligned on cache boundaries. Usually, a unit-stride vector memory instruction will only bring the requested data into the CPU. But if the unit-stride vector memory instructions include a base register post-increment as described above in Section 8.2.6, and this indicates that the next access to the memory vector will start at the end of the current access, all of the last cache line in can be prefetched and held in a small vector cache. The start of the next vector memory access in the 1D vector will hit on this prefetched cache line, reducing total address bandwidth to main memory.

8.8 Multiprocessor Vector Machines

Vector supercomputers have supported increasingly parallel multiprocessor configurations for many years. For example, the number of processors supported in the high-end Cray series has doubled with each generation from 4 in the Cray X-MP to 32 in the current Cray T932. Traditional vector supercomputers extend the single flat memory system to all processors. This large, flat, high-bandwidth memory is very expensive to provide, and is difficult to scale to larger numbers of processors.

One trend in microprocessor-based systems is towards distributed directory-based cache-coherent multiprocessors (CC-MP) [HP96, Chapter 8]. These machines are structured as a set of nodes, where each node includes one or more processors with caches, a portion of global memory, and connections to an inter-processor communications network. Each node contains hardware that implements a directory protocol to maintain coherence between the caches on all nodes. A commercial example is the SGI Origin 2000 series [LL97]. The same form of distributed cache-coherent architecture can be adopted for vector microprocessor based systems. While a full study of the implications of such designs is beyond the scope of this thesis, a few issues are mentioned briefly here.

At first it might appear that adding a vector unit to the processors within a CC-MP would only aggravate interconnection bandwidth bottlenecks. In fact, vector units may actually reduce bandwidth requirements of parallel applications running on CC-MPs. Where there is little or no temporal locality, it is more efficient to move data using uncached vector memory instructions than to use cache lines. Also, where interconnect is not currently the bottleneck, vector units might speed up local processing to the point where the expensive interconnect is fully utilized.

Cached scalar processors exploit spatial locality with long cache lines that group multiple requests to amortize address transaction overhead. If only spatial locality can be exploited, caching the data cannot reduce data bandwidth demands, and can increase total data traffic by evicting data that would otherwise be reused. Also, total interconnect traffic can increase because of the extra coherence traffic. Longer cache lines can incur penalties due to *false sharing* [HP96, Chapter 8]. Accesses with no locality, such as large strides or indexed accesses over large data sets are problematic in current CC-MPs. Although only single words are touched, whole cache lines are moved; this dramatically amplifies bandwidth demands.

Alternatively, the scalar processors could perform uncached memory operations where there is little temporal locality, either by setting page table entries or by using alternate instruction encodings. Where there is spatial locality, uncached scalar stores could potentially reduce address bandwidth demands with merging write buffers that group multiple scalar writes into a single block transfer. Uncached scalar loads are more difficult to merge because these might also be latency critical; if load requests are sent as soon as possible, then opportunities for merging with later loads will be missed. Load merging might still be possible in cases where earlier loads have to be stalled in the local node due to limited interconnect bandwidth. A bigger problem with uncached scalar loads is that it is difficult to tolerate main memory latencies while maintaining high throughput. To address this problem, the Cray T3E MPP system, which maintains a global uncached shared memory, adds an external memory-mapped vector fetch engine to a scalar microprocessor [Sco96].

Vector memory instructions offer several advantages in dealing with accesses with little temporal locality. Vector memory instructions describe multiple independent memory requests and have weak inter-instruction consistency models, which simplifies highly parallel implementations with deep buffers. Vector memory instructions accurately encode spatial locality information, allowing more intelligent data fetch.

Using uncached vector memory instructions, a vector microprocessor can transfer data directly between home node memory modules and local vector register files. No cache transactions need occur between the requesting node and the home node, but the home node may need to fetch dirty data from the current owner of any lines touched and may need to invalidate any cached copies on a store.

The rake cache and histogram caches described in the next chapter should be particularly helpful in a CC-MP environment where the reduction in address bandwidth corresponds to fewer cache directory transactions. Furthermore, the global accesses made by these VP caches do not have to be kept coherent with accesses from other processors, it is sufficient to invalidate them at memory barriers and inter-processor synchronizations; VP cache refills can be treated as uncached block loads and VP cache writebacks can be treated as uncached block stores. Similarly, the transient vector caches described above need not keep fetched cache lines coherent with other processors.

Chapter 9

Virtual Processor Caches

In this chapter, I introduce *virtual-processor caches* (VP caches), in which every virtual processor is given a separate cache. I present two types of VP cache, a spatial VP cache called the *rake cache* and a temporal VP cache called the *histogram cache*. The caches are named after typical access patterns that exploit the qualities of each cache. The rake cache is smaller and can be implemented without the histogram cache. If a histogram cache is implemented, a rake cache is easy to provide using the same hardware. These caches can substantially reduce the address and data bandwidth requirements of vectorized code.

9.1 The Rake Cache

The rake cache is a spatial VP cache that reduces address bandwidth demands by converting strided and indexed rake accesses into unit-stride accesses. While the rake cache is so named because it is an ideal match to *rake* accesses (Section 8.2.5), it can also benefit other access patterns that exhibit the same kind of spatial locality. Even for machines with full address bandwidth support, a rake cache can reduce intra-vector conflicts in memory system crossbars.

The vector machine shown in Figure 9.1 is used as an example to help describe the rake cache. This machine has four 64-bit lanes with a single VMU that can perform either loads or stores. The memory system has only a single address port but can return a 256-bit word every cycle. There is a single address generator, a single-ported TLB, and a single port into the scalar cache tags for vector stores to maintain coherency with the write-through scalar data cache. To simplify the explanation, I assume an ideal memory system which can return a 256-bit word every cycle regardless of addressing pattern.

The memory unit can perform unit-stride loads and stores at the rate of four words per cycle. As described in Section 8.4 on page 150, strides of two and three could run at the rate of 2 words per cycle and 1.33 words per cycle respectively, but all other non-unit stride and indexed operations can only run at the rate of one word per cycle. Each memory access, however, pulls in three other words that are not used immediately. The basic idea behind the rake cache is to cache all of the words fetched in each memory access

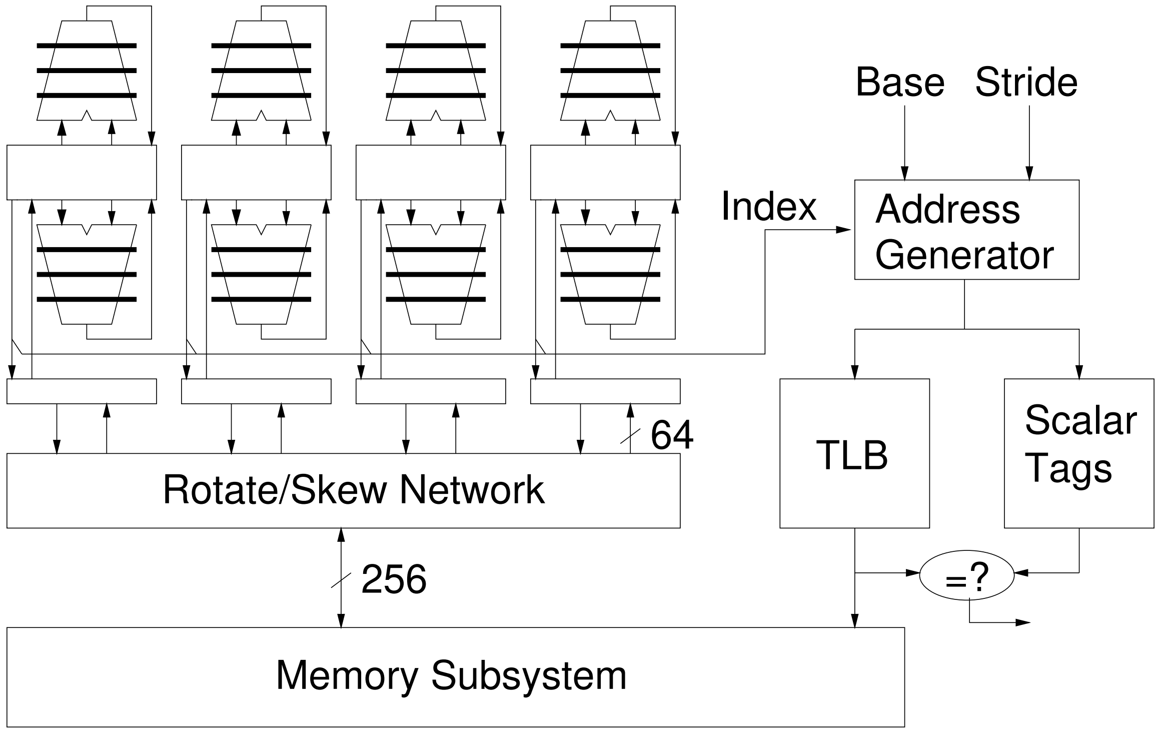

Figure 9.1: Example basic vector unit used to help explain the benefits of a rake cache. The vector unit has 2 VAUs and 1 VMU in 4 lanes. Every access to the memory system transfers a single 256-bit naturally aligned block. While the rotate/skew network allows arbitrarily aligned unit-stride loads and stores to run at the rate of four 64-bit accesses per cycle, strided and indexed loads and stores can only complete at the rate of one element per cycle. The VMU read port is multiplexed between data and indices during indexed stores.

because we expect to soon access neighboring words.

9.1.1 Rake Cache Structure

A rake cache has some number of *rake entries*. Each rake entry has VLMAX cache lines, one per virtual processor. Each cache line in a rake entry has a virtual tag, a physical page number, one valid bit, and some number of dirty bits. The virtual tag contains the virtual address of the cache line, minus any bits required to address bits within one cache line. The physical page number (PPN) caches the translation of the virtual page address into a physical page address. Only enough address bits to hold a physical page number are required as the low order physical address bits are the same as in the virtual tag. The number of dirty bits required depends on the granularity of the smallest write, for example, one per byte if the machine supports byte writes. Figure 9.2 shows an example rake cache with four rake cache entries, for a machine with VLMAX virtual processors and 16 vector registers.

The most flexible rake entry indexing scheme adds a field to each strided and indexed instruction to indicate which rake entry should be used for that access. In a typical vector loop containing a rake access, the same vector register will be used to perform each strided or indexed vector memory access to the rake. An alternative rake cache indexing scheme makes use of this fact by associating a rake entry with the vector register used to access the memory data; the high order bits of the vector register specifier are taken from the vector memory instruction and used as a direct-mapped index into the rake cache. Although this scheme

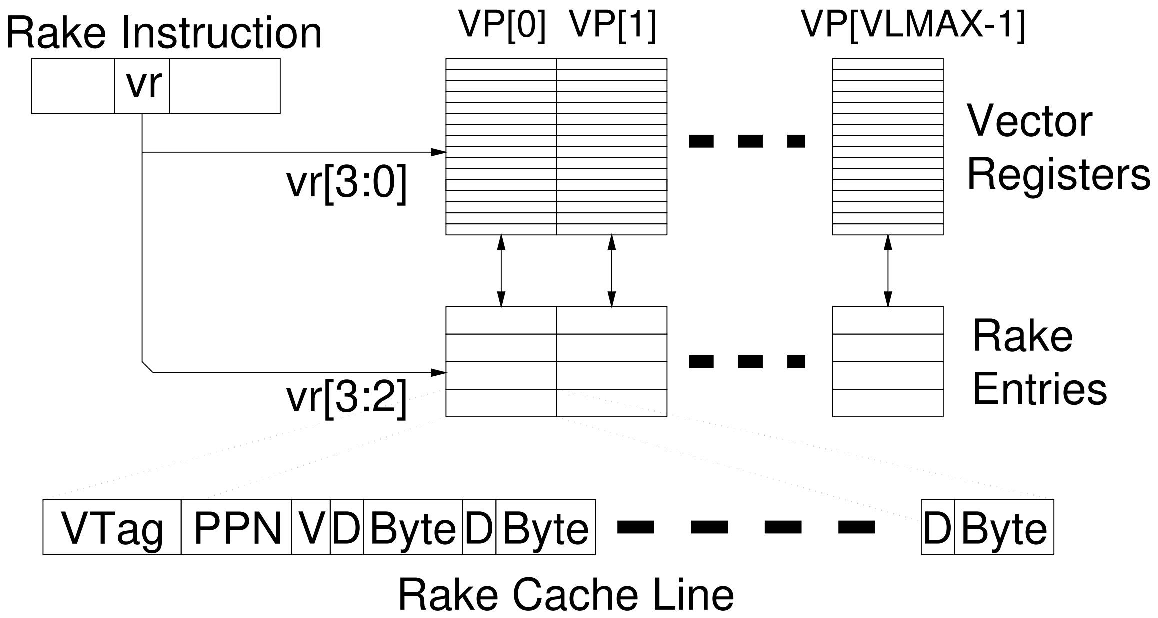

Figure 9.2: Rake cache structure. In this example, the machine has 16 vector registers and 4 rake cache entries. The top two bits of the register specifier in a rake instruction are used to select the rake cache entry. Each cache line has a virtual tag (VTag), a physical page number (PPN), one valid bit (V), and a dirty bit (D) for every byte of storage.

is less flexible than including a rake index field in every instruction, it uses fewer instruction encoding bits which may be important if vector extensions are added to existing scalar instruction sets. This scheme is shown in Figure 9.2 where the four bits of the vector register specifier select one of the 16 vector registers, while the top two bits of the specifier select one of the four rake cache entries. A programmer or compiler aware of the mapping between vector register number and rake entry can avoid rake entry conflicts by careful allocation of vector registers in loops containing rake accesses. If software is not aware of the rake cache, more sophisticated rake entry indexing schemes, such as set-associative, may be appropriate but these are not considered further here.

9.1.2 Rake Cache Allocation

The processor requires a scheme for determining which vector memory instructions to allocate in the rake cache. Unit-stride instructions always bypass the rake cache. Because the presence of a rake access pattern is usually straightforward for the compiler or programmer to determine, the simplest and most accurate scheme is to provide a "rake hint" bit in the encoding of strided and indexed vector memory instructions. If an implementation does not provide a rake cache, the rake hint bit can be ignored and the instructions treated as regular strided or indexed memory instructions.

If adding an explicit rake hint bit in the instruction encoding is not possible, the hardware requires some form of rake prediction scheme. One simple and cheap static prediction scheme is to predict that all strided accesses are rakes, or that all strided *and* all indexed accesses are rakes. This could potentially cause rake cache pollution when both rake and non-rake strided and indexed access patterns occur in the same loop nest. Alternatively, the hardware could perform some form of dynamic rake access prediction scheme, e.g., by checking that subsequent accesses by the same strided or indexed instruction were unit-stride displacements from the previous access, similar to such proposals for stream buffers [PK94].

9.1.3 Rake Cache Examples

As an example of using the rake cache, consider the C code and assembler inner loop in Figure 9.3 which perform the submatrix multiplication of two submatrices into a third submatrix where all are part of larger matrices with differing leading dimensions. This example assumes that vector register number is used to select a rake entry. The compiler has allocated vector registers such that the various rake accesses map to different rake entries, avoiding any inter-rake conflicts. A given cache line is mapped to a single rake and a single VP. This ensures that the cache line will only be evicted when the VP finishes using the data within the cache line and is ready to move to the neighboring line in memory. Note that this example could have been vectorized in the orthogonal direction by interchanging the `i` and `j` loops to give unit-stride memory accesses, but if the number of rows (N) is large and the number of columns (M) is small, the direction shown in Figure 9.3 will yield longer vectors.

Conversely, the compiler or programmer can access the same rake entry with different vector registers. This is useful, for example, for loading different vector registers with the separate fields within

```
/* Original loop. */
for (j=0; j<M; j++)
  for (i=0; i<N; i++)
    C[Crow + i][Ccol+j] = A[Arow+i][Acol+j]*B[Brow+i][Bcol+j];

/* With stripmining and loop interchange. */
for (i=0; i<N; i+=VLMAX)
  for (j=0; j<M; j++)
    for (ii=i; ii<min(i+VLMAX,N); ii++)
      C[Crow+ii][Ccol+j] = A[Arow+ii][Acol+j]*B[Brow+ii][Bcol+j];

/* Assembler code for j loop. */
jloop:
  ldst.v,rake vv0, r3, r4  # Access to A into vv0 => rake entry 0.
  addu r3, 8               # Bump A pointer.
  ldst.v,rake vv4, r5, r6  # Access to B into vv4 => rake entry 1.
  addu r5, 8               # Bump B pointer.
  mul.d.vv vv8, vv0, vv4   # Multiply vectors.
  sdst.v,rake vv8, r7, r8  # Access to C from vv8 => rake entry 2.
  addu r7, 8               # Bump C pointer.
  bne r3, r9, jloop        # Check for end condition.
```

Figure 9.3: Example of multiple rakes in the same loop. The example C code multiplies two submatrices to yield a result in a third submatrix. The compiler stripmines the inner loop, then performs a loop interchange such that the matrices are processed in strips of VLMAX rows. The innermost `i` loop can now be performed with single vector instructions, while the middle `j` loop needs only one rake entry for each rake. The assembler code for the middle `j` loop shows how careful allocation of rake accesses to vector registers avoids conflicts between the different rakes.

```
/* Original loop. */
complex A[][], B[][], C[][];
for (j=0; j<M; j++)
  for (i=0; i<N; i++)
    C[Crow+i][Ccol+j] = A[Arow+i][Acol+j]*B[Brow+i][Bcol+j];

/* Assembler code for j loop after stripmining */
/* and loop interchange. */
jloop:
  ldst.v,rake vv0, r3, r4 # A.real into vv0 => rake entry 0
  addu r3, 8              # Bump A pointer to imaginary
  ldst.v,rake vv4, r5, r6 # B.real into vv4 => rake entry 1
  addu r5, 8              # Bump B pointer to imaginary
  mul.d.vv vv8, vv0, vv4  # A.real*B.real
  ldst.v,rake vv1, r3, r4 # A.imag into vv1 => rake entry 0
  addu r3, 8              # Bump A pointer to next element
  mul.d.vv vv9, vv1, vv4  # A.imag*B.real
  ldst.v,rake vv5, r5, r6 # B.imag into vv5 => rake entry 1
  addu r5, 8              # Bump B pointer to next element
  mul.d.vv vv10, vv1, vv5 # A.imag*B.imag
  sub.d.vv vv8, vv8, vv10 # C.real = A.real*B.real-A.imag*B.imag
  sdst.v,rake vv8, r7, r8 # Store C.real from vv8 => rake entry 2
  mul.d.vv vv12, vv0, vv5 # A.real*B.imag
  addu r7, 8              # Bump C pointer to imaginary
  add.d.vv vv9, vv9, vv12 # C.imag = A.imag*B.real+A.real*B.imag
  sdst.v,rake vv9, r7, r8 # Store C.imag from vv9 => rake entry 2
  addu r7, 8              # Bump C pointer to next element
  bne r3, r9, jloop       # Check for end condition
```

Figure 9.4: Example C code and assembler code for multiplying two complex submatrices to yield a result in a third complex submatrix. Note how the assembler code accesses the real and imaginary parts of each element using vector registers that map to the same rake entry.

structures accessed by rakes. Figure 9.4 shows the complex version of submatrix multiply as an example. Unlike the previous real submatrix multiply example, the complex submatrix multiply can not be vectorized with unit-stride loads and stores.

9.1.4 Rake Cache Implementation

The rake cache requires storage for the rake cache lines and tags. Because a given cache line is only accessed by the corresponding virtual processor, the rake cache tag and data storage can be striped across the lanes. This provides a high bandwidth multiported cache with no possibility of port conflicts. To execute strided and indexed accesses at the rate of one per lane per cycle, an address generator is now required in every lane. Figure 9.5 shows the rake cache circuitry required for one lane of the example machine.

Assuming that the machine has a decoupled pipeline as described in Section 4.7, the tags will be accessed much earlier in the pipeline than the data. The rake cache line storage is split into the the tag portion, which is located by the address generators, and the data portion, which is located between the main memory bus and the vector register file. The index values used to access the cache data storage have to be delayed until the appropriate point in the memory pipeline, and so a small FIFO RAM is required.

Because the index of the rake entry is determined by bits in the instruction word and the index of the cache line within the rake entry is determined by the virtual processor number, lookups in the rake tag memory can occur *before* the virtual address is generated. After the address is generated, only an equality comparison of the prefetched virtual tag and the virtual address is required to determine hit or miss. Thus the rake cache has only a minor impact on memory latency, typically much less than one processor clock cycle.

Writes do not check the valid bit — writes hit if and only if the tag matches. Write hits update the appropriate portion of the cache line and set the appropriate dirty bits. When there is a write miss, the direct-mapped indexing scheme means there is only one possible victim line. If the victim cache line contains any set dirty bits, the modified pieces of the cache line are written back to main memory. Write misses are handled using a write-allocate with write-validate policy. After any dirty data is written back, the virtual tag and PPN is updated, the valid bit is *cleared*, and the cache data and dirty bits are updated to reflect the single word of new dirty data.

Reads will only hit if the tag matches *and* the valid bit is set. On a read miss, any dirty data is written back and the missing line is fetched from memory. As with scalar write-back caches, the victim write back may be postponed until after the missing line is read. The required word is forwarded to the vector register file while the rake cache data storage is written with the new cache line. The valid bit is set and all dirty bits are cleared.

Note that this simple policy generates a read miss even when a line is present in the cache after being brought in by a write. Also, in this case if the dirty victim line is written out *after* the missing read line is fetched from memory, the read will see the old unmodified data. This is of no concern, however, because for the writes to be visible to the read, the programming model requires the insertion of an explicit memory barrier which would have previously flushed the rake cache.

For lanes with rake cache hits, data transfers occur entirely within the lanes; operands move between the rake cache data storage and the vector register file. These transfers occur at the same point in the pipeline as if requests had gone out to memory. This approach simplifies the memory pipeline design. The rake cache does not aim to reduce memory access latency because the vector model and the decoupled vector pipeline (Section 4.7) already hide rake cache miss latency from the point of view of each virtual processor.

Only lanes with rake cache misses need to access the memory system. Read misses need to perform a virtual to physical address translation to obtain the new PPN (and to check for any page faults), potentially write back a dirty victim line, and read the missing memory block from the memory system. Writes also need to perform an address translation to obtain the new PPN, and may also need to write back a dirty line.

We can further exploit the spatial locality in a rake to reduce the number of accesses made to the central TLB. Note that each virtual tag and physical page number holds the mapping for a complete page of data containing many cache lines. As part of checking the full virtual tag to detect a cache line miss, the

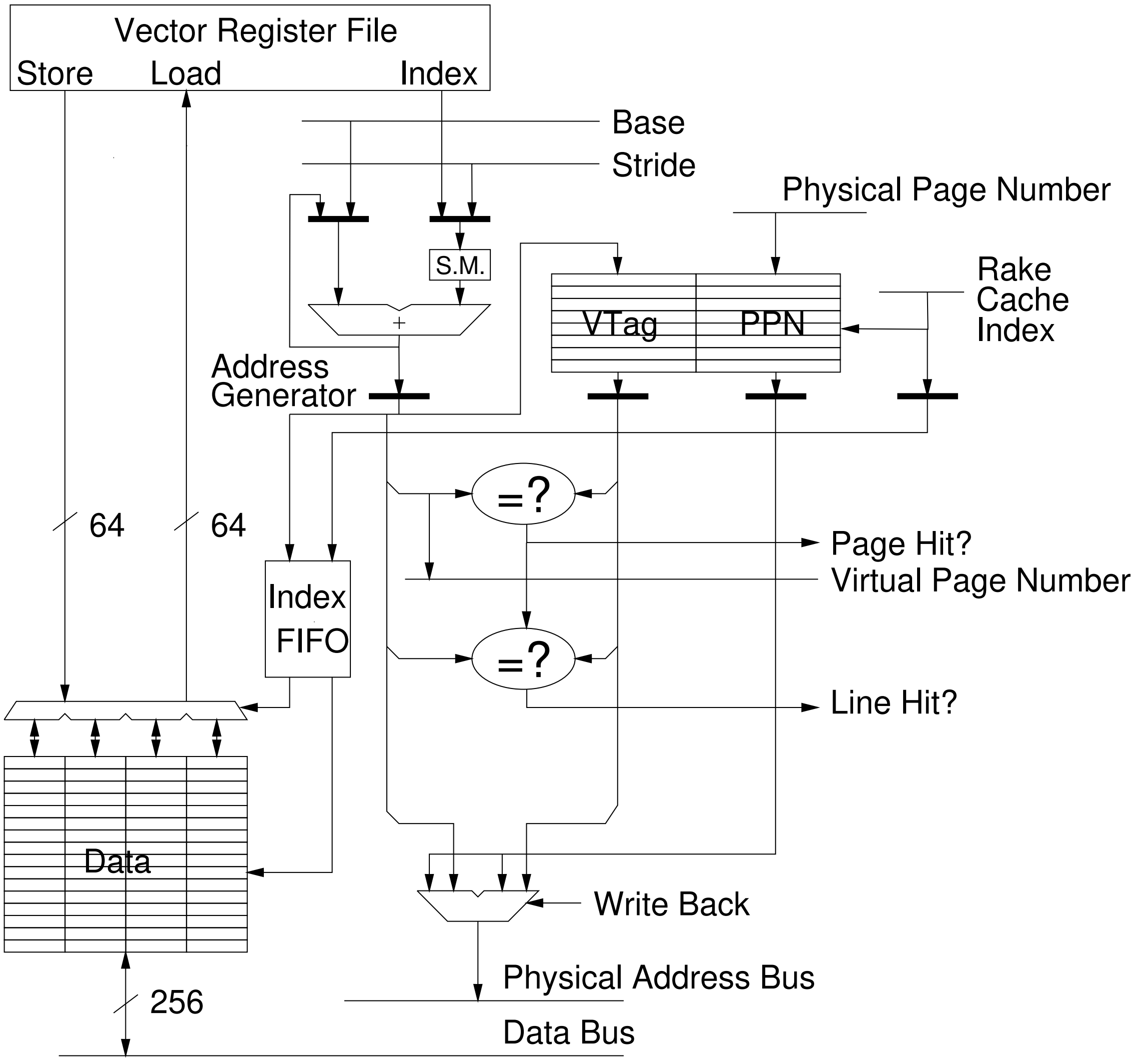

Figure 9.5: Implementation of rake cache for one lane in example architecture. Base addresses and strides are broadcast across the lanes, as are rake cache indices, while index values for an indexed rake come from the vector register file. The box labeled "S.M." is a small shift and carry save array used during the first cycle of a strided access to multiply stride values by 0, 1, 2, or 3, to give the correct starting value for addresses in different lanes. Thereafter, the stride value multiplied by 4 is added to each lane's address register. The broadcast rake cache index contains the rake entry number and the element group number, and this is used to access the rake cache tags. This rake cache index is combined with the local word index of the rake access and held in a local FIFO until the correct point in the pipeline, where the combined index is then used to access the rake cache data. The data RAM is read and written in 256-bit blocks to the memory system, while the lane reads 64-bit words into and out of the vector register file using the store data and load data ports. A rake access which misses the cache but hits in the same virtual page, performs main memory accesses using the existing PPN. If there is also a page miss, the virtual page number is transmitted to the central TLB for translation. Handling of valid and dirty bits is omitted for clarity.

virtual page number of the tag is compared with the virtual page number of the new address. If these match, the stored PPN can be used without requiring another TLB lookup. In effect, the rake cache lines act as a distributed TLB with one entry per unit-stride stream. This saves the energy of accessing the central TLB and also increases effective TLB capacity. The rake cache must be flushed before any changes are made to system page tables.

A priority resolution network monitors hit signals across all four lanes, and serializes requests to the central TLB and the memory system. The physical address used to access the memory is obtained by concatenating the PPN to lower order address bits from the generated virtual address. All write-backs check the write-through scalar data cache tags and invalidate matching lines to maintain coherence. If the scalar cache is virtually indexed, then some bits of the virtual address may also have to be transmitted to enable lookup of scalar tags.

A rake cache is a fairly small structure. Assume our example has 32 64-bit elements per vector register (32 VPs). The total storage for each rake entry is roughly 32*(2+4) words or 1536 bytes. A reasonably sized rake cache for this machine, with say 4 entries, would require roughly 6 KB of storage.

During some operations, the rake cache storage needs to provide sufficient bandwidth for a read and a write to maintain full performance during the frequent cache misses. On a tag miss, the old tag must be read for the tag comparison and then subsequently updated with the new tag. On a victim write back, the old data must be read out before new data is written into the cache line. While the bandwidth could be provided with dual-ported storage cells, the regular nature of vector memory instruction accesses allows two single-ported storage banks to be interleaved by element group, similar to element partitioning in the vector register file (Section 5.2). The write of a new tag or new data to one bank can occur while the second bank provides the tag or data access for the next element group.

The example above used a simple memory system for clarity. There are often more complicated tradeoffs between address and data bandwidth as described earlier in Section 8.1, and these affect the choice of rake cache line length, as do the rake depths found in applications. The appropriate number of rake cache entries also depends the rake widths found in applications and on the degree of loop unrolling required. Chapter 11 contains data on the number and type of rakes present in vectorizable applications.

9.1.5 Rake Cache Coherency and Consistency

The rake cache contains multiple parallel caches, one per virtual processor per rake cache entry. The same cache line may reside in all caches; one pathological case is for a rake stride of zero. If a loose intra-instruction consistency model (Section 8.3) is adopted for rake accesses, the rake cache is free of the usual worries associated with maintaining multiple cached copies of the same memory location. No guarantees are made on the order in which elements within a single vector instruction are written. Software is required to insert explicit memory barrier instructions to guarantee the ordering of accesses between separate vector memory instructions.

When a memory barrier requires that all writes be made visible to all reads, the rake cache entries are flushed by writing back any dirty data to memory and then clearing all valid and dirty bits. If the same location was written in two different rake cache lines, write backs can happen in any order and only one of the updates remains visible. If a memory barrier requires that new writes are made visible to subsequent reads, all rake cache valid bits are cleared.

Weaker VP consistency barriers can be used to order accesses by each VP, without requiring that all accesses are visible across all VPs. For example, in a graphics rendering algorithm where scan lines are distributed across VPs, writes made to the Z-buffer by one VP need only be made visible to that VP. An intervening PWAPR consistency barrier (Section 8.3) will ensure that the write is made visible to the next read by that VP.

The rake cache is virtually tagged to allow rake cache tags to be checked without TLB accesses. The use of virtual tags can give rise to *synonyms* [HP96, pages 422–425] where two cache lines with different virtual addresses map to the same physical address. Because the loose vector memory consistency model already allows two rake cache lines to contain different dirty data for the same location, synonyms require no special handling. If the virtual tag does not include an address space identifier, however, the rake cache will have to be flushed on virtual address space changes.

Finally, there is also no problem with *false sharing* [HP96, Chapter 8], because no coherency checks are required and because each cache line only writes back modified bytes.

9.1.6 Rake Cache Performance

The memory unit in the example machine processes up to four rake accesses per cycle, one per lane. A rake write access can either result in a hit to the rake cache with zero memory accesses required, or a miss which requires at most one memory access to write back a dirty victim line. A rake read access can result in a hit which requires zero memory accesses, a rake miss with a clean victim which requires a single memory access for the missing line, or a rake miss with a dirty victim which requires two memory accesses, one for the missing line read and one for the victim write back. Rake reads will only find dirty victim lines if a rake entry is shared between read and write accesses. To simplify the following performance analysis, I assume that read-modify-write rakes are allocated two rake entries, one for read accesses and another for writes; this ensures that at most one memory access is required to service a rake cache miss.

The memory system can service a single rake cache miss for one lane and return the data at the same point in the pipeline as when the other three lanes transfer data from the rake cache. Hence if zero or one of the requests in an element group causes a rake cache miss, the rake accesses can proceed at the rate of four per cycle. If more than one rake cache miss occurs in an element group, the memory system will take multiple cycles to service the multiple rake cache miss requests. The simplest strategy for dealing with multiple rake cache misses in one cycle is to stall further element groups until all rake cache misses for the current element group are serviced.

For strided rakes, the worst case occurs for rake strides that are multiples of four. With a naturally-aligned four word cache line, all four virtual processors in an element group will experience their rake cache misses on the same cycle, but the next three rake accesses by this element group will experience cache hits. Hence it will take 4+1+1+1 cycles to perform every 4+4+4+4 word accesses, or a sustained average of 2.29 words per cycle after initial misses warm up the rake cache. Rake strides of the form $4n + 2$ will take 2+1+2+1 cycles to complete 4+4+4+4 accesses, for an average of 2.67 words per cycle. The best case is for odd strides, which guarantee only a single rake cache miss per cycle and so can sustain 4 words per cycle. Assuming a uniform distribution of rake strides, average throughput will be 3.24 words per cycle after startup. In many situations and particularly for loop raking, however, it is possible to select an odd stride and hence achieve optimal throughput [ZB91].

For indexed rakes, throughput depends on the rake index vector. Assuming a uniform distribution of index values, expected average throughput is 2.84 words per cycle, with the same worst case of 2.29 words per cycle as for strided rakes.

This simple rake cache design experiences stalls when multiple memory accesses must be made on the same cycle. By adding some buffering to the rake cache it is possible to spread the rake cache misses over time so that the average performance for deep rakes is maintained at 4 words per cycle regardless of rake stride or rake index vector. The following sections describe how to add rake write buffers and rake read prefetch buffers.

9.1.7 Rake Cache Write Buffers

A write buffer is a straightforward addition to the rake cache that improves performance for even rake strides or non-uniformly distributed rake index vectors. If a rake access by an element group requires more than one memory access, any writes that cannot be serviced immediately are deposited in a write buffer and scheduled for write back when the memory port is free. The write buffer must be flushed when a memory barrier instruction requires that outstanding writes be completed. To handle the worst case situation where every VP in a rake instruction requires a write back, each write buffer entry requires VLMAX lines and each line is permanently associated with a given VP. Each write buffer cache line contains a physical address, a full bit, and storage for a data line together with dirty bits. The full bit indicates that this is a full write buffer line. The data and dirty bits are simply copies from the rake cache victim line. The write buffer is striped across the lanes, so victim transfers from the rake cache to the write buffer are contained within the lane.

A single loop nest may contain multiple write rakes with differing rake strides or rake index vectors. At first, this would appear to lead to a tricky write back scheduling problem. At any time, multiple rake write buffers may be filled with any number of dirty lines awaiting write back, and ideally every dirty line in the write buffer would be written back to memory before the corresponding VP evicted another dirty victim for the same rake. Fortunately, the regular predictable pattern of a rake access makes it straightforward to provide optimal scheduling of the write back of this dirty data.

For example, consider the worst case for a rake store on the example machine where the rake stride is a multiple of four. Starting from a clean rake cache, on the first access to the rake, all rake cache lines are write validated to the new line and no memory traffic is generated. When the rake reaches the end of the first rake cache line, all VPs will generate a rake cache miss and try to write back one dirty line each. Each element group would take four cycles to service at this point without a rake write buffer. With a write buffer, we can retire the first lane's write back to memory while holding the other three lanes' writes in their respective write buffer lines. This allows the rake store to proceed at the rate of four words per cycle, leaving three out of four of the lines within the rake's write buffer full of dirty data. As described above, the next three executions of the same rake store instruction will experience no rake cache misses on any VP, as they will hit in their respective rake cache lines.

The key observation is: *we can schedule the outstanding write backs during subsequent write accesses to the same rake.* This policy simplifies the design of the write buffer, because the same index that is used to index the tags can be used to scan the write buffer entries to find waiting write backs. As we process each element group, if there are no rake cache misses, we write back one of the buffered writes in that element group to memory. Within the element group, we know which lane will next require a write back because the word index of its current access will be closest to the end of the current cache line. If multiple VPs within the same element group have the same highest word index, we select the one in the earliest lane; this prioritizes lower numbered VPs which will be those used if the vector length is reduced at the end of a rake. For both strided and indexed rakes, this policy ensures we empty the write buffer lines in optimal order.

This policy also performs correctly in the case of a read-modify-write rake, spreading memory traffic over the read rake and write rake instructions. After startup, no rake writes will experience a rake cache miss because the preceding read of the same line will have brought in the required data. However, when a VP moves to read the start of the next cache line, it will have to write back the dirty data from the last cache line as well as fetch the missing data. This requires two memory accesses. If the victim write back is held in the rake write buffer then only one read memory access is required immediately. The subsequent rake write, which never experiences a rake cache miss, leaves the memory port free to perform the write backs.

The number of write buffer entries may be less than the number of rake cache entries, with write buffers allocated to rakes dynamically as needed. A read-only rake does not require a rake write buffer, and the allocation policy may choose not to allocate a write buffer entry for write rakes with odd stride (there may still be some benefit if there is other memory port traffic).

The rake write buffer ensures that all rake writes always proceed at the rate of four words per cycle after the startup period, *regardless of rake stride or rake index vector*. The predictable nature of rake accesses also makes it possible to perform a highly accurate read prefetch, and achieve the same result for rake reads.

9.1.8 Rake Cache Read Prefetch Buffer

Similar to the write buffer, the rake cache read prefetch buffer makes use of otherwise unused memory cycles during read rake accesses to prefetch the next blocks required in each unit-stride stream. Each entry in the read prefetch buffer includes a virtual tag, a PPN, and a valid bit. Rake reads check the tags of both the current and prefetched cache lines. On a prefetch hit, prefetch data is moved into the rake cache line, and the prefetch buffer is marked as empty.

For each element group within a read rake, if there are no rake cache misses or other memory system accesses, the read prefetch brings in the next block for the VP that has an empty prefetch buffer and whose current access is closest to the end of the current rake cache line. If the prefetch generates any page faults during translation, the prefetch is discarded. The fault will be handled during the next rake load instruction.

Note that unlike the write buffer, the read prefetch buffer may generate additional memory traffic. But unlike prefetching schemes for scalar processors, the prefetch is used to spread memory traffic evenly over all cycles *not* to hide memory latency. Each unit-stride stream only fetches at most one block ahead, and only then during otherwise unused memory cycles.

A single set of tag and data storage can be dynamically allocated between rake read prefetch and rake write buffer duties. The system need not allocate a read prefetch buffer for write-only rake, and need not allocate a prefetch buffer for odd rake strides.

9.2 The Histogram Cache

The histogram cache is a temporal VP cache. It provides each VP with its own cached workspace. While the histogram cache is named after its use in histogramming operations, where each VP has its own set of histogram buckets, it has a much wider potential range of uses. For example, a histogram cache should be able to reduce address and data bandwidth demands in sorting, FFTs, texture mapping, and image compression DCTs (Section 11.2.4).

The histogram cache is only accessed by strided and indexed load and store instructions that have an explicit "histogram cache" hint bit set. Unit-stride instructions always bypass the histogram cache. An implementation may choose to omit the histogram cache in which case the histogram hint bit can be ignored and the instructions treated as regular strided and indexed operations.

The design of the histogram cache is very similar to the rake cache, and both can share the same hardware. As with the rake cache, tag and data storage are striped across the lanes. Unlike the rake cache, the histogram cache is indexed with virtual addresses; lines are not associated with rake entries but with the VP as a whole. Again, the lines are virtually tagged to avoid TLB lookups for cache hits.

The histogram cache will often be used to cache a contiguous region of memory as workspace for each VP, and so a direct-mapped indexing scheme should suffer few conflicts. Greater associativity may help some applications, and simplifies integrating a rake cache with the histogram cache. The set-associative tag compare involves reading and comparing multiple cache tags to determine the correct way within the set early in the pipeline, but then only a single cache data access is needed later in the pipeline.

As with the rake cache, the histogram cache associates a virtual to physical mapping with each cache line. Unlike the rake cache, it is less likely that the virtual page number of a line will match when the rest of the tag misses. Often a string of histogram cache misses will occur when a VP moves to a new workspace. In this case, there is considerable locality across the new cache lines but not necessarily between old and new cache lines. To reduce central TLB accesses in this case, the histogram cache can maintain a micro-TLB, for example with one or two entries, that holds the most recent translations for that VP. This should capture most of translation misses as the VP moves to a new workspace region.

The histogram cache can also benefit from the addition of a write buffer. Unfortunately, the more unpredictable pattern of accesses to the histogram cache makes scheduling write back accesses more difficult, but the number of these write backs is expected to be much less than in the rake cache. To simplify a combined hardware design, a similar policy can be used as with the rake cache, writing back dirty victim lines in free memory slots during subsequent histogram cache instructions. The unpredictable pattern of histogram cache accesses also makes it difficult to implement a read prefetch buffer.

9.2.1 Histogram Cache Consistency

As with the rake cache, weaker VP consistency barriers can be used to order accesses by a VP to its local workspace *without* requiring the workspace be made coherent with main memory.

Hardware will also have to ensure that any writes held in a write buffer are made visible to subsequent read misses after a histogram consistency operation. One simple scheme always drains the write buffer storage to memory before the next histogram read miss is serviced. Alternatively, the write buffer storage can be checked for matching addresses with any matching data forwarded to a subsequent load. The latter approach might be straightforward to implement using circuitry already present in a combined rake/histogram cache with rake read prefetch buffers.

The histogram cache also makes use of a slightly enhanced tag and dirty bit policy compared with the rake cache to reduce write traffic for write-allocated lines. In the histogram cache, reads will now also hit if the tag matches but the valid bit is *clear*, provided dirty bits are set in the required portions of the cache line. If the tag matches and the valid bit is clear, but the required dirty bits aren't set, the line is fetched from memory and merged with the dirty data in cache. The valid bit is then set. In effect, the dirty bits are used as sub-block valid bits in a write-allocated line. If a rake cache is implemented as part of the histogram cache, it can also use this enhanced policy.

The histogram cache is expected to be considerably larger than the rake cache. For example, to allow each VP to hold 256 buckets each of 32 bits for sort histogramming requires 1 KB of storage per VP, or 32 KB data storage (plus another 8 KB tag storage) total for the example machine with 32 VPs.

As with the rake cache, the loose intra-instruction consistency model avoids the need to track coherency across the multiple VP caches and avoids the synonym problem with virtual tags. Again, regular memory barrier instructions cause dirty data to be flushed from all cache lines. Because the histogram cache is larger than the rake cache, the time required to flush the cache at a memory barrier is potentially more worrying. In the example design, the 32 KB histogram cache would take 1024 cycles to drain if all lines were dirty. While the histogram cache will not necessarily increase the number of transactions with memory, it can buffer many writes up until a barrier then cause a long delay as all dirty lines are written out. But the histogram is only allocated under software control to data expected to have temporal locality, so it is expected that this writeback overhead will be amortized over multiple accesses.

An even greater saving is possible for some applications. Often, the histogram cache is used as a temporary workspace, and there is no need to write back dirty lines at the end of an operation. An “histogram invalidate” instruction can be provided whose semantics are that all locations written by any histogram store instructions since the last memory barrier will now have undefined values. When a histogram invalidate instruction is encountered, all dirty bits can be cleared without writing back any dirty data.

Whereas the rake cache speeds an access pattern common in vector code and which is relatively easy for a compiler to spot, algorithms may need substantial restructuring to make use of the histogram cache and automatic compilation for the histogram cache will be more difficult. Nevertheless, the histogram cache has the potential to substantially improve the cost/performance of vector microprocessors.

9.2.2 Example: Performing a Histogram Using the Histogram Cache

To help understand the operation of the histogram cache, consider performing a vectorized histogram in the example machine. The basic scheme is to have each VP perform a separate histogram on a subset of the data, then sum the values in corresponding buckets across VPs. There are three steps in the algorithm: clear all buckets, read all input values incrementing appropriate buckets, then sum buckets across VPs and write out the final histogram. Assume that initially there has been a recent memory barrier instruction and so the histogram cache contains no dirty data.

In the first step, each VP is allocated a contiguous region of memory into which it will accumulate its own histogram. Strided histogram stores are used to clear this bucket memory, with the stride being equal to the size of each VP’s bucket space. Note that because of the write-validate policy and because the cache starts clean, these strided stores occur entirely within the histogram cache, clearing four words per cycle while generating no external memory traffic.

In the second step, unit-stride loads are used to read the data to be histogrammed. These unit-stride loads bypass the histogram cache. Vector arithmetic instructions convert the data values into the appropriate local bucket indices. These indices are then used in indexed histogram load to read the bucket value, the value is incremented, then an indexed histogram store is performed to write back the updated value. A VP read-after-write consistency barrier must now be issued so that the next indexed histogram load can observe the updated bucket values. These steps are repeated until all data values have been scanned. Note that during this phase, the only main memory traffic is the unit-stride scan of the data values; all of the indexed histogram bucket accesses are performed within the histogram cache and run at the rate of four per cycle.

In the third step, corresponding buckets across multiple VPs are combined to give the final bucket values. Strided histogram loads are used to retrieve bucket values into vector registers. These histogram loads occur entirely in the histogram cache at the rate of four words per cycle. Then a vector register reduction is performed to yield the final sum, and the single result is written out to the final histogram location. This is repeated for all buckets. Note that multiple independent reductions can be interleaved to increase throughput (Section 10.1). When all final sums have been calculated, the histogram bucket values are no longer needed, and so a histogram invalidate instruction is issued. The only memory traffic during this stage is the write back of the final results.

Throughout the whole histogram process, the only memory traffic was the read of the input data and the write of the final histogram — the minimum possible. All intermediate traffic was caught by the histogram cache.

9.3 Combined Rake and Histogram Cache

The histogram cache is mostly a superset of the rake cache, and both can be implemented using the same hardware. The two caches complement each other well, helping different kinds of memory access. For example, during a vectorized radix sort [ZB91], the rake cache speeds the rake scan of the input data while the histogram cache holds the histogram buckets. For radix sort on the example machine, a 32 KB histogram plus rake cache reduces data bandwidth by around 57% and address bandwidth by around 78%.

The main distinguishing feature of the rake cache is that it uses software allocation of rake entries to index cache lines and eliminate inter-rake conflicts. Higher associativity in the histogram cache could achieve the same effect, but the high turnover in cache lines during a rake operation (at least one miss per cycle in the example machine) amplifies any flaws in allocation policy. Also, the histogram cache is larger, and will tend to leave more lines dirty to be flushed at the next memory barrier instruction, whereas the rake cache evenly distributes write traffic throughout the execution of a loop nest.

While we could partition cache storage statically into histogram and rake cache functionalities, it is straightforward to dynamically allocate storage between the two if we add some degree of set-associativity to the histogram cache. Then we can allow the rake cache to steal at most one line from each set; the histogram cache will continue to function albeit with reduced capacity. We add a "rake-allocated" bit to the tag storage of one line in each set. After a memory barrier instruction, all dirty lines are flushed to memory and all rake-allocated bits are cleared, thereby allocating all rake entries to the histogram cache by default. The first rake access to a set will allocate a cache line, possibly writing back any dirty data caused by earlier histogram accesses. Subsequent rake accesses to the same set will proceed to use the same line. Once a cache line is allocated to the rake cache, it will not be evicted to service histogram cache misses. The rake cache lines are deallocated at the next memory barrier instruction.

A histogram cache can also benefit from the addition of a write back buffer. The same storage can be dynamically allocated between rake cache write buffers, rake cache read prefetch buffers, and histogram cache write buffers. Again, this allocation is cleared at the next memory barrier instruction.

9.4 VP Cache Summary

VP caches have several advantages over providing a single large cache shared by all VPs:

- The caches can be divided into banks by striping across lanes rather than by interleaving across addresses. This eliminates address and data crossbars, and the conflicts that arise in their use.

- By keeping data local to a lane, we reduce energy consumption compared to accessing a large central multiported cache.
- The VP caches effectively provide a high degree of associativity and avoid many of the conflicts that would occur in a shared cache, e.g., for strides that are powers of 2.
- Similar arguments hold for the TLB, with reductions in conflict misses, access port conflicts, and energy consumption.

The main disadvantage is that shared data items may be duplicated in multiple caches, potentially wasting capacity compared to a shared cache.

Chapter 10

Virtual Processor Communication

The virtual processors (VPs) in a vector machine usually only communicate via vector memory instructions that access their common shared memory. In some cases, other forms of inter-VP communications can improve performance but these can incur significant hardware costs to provide the necessary inter-lane communication. This chapter discusses these alternative forms of inter-VP communication.

10.1 Reductions

Reduction operations occur frequently in vectorizable code. Perhaps the most common is the sum reduction; C source code for a simple example is shown in Figure 10.1. Although the loop-carried dependence on the `sum` scalar variable would at first appear to prohibit vectorization, if the additions can be assumed to be associative, they can be reordered to allow parallel execution.

The loop can be stripmined, with each VP accumulating a partial sum in parallel. At the end of the loop, the partial sums must be summed to yield a single value. The sum vector can be repeatedly split in two and the halves added using a vector addition. To sum the first element of the first half of the vector with the first element of the second half of the vector using a vector instruction requires that both are present in the same VP.

This communication can be performed through memory as shown in Figure 10.1. While this approach requires no hardware support, it has two main disadvantages. First, to move N elements from the end of one vector register to the start of another in each reduction step, 2N elements are stored then N elements are loaded. Second, significant latency is added due to the need to wait for the load to see the results of the store.

The data alignment is the critical operation in the reduce. This can be accelerated by adding a vector extract instruction that moves the end of one vector register to the start of another. An instruction to perform this function (called "vector word shift") was added to the Cray C90 because increases in memory latency would otherwise have caused a degradation in the performance of reductions [Oed92]. The Torrent ISA

```
/*
sum = 0;
for (i=0; i<N; i++)
  sum += A[i];

*/

    cfvu t0, maxvl       # Get maximum vector length.
    ctvu t0, vlr         # Set maximum vector length.
    mov.vs vv1, zero     # Clear all partial sums.

loop:
    setvlr t1, a0        # Set stripmine length.
    lw.v vv2, a1         # Get values.
    sll t2, t1, 2        # Multiply by 4 bytes.
    addu a1, t2          # Bump A pointer.
    addu.vv vv1, vv2     # Add in to partial sums.
    subu a0, t1          # Subtract elements done.
    bnez a0, loop        * Any more?

reduce:
    addu t3, sp, work    # Get workspace region on stack.
    li t10, 1            # Stop when vlr==1.
    ctvu t0, vlr         # Reset to maximum length.
rloop:
    sw.v vv1, t3         # Store entire vector.
    srl t0, 1            # Halve vector length.
    ctvu t0, vlr
    sll t2, t0, 2        # Convert to byte offset.
    addu t3, t2          # Move halfway up vector.
    membar,vravw         # Need to see stores.
    lw.v vv2, t3         # Get top half of vector.
    addu.vv vv1, vv2     # Add in to bottom half.
    bne t0, t10, rloop   # Finished?

    # vlr==1.
    sw.v vv1, a2         # Write result to memory.
```

Figure 10.1: Example of a sum reduction. The assembly code uses memory loads and stores to align top and bottom halves of the partial sum vector for vector addition.

```
reduce:
    li t10, 1           # Stop when vlr==1.
rloop:
    srl t0, 1           # Halve vector length.
    ctvu t0, vlr
    vext.v vv2, vv1, t0 # Top half of vv1 into start of vv2.
    addu.vv vv1, vv2    # Add in to bottom half.
    bne t0, t10, rloop  # Finished?

    # vlr==1.
    sw.v vv1, a2        # Write result to memory.
```

Figure 10.2: Sum reduction rewritten using vector extract instruction. The scalar operand to the `vext.v` instruction specifies the start element in the source from which elements should be moved into the destination. The value in the vector length register specifies how many elements to move.

[AJ97] includes a vector extract instruction for this same purpose. Figure 10.2 shows the reduction rewritten to use the vector extract instruction. Now, a single instruction moves just N elements from the end of one vector register to the start of another, and with much lower latency. The vector extract instruction provides a primitive that can be used to build many different types of reduction operation.

The vector extract instruction can be implemented by configuring the VMU to perform the equivalent of a unit-stride store and a unit-stride load simultaneously without actually touching memory banks. The skew network can be set to implement the desired vector shift across lanes. A simple optimization is possible for extract indices that are a multiple of the number of lanes. In this case, which is common during the reduction sequence, elements move between VPs mapped to the same lane. The VMU crossbar can be bypassed in this case, further lowering latency and reducing energy consumption.

As an example of the benefit of a vector extract instructions, T0 would take 50 cycles to reduce a 32 element vector to a single value using memory loads and stores. Using the vector extract instruction and the intra-lane memory crossbar bypasses, the time for a reduction drops to 29 cycles. Without the intra-lane bypass paths, the reduction would take an extra 6 cycles, 4 cycles for extra data transfers and 2 cycles for extra stalls because the adds could no longer chain with the extracts because they would now run at different rates.

Some architectures combine data movement with arithmetic operations by allowing vector arithmetic operations to begin reading one or both source vector registers starting at any element position [Con94, WAC$^+$92, DHM$^+$88]. While straightforward to provide in single lane implementations, this instruction set design does not scale well to multiple lanes. A crossbar interconnect is required between all functional unit pipelines and all lanes, which could increase latency for all vector arithmetic operations.

Some other architectures have gone one step further and provided complete reduction instructions [Buc86]. Reduction instructions cause complications when defining an architecture. To allow implementations flexibility in implementing the order of the operations within the reduction, the results of this instruction need not necessarily be bit-compatible between two implementations. Worse, floating-point reductions could signal different exceptions depending on the operation order. Also, there are many forms of reduction operation, each of which would require a separate reduction instruction.

While a reduction instruction reduces instruction bandwidth, reduction performance with the vector extract instruction is not limited by instruction issue rate but by functional unit and interconnect latencies. A monolithic reduction instruction locks up VAU and inter-lane communication resources for the duration of the instruction's execution, while the primitive vector extract coupled with conventional vector arithmetic instructions allow a compiler to schedule other useful work during reduction operation latencies. This is particularly important when multiple simultaneous reductions are being performed. For example, T0 can perform 12 simultaneous reductions of 32 elements in 141 cycles, or 11.75 cycles per reduction. This is over twice as fast as performing the reductions individually.

10.2 Compress and Expand

Compress instructions select the subset of an input vector marked by a flag vector and pack these together into contiguous elements at the start of a destination vector. Expand instructions perform the reverse operation to unpack a vector, placing source elements into a destination vector at locations marked by bits in a flag register.

Compress and expand operations can be used to implement conditional operations in density-time [HP96, Appendix B.5]. Instead of using masked instructions (Chapter 6), flagged elements can be compressed into a new vector and vector length reduced to lower execution time. After the conditional instructions are completed, an expand operation can update the original vectors. As the number of lanes increases and the number of clocks per vector instruction drops, the advantages of compress/expand over masked instruction execution for this form of conditional execution diminishes. Also, these styles of conditional execution require more interlane communication, either through vector register reductions or via the vector memory.

Compress is more useful for filtering data where the vector length is reduced permanently for subsequent processing, not just temporarily for masking purposes. This form of filtering cannot be implemented efficiently using only masked instructions. This filtering operation occurs in vectorized hash-join [Mar96] and in vectorized garbage collection (Section 11.6.5).

There are several alternative ways of providing compress and expand functionality. One approach is a register-register instruction that reads a source flag register and a source vector data register, and packs the flagged elements of the source into the first elements of the destination vector register. The advantage of this approach is that it does not require access to memory. The disadvantages are that whole data words must move between lanes. Expand instructions can be implemented similarly.

Another approach, used by the Cray architecture, is to write a compressed index vector holding the indices of the elements. The elements can then be retrieved using a vector gather. The advantage of this approach is that only the flag vector, but no data, needs to be transferred between lanes [Smi97]. The disadvantage is that the data may already have been read into vector registers but will have to be read again using a gather to select a subset. The performance impact of this second gather can be reduced by caching the first access to the vector. Expand operations can be mapped to scatters using the compressed indices.

A variant implemented in the Titan graphics mini-supercomputer [DHM+88] is to support compress and expand functionality as part of loads and stores. The Titan compressed loads would compress flagged elements as they were read from memory, and compressed stores would compress elements from a vector register storing the packed vector in memory. Expand variants were also supported. For cases where the packed vector must move between registers and memory, these instructions avoid the inter-lane traffic and excess memory traffic required for register-register compress or compressed index vectors. Adding these compressed memory instructions does not require much additional complexity on top of masked load/stores and register-register compress, but does require additional instruction encodings.

If the VMU crossbar wiring is used to perform inter-lane communication, then these variants all have similar costs. The register-register compress avoids the use of memory address bandwidth to move data and so might have the highest performance for machines with limited address bandwidth. The index vector compress is easy to simulate with a register-register compress.

10.2.1 Compress Vector Length

Code that uses compress for filtering, almost invariably requires that the number of packed elements be known by the scalar unit so that it can update counters and address pointers. Figure 10.3 shows an example compress loop to remove NULL pointers from an array.

This code is exposed to memory latencies by the scalar read of the flag population count. The store of packed values cannot proceed until the vector length, which is dependent on the popcount, is received by the scalar unit. As with reads of count trailing zeros in data-dependent loop exits (Section 6.8 on page 121), this exposes the machine to memory latencies.

Similar to the way in which flag priority instructions create a mask vector of the required vector length, a flag compress instruction can be added to perform the same function for compress loops. Consider the code in Figure 10.4. The compress flags instruction creates a packed flag vector which functions as a mask of length equal to the population count. This allows the vector store to be issued before the population count is received by the scalar unit.

10.3 Vector Register Permutations

There are some data movements which require considerable address bandwidth if performed via the vector memory system but which can be replaced with instructions that rearrange elements held in the vector register file.

One example is the butterfly used in some formulations of the FFT algorithm. This operation can be performed using strided memory accesses but then requires one address per data element. A butterfly permute

```
C code:
size_t packptrs(size_t n, const int* src, int* dest)
{
    size_t i;
    size_t p = 0; /* Number of packed pointers. */
    for (i=0; i<n; i++)
        if (src[i]!=NULL)
            dest[p++] = src[i];
    return p;
}

Assembly code:
packptrs:
    li v0, 0     # Initialize counter.
loop:
    setvlr t0, a0
    ld.v vv1, a1
    sll t1, t0, 3           # Multiply by 8 bytes.
    addu a1, t1             # Bump src pointer.
    u.fnez.f vf16, vv1      # Flag non-nulls.
    popcnt t2, vf16         # Count set bits.
    cmprs.v vv2, vv1, vf16  # Pack non-nulls.
    ctvu t2, vlr            # Set vector length.
    sd.v vv2, a2            # Store to memory.
    addu v0, t2             # Accumulate count.
    sll t3, t2, 3           # Multiply by 8 bytes.
    addu a2, t3             # Bump dest pointer.
    subu a0, t0             # Subtract elements this time.
    bnez a0, loop           # Any more?

    j ra
```

Figure 10.3: Code to remove NULL pointers from an array using a register-register compress instruction.

```
packptrs:
    li v0, 0     # Initialize counter.
loop:
    setvlr t0, a0
    ld.v vv1, a1
    sll t1, t0, 3           # Multiply by 8 bytes.
    addu a1, t1             # Bump src pointer.
    u.fnez.f vf16, vv1      # Flag non-nulls.
    popcnt t2, vf16         # Count set bits.
    cmprs.f vf0, vf16       # Compress flags.
    cmprs.v vv2, vv1, vf16  # Compress data.
    sd.v,m vv2, a2          # Store to memory under mask.
    addu v0, t2             # Accumulate count.
    sll t3, t2, 3           # Multiply by 8 bytes.
    addu a2, t3             # Bump dest pointer.
    subu a0, t0             # Subtract elements this time.
    bnez a0, loop           # Any more?

    j ra
```

Figure 10.4: This version of the `packptrs` routine uses the compress flags instruction to avoid the latency of a round-trip to the scalar unit to reset the vector length of the store.

instruction can perform these permutations entirely within vector registers avoiding the strided memory accesses.

The simplest butterfly permute reads one vector register source and writes one vector register destination. A scalar register operand specifies distance between elements that are to be swapped, and this must be a power of 2. Figure 10.5 shows the operation of this instruction.

Various other forms of vector register permute can be envisaged. Some media processor architectures have included an extensive selection of permutation primitives that operate on the multiple subword elements held in a single wide data word [Han96]. Existing media processing instruction sets have extremely short vectors that are processed in a one cycle, which makes fast permutations simpler to implement, and have no support for strided or indexed memory accesses, which makes element permutations more desirable.

In contrast, vector processors have longer vector registers that are only read and written one element group at a time, which complicates the implementation of some permutes, and have fast support for strided and scatter/gather memory accesses, which diminishes the benefits of permutations. For example, a simple, fast implementation of an arbitrary element permutation would read out the entire source vector into a small multiported RAM, then read back elements in the permuted order. But this would be no faster than simply storing the elements to a cache memory unit-stride then reading the elements back with a gather (assuming the cache had full address bandwidth). The cost of the permutation box is likely better spent on a more generally useful high address-bandwidth cache, such as the VP caches described in Section 9. The VP caches can perform many data reordering operations at high speed while consuming little main memory address and

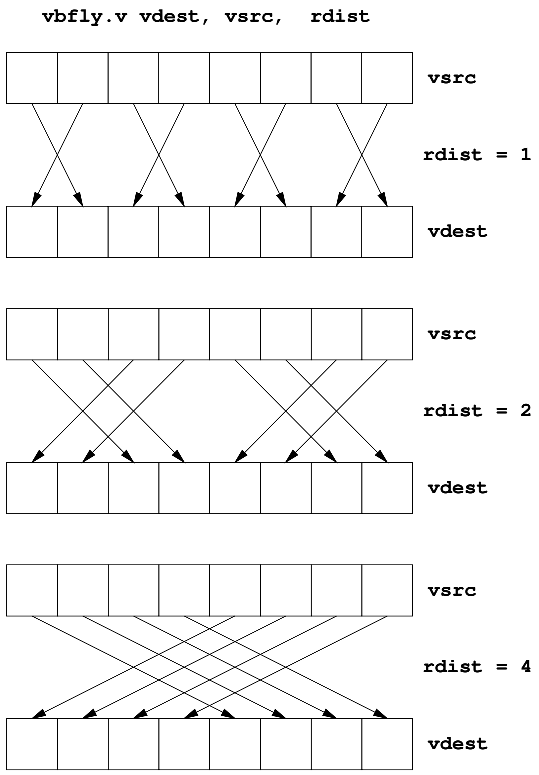

Figure 10.5: Example of a vector butterfly permute instruction. This variant reads all elements from one source register and writes the result to a second vector register. The scalar register operand specifies the distance over which elements should be moved.

data bandwidth.

The butterfly permute is an example which can be implemented at full speed while only reading and writing one element group per cycle. The crossbar configurations required by the butterfly are almost all already available using a simple rotator. For example, adding the butterfly instruction to the T0 design would have only required adding a single additional control wire to the existing datapath multiplexer control network.

Chapter 11

Applications

This chapter describes the vectorization of a range of kernels and applications running on the T0 vector microprocessor. In this chapter, this data is used to evaluate the T0 design. The next chapter extracts statistics from these codes to help guide future vector microprocessor design decisions.

Section 11.1 describes basic linear algebra kernels which are used in many application areas. Section 11.2 describes image processing functions, and includes a comparison with commercial microprocessors with multimedia instruction set extensions. Section 11.3 describes an audio synthesis application, Section 11.4 describes a cryptographic kernel, and Section 11.5 describes two neural net applications. Section 11.6 describes the vectorization of the SPECint95 benchmarks, and shows how vector units can improve cost-performance even for workloads with low levels of vectorization.

11.1 Linear Algebra Kernels

Linear algebra kernels are an important component of many applications, including multimedia and human-machine interface tasks as well as scientific codes. The scientific community has defined a standard set of Basic Linear Algebra Subroutines (BLAS) [LHKK79, DCHH88, DCDH90] for which most vendors supply highly-optimized libraries. All of the BLAS routines are highly vectorizable, with vector lengths determined by the application program calling the library. The following sections examine dot product, matrix-vector multiply, and matrix-matrix multiply, as representative examples of BLAS level 1, 2, and 3 routines respectively.

11.1.1 Dot Product

BLAS level 1 routines [LHKK79] are in the form of 1D vector operations, with $O(N)$ operations performed on length N vectors. One important routine is the vector dot product, $z = \sum_i x_i \cdot y_i$. Dot product can be vectorized by stripmining multiplies and adds, and by accumulating a vector of partial sums. The end of the stripmined loop requires a reduction to yield the final scalar sum. The standard BLAS DOT routine supports input vectors at arbitrary stride.

Figure 11.1 shows the performance of the Spert-II system running dot product with unit-stride arguments. The routine takes 16-bit fixed-point input operands and produces a 32-bit fixed-point result. The performance with and without use of the vector extract instruction is shown. The code implements an optimization for reductions of short vectors, where a scalar lookup table is used to indicate the number of reduction steps required for short vectors. For example, vectors of 17 or greater elements require 5 reduction steps with vector lengths $\{16, 8, 4, 2, 1\}$ while a vector of length 7 requires only three steps with vector lengths $\{4, 2, 1\}$.

Dot product requires two memory loads for each multiply and add, and so performance is memory-bandwidth limited to half of peak arithmetic rate on Spert-II, or 320 MOPS. The inner loop is software pipelined in two stages performing two vector loads and a vector multiply in one stage, and a vector add in the second stage.

The vector extract instruction averages around a 30% speedup on vectors less than 100 elements long, with smaller speedups on longer vectors. Machines with a greater turnaround latency from storing results in memory to loading data from memory would see a much bigger improvement [Oed92].

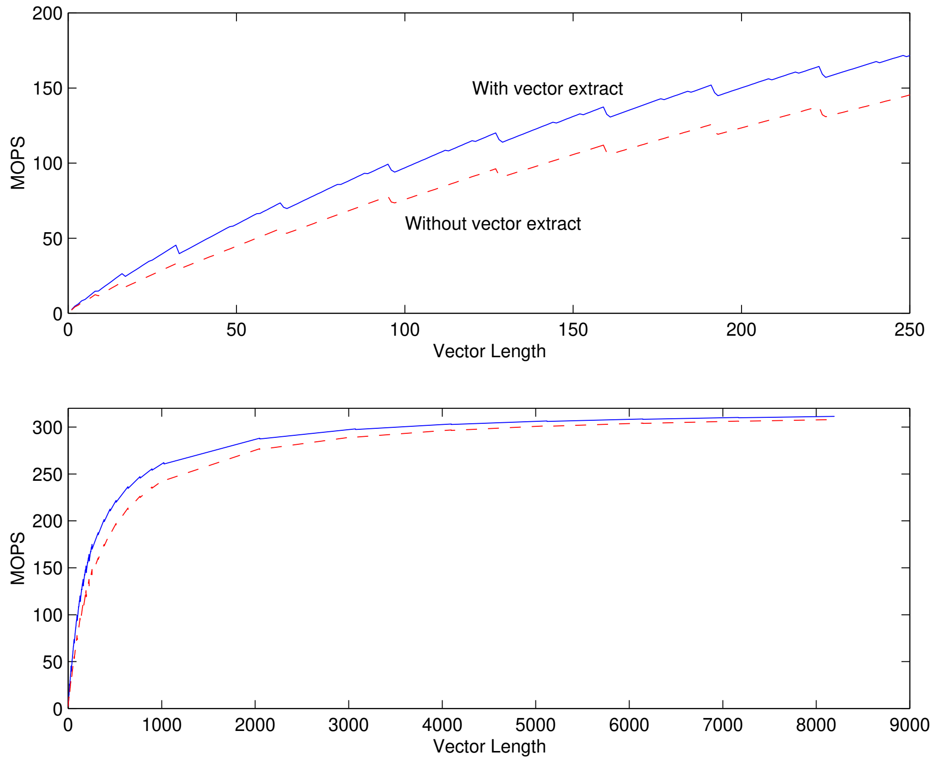

Figure 11.1: Performance of Spert-II on dot product with unit-stride operands. Performance is measured in millions of fixed-point operations per second. The dotted graph shows the performance achieved when the final reduction is performed through memory instead of with the vector extract instruction.

11.1.2 Matrix-Vector Multiply

BLAS level 2 routines [DCHH88] typically perform matrix-vector operations with $O(N^2)$ operations per call. Matrix-vector multiply is one of the more important operations in BLAS 2 and there are two basic forms: $V^T \times M$ and $M \times V$. If each dot product within the matrix-vector multiply is assigned to one VP, then $V^T \times M$ performs a multi-column access to the matrix, while $M \times V$ performs a strided rake access. In both cases, the vector V is accessed as a scalar stream.

Because T0 has limited address bandwidth and no rake cache, an alternative strategy was used for $M \times V$ to allow unit-stride accesses. The vector V is accessed with unit-stride vector accesses, and multiplied with unit-stride vectors from each matrix row. The V values can be reused by operating on 12 matrix rows at a time (limited by the number of vector registers). At the end of each stripe, each of the 12 partial sums is reduced to a single sum before storing into the result vector.

The T0 implementation of $V^T \times M$ has two main loops. The first operates on 128 column by 2 row tiles of the matrix, while the second handles left over columns up to 32 at a time, unrolled to process 4 rows at a time.

Figure 11.2 plots the performance of the $V^T \times M$ and $M \times V$ T0 routines on square matrices. The $V^T \times M$ form achieves up to 480–616 MOPS, and $M \times V$ achieves up to 400–487 MOPS. The fluctuations in performance with different matrix sizes are caused by a combination of stripmining overhead, idle lanes, and aligned versus misaligned unit-stride accesses.

Figure 11.2 also plots the ratio in performance between the $V^T \times M$ versus $M \times V$ forms. This represents the potential speedup for $M \times V$ if a rake cache was added to the T0 design, or alternatively the penalty for not providing full address bandwidth. For large matrices, the $V^T \times M$ is around 20% faster, but for smaller matrices can be over 3 times as fast.

11.1.3 Matrix-Matrix Multiply

BLAS level 3 routines [DCDH90] are primarily matrix-matrix operations that perform $O(N^3)$ arithmetic operations per call. Matrix-matrix multiply is the most important routine within BLAS3 and there are three basic forms: $C = A \times B$ (NN), $C = A \times B^T$ (NT), and $C = A^T \times B$ (TN).[1] Memory bandwidth requirements can be reduced by holding several rows of the destination C matrix in vector registers. The NN and TN cases can both be handled with multi-column accesses to the B matrix and scalar accesses to the A matrix. The NT case performs a strided rake access to the B matrix.

The T0 routines for the NN and TN cases hold a 4 row by 32 column of the C matrix in 4 vector registers, with the inner loop unrolled to perform two 4×32 outer-products in each iteration. The NT routine holds a 12 row by 32 column tile of the C matrix to amortize the overhead of performing slow strided accesses to the B matrix.

Figure 11.3 plots the performance of the NN and NT cases for square matrices. The NN case achieves over 99% of peak on large matrices (635 MOPS), while the NT case is limited to 90% of peak

[1] The TT case is identical to NN with the source matrices swapped.

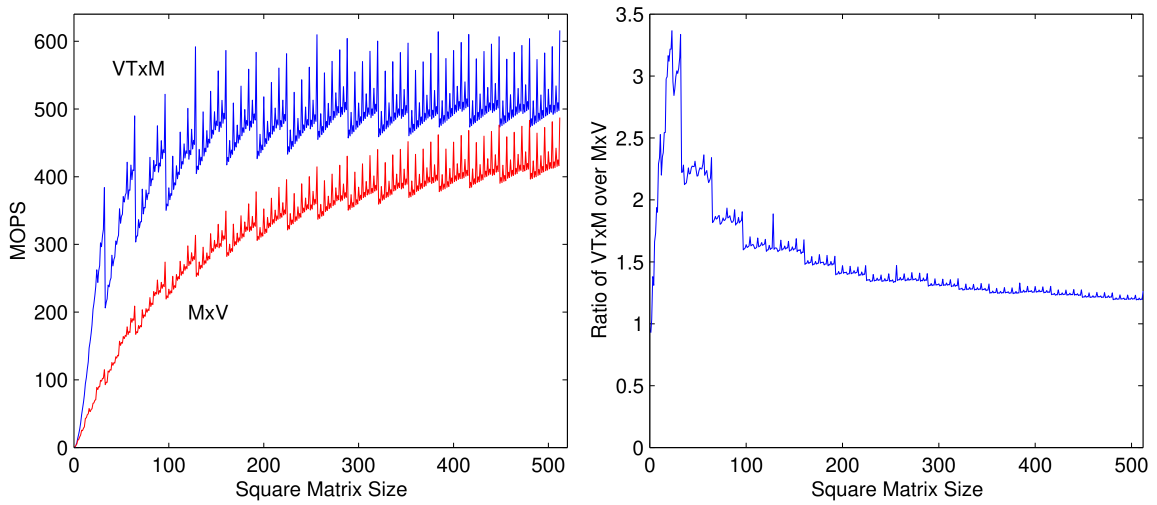

Figure 11.2: Performance of Spert-II matrix-vector multiply routines on square matrices. Input values are 16 bits, and output values are 32 bits. The second graph shows the ratio of the $V^T \times M$ versus $M \times V$ forms.

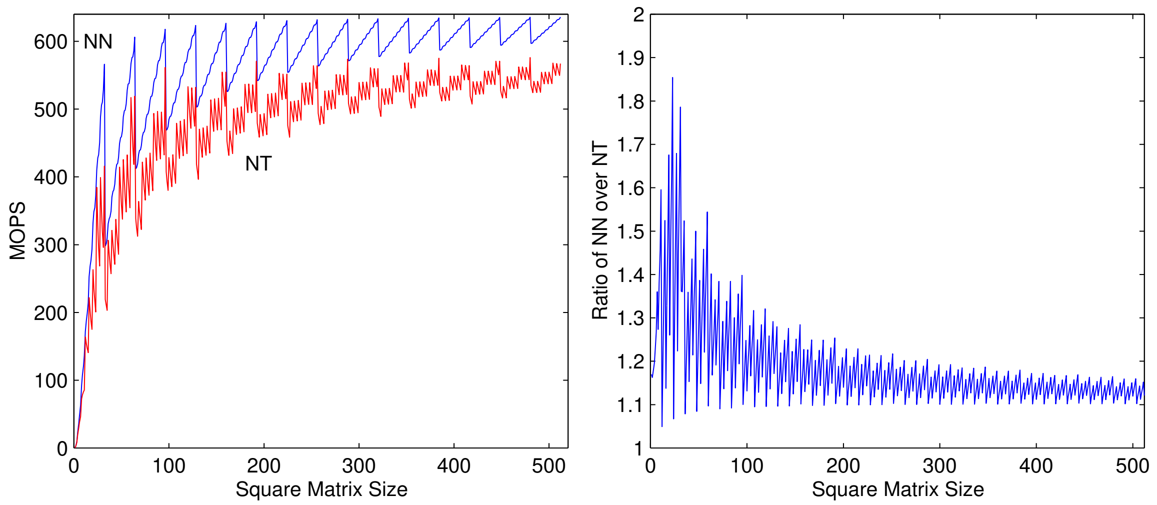

Figure 11.3: Performance of Spert-II matrix-matrix multiply routines on square matrices. Input values are 16 bits, and output values are 32 bits. The left plot shows performance in MOPS for $C = A \times B$ (NN) and $C = A \times B^T$ (NT) multiplies. The second graph shows the ratio of the NN versus NT performance.

(576 MOPS). The figure also plots the ratio of performance of the NN to NT cases to show the potential benefit from adding a rake cache to speed strided accesses for NT. The potential performance improvement is much less than with matrix-vector because matrix-matrix operations can be tiled effectively in vector registers and so are less sensitive to memory performance. The performance on the NT case would also improve with more vector registers to allow greater reuse of the B vector.

11.2 Image Processing

This section reports on the performance of several image processing kernels running on T0, and compares this with the performance of hand-tuned assembler code for commercial DSPs and the multimedia extensions of various commercial microprocessors. When normalized to the same clock rate, the performance results here show that T0 using single instruction issue plus vectors is considerably faster than either the DSPs or superscalar microprocessors with multimedia extensions. Because of its outdated fabrication technology, T0 has the lowest clock rate among the processors compared, yet the per cycle advantage is large enough that in several cases T0 has greater absolute performance than more modern microprocessors on these multimedia tasks.

11.2.1 Convolution

A common operation in image processing is convolution with a small kernel. The performance of a 3×3 box filter routine for T0 is shown in Figure 11.4. The routine takes as input an image stored as unsigned 8-bit pixels, calculates the convolution with an arbitrary signed 16-bit kernel, and writes the scaled, rounded, and saturated result as a signed 8-bit image. The kernel values are read into into nine scalar registers. The source image is read with a neighbor access pattern to align the three pixels on each row with each VP. Each source vector is reused three times by three output pixels down each column. The routine writes output pixels in a multi-column access pattern. Each pixel requires nine multiplications and eight additions. The T0 implementation performs one further operation to give accurate scaling, rounding, and saturation from a 32-bit internal accumulator to the 8-bit output format. Table 11.1 compares T0 performance against that of numbers published for other systems.

11.2.2 Compositing

Compositing (or alpha blending) is a common image processing operation where two images are merged according to values in a third alpha image. The alpha pixels specify how the two images are to blended at that pixel. For monochrome images with 8-bit pixels, the blending function can be written as:

$$Z_{ij} = X_{ij} + \frac{\alpha_{ij}}{255}(Y_{ij} - X_{ij})$$

where X_{ij} and Y_{ij} are the two source image pixel arrays, α_{ij} is the alpha array, and Z_{ij} is the blended output image.

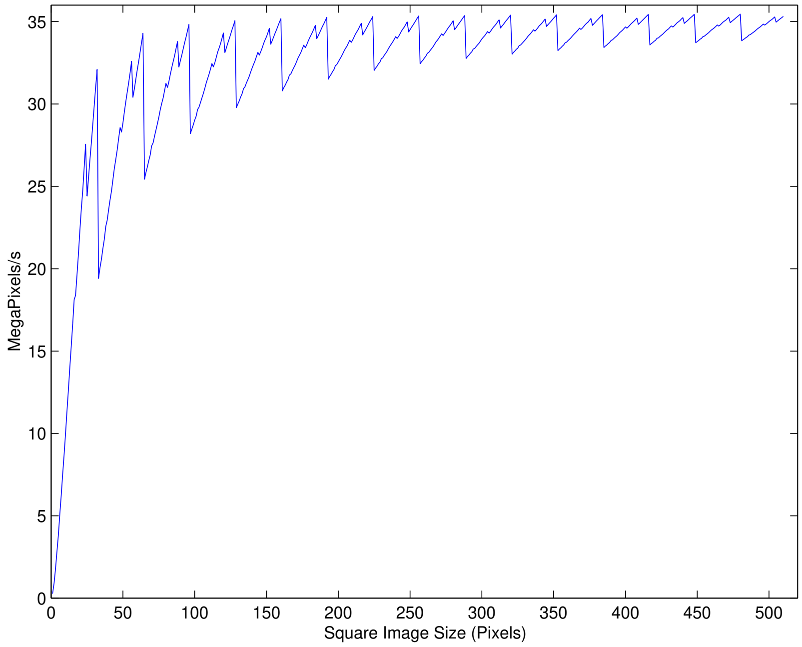

Figure 11.4: Performance of Spert-II 3×3 image filter routine on square images.

System	Clock Rate (MHz)	Performance (MegaPixels/s)	Performance (Cycles/Pixel)
Spert-II	40	35.4	1.13
Ultra-1/170 VIS (in cache) [Ric96]	167	27.0	6.19
HP PA-8000 MAX-2 (in cache) [Lee97]	180	64.3	2.80
TMS320C80 (1 ADSP) [Tex97]	50	5.9	8.50
TMS320C80 (4 ADSPs) [Tex97]	50	*23.6	*2.13
TMS320C82 (1 ADSPs) [Gol96]	50	7.7	6.50
TMS320C82 (2 ADSPs) [Gol96]	50	*15.4	*3.25

Table 11.1: Performance of Spert-II 3×3 image filter compared with other systems. Both absolute performance and clock cycles per pixel are shown. *The numbers for multiple processors on the TMS320C8x DSPs have been obtained by scaling the numbers for one processor.

Figure 11.5 plots the performance of Spert-II on two forms of compositing, and Table 11.2 compares the performance of Spert-II against other systems. The first is for a single monochrome channel. The VIS version of the code [Ric96] uses a shift to divide alpha by 256, not 255, which causes errors in the compositing process. The T0 routine performs the divide using a reciprocal multiply, rounding correction, and shift to ensure bit accuracy and performs at 0.44 cycles/pixel. The T0 performance could be improved to 0.41 cycles/pixel with an extra iteration of unrolling, and to 0.38 cycles per pixel if the same alpha approximation was used as in the VIS code. The VIS version requires over 5.8 times as many cycles per pixel on equivalent code. The best T0 performance is obtained with images that start on 16-bit boundaries; performance drops to 0.5 cycles per pixel with non-16-bit aligned images, regardless of alpha approximation or loop unrolling.

The second form of compositing takes a color image held in a four-byte pixel, with one byte each for alpha, red, green, and blue values, and alpha blends this into a two-byte pixel with 5 bits for each of red, green, blue. The Intel P55C Pentium MMX version of this routine [Cor96d] requires more than 20 times as many cycles per pixel.

11.2.3 Color Space Conversion

Color space conversions are used in image compression algorithms and in other forms of image processing. The most common color space conversions are between RGB and YUV representations. Table 11.3 compares the performance of T0 and other systems on color space conversions. The performance numbers

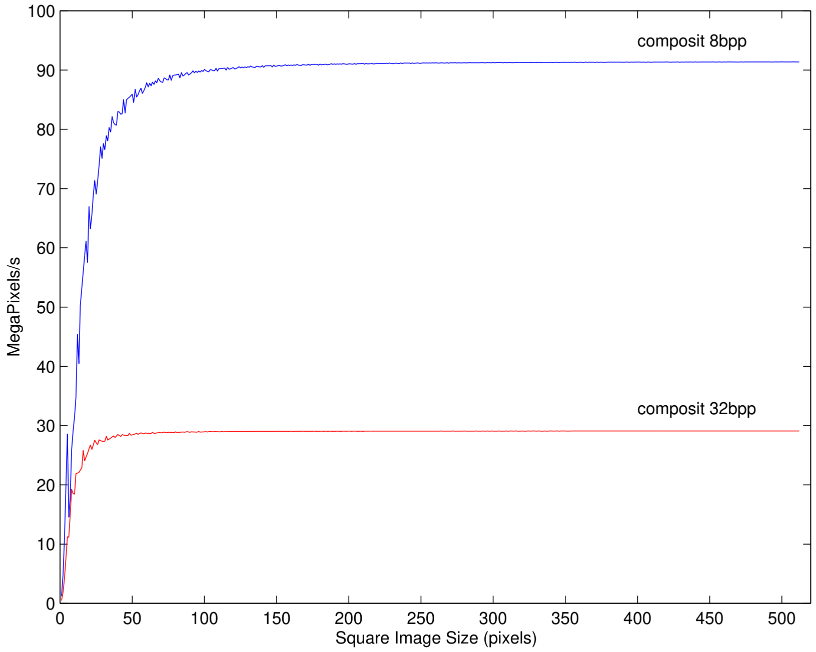

Figure 11.5: Performance of Spert-II on compositing operations.

System	Clock Rate (MHz)	Performance (MegaPixels/s)	(Cycles/Pixel)
Single 8-bit channel			
Spert-II	40	91.4	0.44
Spert-II (non-16-bit aligned)	40	80.0	0.5
Ultra-1/170 VIS (α approx., in cache) [Ric96]	167	75.0	2.22
Single 8-bit channel (estimated times)			
Spert-II (with unrolling)	40	97.6	0.41
Spert-II (with α approx.)	40	106.7	0.38
Blend 32-bit packed ARGB pixels into 15-bit RGB pixels			
Spert-II	40	29.10	1.38
P55C Pentium MMX (in cache) [Cor96d]	200	7.14	28.00
Blend 32-bit packed ARGB pixels into 15-bit RGB pixels (estimated times)			
Spert-II (with α approx.)	40	30.48	1.31

Table 11.2: Performance of Spert-II compositing compared to other systems. Both absolute performance and clock cycles per pixel are shown.

System	Clock Rate (MHz)	Performance (MegaPixels/s)	(Cycles/Pixel)
Packed 24-bit RGB to YUV			
Spert-II	40	42.6	0.94
P55C Pentium MMX [Cor96c]	200	25.0	8.00
4:2:0 YUV to packed 24-bit RGB			
Spert-II	40	47.0	0.85
P55C Pentium MMX [Cor96a]	200	< 42.1	> 4.75
Ultra-1/170 VIS [Ric96]	167	< 70.3	> 2.38

Table 11.3: Performance of Spert-II color space conversion compared to other systems. Both absolute performance and clock cycles per pixel are shown.

for the UltraSPARC VIS [Ric96] and Pentium MMX [Cor96a] systems are upper bounds, and would be lower in practice.

11.2.4 Discrete Cosine Transform for Image Compression

Many image and video compression standards, including JPEG, MPEG, and H.261/H.263, are based on a two-dimensional 8×8 discrete cosine transform (DCT). The forward DCT transforms image pixels into spatial frequency components. Compression is achieved by quantizing the spatial frequency components so that fewer bits are needed to represent them. Typically, higher frequency components are quantized more coarsely to take advantage of the human visual system's reduced sensitivity to high spatial frequencies. The coefficients are ordered in increasing spatial frequency (using a "zigzag" scan of the 2D coefficient array) so that after quantization the higher order coefficients will contain long runs of zeros which can be encoded

System	Clock Rate	Performance	
	(MHz)	(MegaPixels/s)	(Cycles/Pixel)
Spert-II	40	18.52	2.16
Spert-II (scalar, IJPEG C code)	40	0.65	61.54
Ultra-1/170 (IJPEG C code)	167	5.09	32.74
TMS320C82 (1 ADSP) [Gol96]	50	*12.99	*3.85
TMS320C6201 [Tru97]	200	*56.14	*3.56

Table 11.4: Performance of Spert-II forward DCT plus zigzag ordering compared with other implementations. *Times do not include zigzag ordering.

efficiently [PM93].

Forward DCT

The T0 image compression DCT routine takes 8-bit image pixels and produces zigzag-ordered coefficients ready for quantization. The routine first performs 1D DCTs down the columns, then 1D DCTs across the rows. The LLM 8-point DCT algorithm [LLM89] is used giving results accurate to the IEEE specifications, and bit-identical to the code from the Independent JPEG group. The routine is vectorized across 1D DCTs, so the vector length is eight times the number of DCT blocks and is therefore equal to the image width in pixels. T0 has a vector length of 32, and so performs 4 2D DCTs at a time in each iteration of the stripmined loop.

The routine first performs DCTs down the columns to allow the pixel values to be loaded with eight unit-stride byte loads. The resulting 16-bit column DCT coefficients are then packed two to a 32-bit word, and four indexed word stores are used to store the coefficients transposed in a memory buffer allocated on the stack. The next stage performs the row DCTs, beginning with eight unit-stride halfword loads from the transpose buffer. The final coefficient values are written to the result buffer using eight indexed halfword stores to perform the zigzag ordering. The performance of this routine is shown in Figure 11.6, and compared against other systems in Table 11.4.

Performance is limited by T0's single address port. A breakdown of memory port activity is shown in Table 11.5 for one loop iteration that performs four 8×8 DCTs in parallel. The stores into the tranpose buffer are indexed rakes. For other vector machines with interleaved memory subsystems, the rake index vector can be chosen to avoid any memory bank conflicts. The stores into the coefficient buffer perform a permuted write with significant spatial locality, and so a small write cache could significantly reduce the address bandwidth of these stores. An additional eight vector registers would allow the loads of the zigzag indices to be moved outside of the inner loop to give a 6% speedup. All of the arithmetic instructions are overlapped completely with the memory instructions and occupy the two VAUs for a total of 220 cycles (40% utilization).

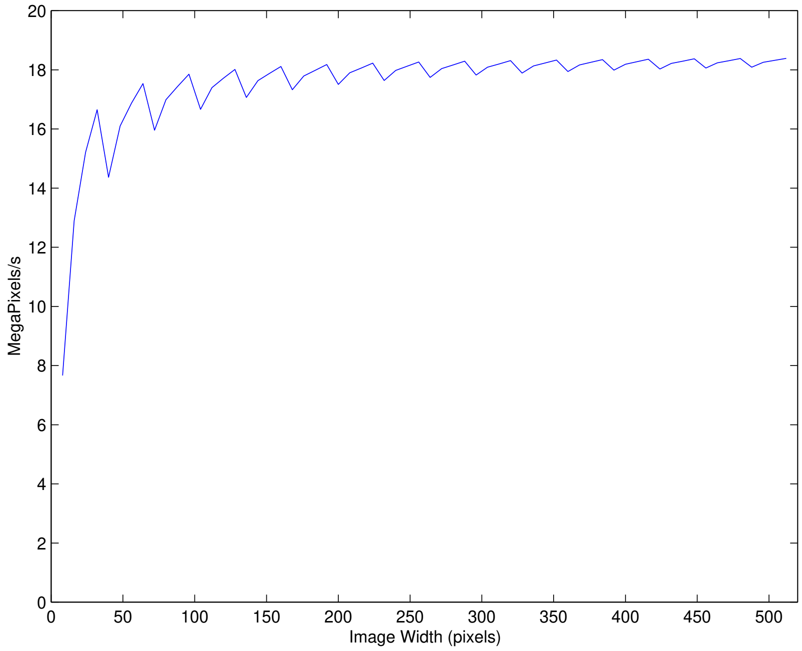

Figure 11.6: Performance of Spert-II on forward 8×8 DCTs. Timings are for one row of blocks, 8 pixels deep.

Function	Access Type	Cycles	Percentage
Load pixels	Unit-stride loads	32	5.8%
Transpose coeffs.	Indexed stores	152	27.5%
Load row coeffs.	Unit-stride loads	32	5.8%
Load zigzag indices	Unit-stride loads	32	5.8%
Store zigzag coeffs.	Indexed stores	304	55.0%
Idle		1	0.2%
Total		553	100.0 %

Table 11.5: Breakdown of memory port activity for forward DCT plus zigzag ordering for four 8×8 blocks on Spert-II.

Inverse DCT

Figure 11.7 plots the performance of a T0 library routine that dequantizes coefficients, performs inverse DCTs, then writes 8-bit image pixels to an image buffer. As with the forward DCT code, this routine uses the LLM algorithm and gives results bit-identical to the Independent JPEG Group code.

The code has two main loops. The first loop reads in coefficients in groups of eight, dequantizes them, then writes them to an internal buffer in a transposed order convenient for the subsequent inverse DCT loop. The inverse DCT loop reads in dequantized coefficients using unit-stride loads, performs the inverse 1D DCT down the columns, then writes intermediate values in transposed manner using indexed stores of two 16-bit values packed into one 32-bit word. The next phase reads in intermediate values using unit-stride loads, performs the inverse 1D DCT across the rows, clips output pixels to representable range, then uses indexed stores to write out 32-bit words each holding four packed 8-bit pixels.

Performance is limited both by the single address port and by a lack of registers. A breakdown of memory port activity is shown in Table 11.7. The two VAUs are busy for 58.6% of the cycles. Neither the VMU nor the VAUs are saturated because dependencies within each stripmine iteration limit performance. With more vector registers, two stripmine iterations could be interleaved, and the memory port could then be saturated during the inverse DCT loop. An additional 8 vector registers would give an estimated 18% speedup. Stores to both the tranpose buffer and the output image buffer are indexed rakes and so could benefit from a rake cache. Also increasing address bandwidth would allow the ALUs to become saturated in the

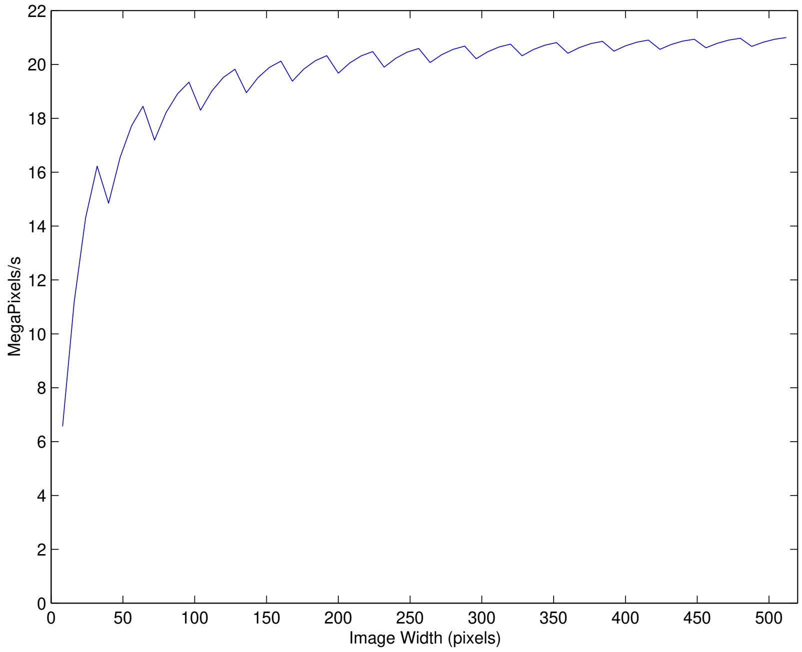

Figure 11.7: Performance of Spert-II on 8 × 8 inverse DCTs including dequantization. Timings are for one row of blocks, 8 pixels deep.

System	Clock (MHz)	Performance (MegaPixels/s)	(Cycles/Pixel)
Spert-II	40	21.42	1.87
HP PA-8000 MAX-2 [Lee97]	180	*78.60	*2.29
Pentium P55C MMX [Cor96b]	200	†*53.33	†*3.75
TMS320C82 (1 ADSP) [Gol96]	50	*8.77	*5.7
Statistical Techniques			
Spert-II (vector, estimated)	40	≈32	≈1.25
Spert-II (scalar, IJPEG C code)	40	≈0.84	≈48
Ultra-1/170 (IJPEG C code)	167	≈12	≈14

Table 11.6: Performance of Spert-II inverse DCT including dequantization compared with other implementations. The performance quoted for the statistical techniques assumes around 7–8 non-zero coefficients per block. *Times do not include dequantization. †Inverse DCT does not meet IEEE 1180 specifications.

Function	Access Type	Cycles	Percentage
Dequantization			
Load quantized coeffs.	Unit-stride loads	32	6.7%
Store dequantized coeffs.	Unit-stride stores	32	6.7%
Idle		48	10.0%
Inverse DCT			
Load column coeffs.	Unit-stride loads	32	6.7%
Load transpose indices	Unit-stride loads	4	0.8%
Transpose coeffs.	Indexed stores	152	31.8%
Load row coeffs.	Unit-stride loads	32	6.7%
Store pixel values	Indexed stores	76	15.9%
Idle		70	14.6%
Total		478	100.0%

Table 11.7: Breakdown of memory port activity for dequantization plus inverse DCT for four 8×8 blocks on Spert-II.

inverse DCT loop, for a combined potential speedup of around 48%.

An attractive alternative approach to vectorizing the iDCT would be to use a statistical method. These takes advantage of the many zero coefficients in a typical compressed image by implementing the entire DCT as a sparse matrix-vector multiply. Each non-zero coefficient would be multiplied by the appropriate 64-element basis vector and summed into the 64 output pixel accumulators. At the end of each block, the pixel values would be clipped and stored into the output image buffer. For T0, each non-zero coefficient would require 8 cycles to perform 64 loads, 64 multiplies, and 64 adds. An additional 28 cycles would be required to clip the output values, and store the output pixels into an image buffer. In the `ijpeg` benchmark described later (Section 11.6.4), there are an average of 7.4 non-zero coefficients per block. This would imply an average of around 80 cycles per inverse DCT for the vector computation without increasing address bandwidth or the number of vector registers. The approach would bring additional time savings in other parts of the code, as it also avoids the reconstruction of runs of zero coefficients. A disadvantage is that this would require merging of several stages of the decoding process, complicating the code structure.

11.3 Audio Synthesis

Hodes [Hod97] has ported a high quality additive audio synthesis algorithm to T0. This application requires the generation of multiple sinusoidal partial oscillators whose control parameters are constantly varying in real-time. Vectorization is across the partials, with vector length given by the number of partials. Increases in performance allow more partials to be generated in real-time, and hence vector length grows with increasing processor performance. The T0 implementation generates up to 608 individually controllable partials in real-time (16-bit audio with 44.1 kHZ sampling rate).

The T0 implementation uses a recursive technique to calculate outputs in the time domain. Alternatively, oscillators can be generated in the spectral domain by taking an inverse Fast Fourier Transform (FFT) of the desired spectrum. The spectral technique allows noise sources to be added easily and can be more efficient for large numbers of oscillators, but adds more overhead for each audio output channel and has a larger real-time control latency. The spectral technique was not used on T0 because the limited precision arithmetic would not allow sufficiently accurate FFTs, but a floating-point vector microprocessor could also use the spectral approach.

For comparison, the spectral technique running on a 180 MHz R10000 system achieves around 1000 partial oscillators [Hod97]. With a 4.5 times greater clock rate and a more efficient algorithm, the superscalar microprocessor generates only 1.6 times more real-time oscillators than T0.

11.4 IDEA

The IDEA (International Data Encryption Algorithm) block cipher is used in several cryptographic applications [Sch96]. Table 11.8 shows the performance of the T0 implementation for various modes and message lengths, and compares this with compiled C code for a Sun Ultra-1/170 workstation.

For CBC-mode encrypt, the vector code uses the vector unit with vector length 1. This gives a significant speedup (2.9) over straight PGP code compiled for T0 primarily due to the fast 16-bit integer multiplier available in the vector unit. The performance of a 500 MHz Alpha 21164 running compiled C code is around 4 MB/s [NL97].

This is an example of vector code with very low memory bandwidth demands; T0 achieves high performance primarily because of the large array of parallel arithmetic units.

	Spert-II PGP Code	Spert-II Vector code	Ultra-1/170 PGP Code
ECB encrypt/decrypt			
1 block	0.24	0.49	1.95
32 blocks	0.24	11.44	1.96
1024 blocks	0.24	14.04	1.96
CBC encrypt			
1 block	0.24	0.65	1.85
32 blocks	0.24	0.70	1.85
1024 blocks	0.24	0.70	1.84
CBC decrypt			
1 block	0.23	0.47	1.91
32 blocks	0.24	10.26	1.91
1024 blocks	0.24	13.01	1.91

Table 11.8: Performance of the IDEA encryption algorithm in MB/s.

11.5 Neural Network Algorithms

This section describes two neural network training applications. The first is a large feed-forward backpropagation neural network used in a speech recognition system, while the second is a small Kohonen self-organizing feature map used in speech coding.

11.5.1 Error Backpropagation Training

Artificial neural networks trained with the error backpropagation training algorithm [W.74] have been successfully applied to many pattern recognition problems. The Spert-II system was originally developed to train large artificial neural networks to produce phoneme estimates for a speech recognizer [WAK+96]. These phoneme networks contain between 100,000 and 2,000,000 weights and are trained using the backpropagation learning algorithm on databases containing millions of acoustic feature vectors.

Figure 11.8 compares the performance of the Spert-II system against that of two commercial workstations, the Sparcstation-20/61 and an IBM RS/6000-590. The Sparcstation-20/61 contains a 60 MHz SuperSPARC processor with a peak speed of 60 MFLOPS. The RS/6000-590 contains a 66.5 MHz POWER2 processor with a peak speed of 266 MFLOPS. The workstation versions of the code are highly tuned, and include manual loop unrolling and register and cache blocking optimizations. The first graph shows performance for forward propagation as used during recognition, while the second graph shows performance for error backpropagation training. The Spert-II board is up to six times faster than the IBM workstation.

A longer description of backpropagation training algorithm implementations on Spert-II is available [ABJ+98].

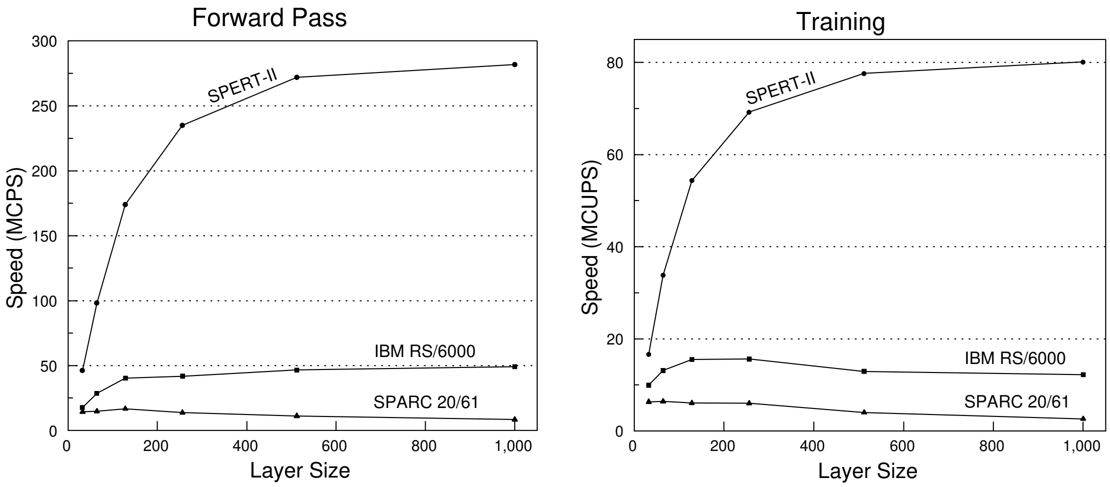

Figure 11.8: Performance evaluation results for on-line backpropagation training of 3 layer networks, with equal numbers of units in the input, hidden, and output layers. The forward pass graph measures performance in millions of connections per second (MCPS), while the training graph shows performance in millions of connection updates per second (MCUPS). The workstations perform all calculations in single precision IEEE floating-point, while Spert-II uses 16 bit fixed-point weights.

11.5.2 Kohonen Self-Organizing Feature Maps

Kohonen self-organizing feature maps (KSOFM) [Koh82] are a form of neural network that can learn to cluster data into topological maps with no external supervision. They have been used in a wide range of applications including image classification and data compression [MS95].

Table 11.9 compares the performance of Spert-II against others reported in the literature for a speech coding task supplied by EPFL, Switzerland [CI94]. This benchmark has 12-dimensional input vectors mapped to a 2-dimensional 10×10 neuron grid. A full description of the Spert-II implementation has been published separately [Asa97]. The Cray T3D is an massively parallel processor array built from 150 MHz Digital Alpha 21064 chips. The Adaptive Solutions CNAPS system [HHK93] is a SIMD processor array, and numbers are presented here both for an EPFL implementation of KSOFM as well as the Adaptive Solutions (ASI) implementation.

Spert-II is faster than these other implementations, especially for the smaller benchmark network (most of the networks used in practice are small [MS95]). This is because the Kohonen net training algorithm is difficult to parallelize efficiently. The algorithm repeatedly finds the neuron whose weight vector has the smallest distance from each input pattern and then updates only a few neighboring neurons' weights. In a parallel processor array, finding the winning neuron requires a global minimum operation and during the weight update phase most processors are idle.

In contrast to the processor arrays, Spert-II has the advantage that the parallel processing elements in the vector unit are tightly coupled and share the same memory space. The minimum operation is performed used a reduction operation within the vector registers, and the whole vector unit can be applied to the weight update.

System	Clock (MHz)	Measured 10×10 benchmark		Estimated peak	
		Number of Processors	(MCUPS)	Number of Processors	(MCUPS)
Sparcstation-20/51	50	1	3.46	-	-
Cray T3D	150	25	4.75	256	435
CNAPS (EPFL)	20	100	4.50	512	23
CNAPS (ASI)	20	-	-	256	54
Spert-II	40	1	45.20	1	213

Table 11.9: Performance numbers for the Kohonen Self-Organizing Feature Map [MS95, ICK96]. Performance is meaured in millions of connection updates per second (MCUPS). The estimated peak numbers assume arbitrarily large networks.

11.6 SPECint95

The SPEC95 benchmarks [Cor95] have become popular both to measure performance of commercial processors and to provide the workload for compiler and architecture research studies. The SPEC95 benchmarks are divided into SPECfp95, which contains floating-point codes, and SPECint95, which contains compute-intensive integer codes. SPECfp95 contains many programs originally developed for vector supercomputers and which are known to be highly vectorizable [Cor95], but the SPECint95 benchmarks are generally regarded as non-vectorizable. In this section, I show that half of the SPECint95 codes, including some described as *non-vectorizable* by the SPEC documentation, can be accelerated significantly with vector execution, though this occasionally requires some minor modifications to the source code.

11.6.1 Methodology

From Amdahl's law [Amd67], we know that vector speedup is given by

$$\frac{T_s}{T_v} = \frac{1}{(1-f) + f/V}$$

where T_s is the execution time of the program running in scalar mode, T_v is the execution time for the program running in vector mode, f is the fraction enhanced by vector execution (the *vector coverage*), and V is the vector speedup for that fraction. More generally, programs have several different portions of their runtime that can be accelerated to differing degrees, which we can express by rewriting Amdahl's law as

$$\frac{T_s}{T_v} = \frac{1}{(1-\sum_i f_i) + \sum_i (f_i/V_i)}$$

where V_i is the vector speedup for fraction f_i of the runtime.

The approach taken in this study has two main steps. The first step is to profile the SPECint95 codes running on conventional superscalar microprocessor-based systems to identify routines that are both time-consuming and vectorizable. Two workstations were used for the profiling measurements: a Sun Ultra-1/170 workstation running Solaris 2.5.1 and an Intel Pentium-II workstation running NetBSD1.3. The Sun machine has a 167 MHz UltraSPARC-I processor [CDd+95], with in-order issue of up to 4 instructions per cycle. The Sun C compiler version 4.0 was used to compile the code using the flags: `-fast -xO4 -xarch=v8plus -dn`. Execution time profiles were obtained using either the `-p` compiler option with `prof`, or with high-resolution timer calls (Solaris `gethrtimer`) embedded in the application. The Pentium-II processor [Gwe97] runs at 300 MHz with out-of-order execution of up to 3 CISC instructions per cycle. The `gcc` version 2.7.2.2 compiler was used to compile the code with optimization flags: `-O3 -m486`. Execution time profiles were obtained using either the `-pg` compiler option with `gprof`, or with high-resolution timer calls (NetBSD `gettimeofday`).

The second step is to port the codes to run on Spert-II. Three versions of each code were obtained by modifying the SPECint95 sources.

The first version contains only changes required to port the code to the Spert-II board. In some cases, the benchmark was reduced in size to fit in the small 8 MB memory. Also, a few modifications were made to port to the limited operating system environment on the board. No changes were made to the main computational routines in this version. This version is called the *original* code in the results.

The second version contains source code optimizations to improve performance when running in scalar mode to allow a fairer comparison against the hand-vectorized code. Some changes were made to work around the limitations in the `gcc` compiler used for T0 and the Pentium-II. In particular, `gcc` can only inline a function if its definition textually precedes the call site, so a few important function definitions were reordered to mimic more sophisticated inlining. As described below, some other modifications were made to each benchmark to improve scalar performance, e.g., explicit loop unrolling in `m88ksim`. Also, any vectorization changes that improved scalar performance were also included. This version is called the *scalar-optimized* code in the results.

The final version is locally restructured to allow vectorization, and vectorizable routines are manually translated into assembly code. The SPECint95 codes were not written with vectorization in mind, and in some cases, have been highly tuned for scalar execution. This artificially limits the potential for vectorization, and so minor changes were made within source code modules provided there were no changes to global data structures or program behavior. This version also incorporates any beneficial scalar optimizations from the scalar-optimized code and is called the *vectorized* code in the results.

The vectorized codes are linked with the T0 standard C library, which includes vectorized versions of the `memcpy`, `memset`, `strlen`, `strcpy`, and `strcmp` routines. The original and scalar-optimized codes are linked with an alternative C library containing hand-tuned scalar assembler versions of these routines.

Execution is profiled both before and after vectorization. These timings give values for the vector speedup, V_i, as well as the fraction of runtime, f_i, for each piece of vectorizable code. For the Spert-II system, the `gcc` cross-compiler version 2.7.0 was used for scalar code. The vectorizable code was manually translated into assembly code, as described below. Timings were obtained using the on-chip cycle counter. The workstation systems are also profiled to give values for f_i.

The following sections describe the modifications made to each benchmark in more detail.

11.6.2 m88ksim

The m88ksim benchmark times a simulator for the Motorola 88100 microprocessor. The standard reference input first loads a memory test program which is too large to fit into Spert-II memory. The simulator execution profile is not highly dependent on the simulated program, so the input was changed to run only the last portion of the benchmark which simulates a Dhrystone executable. This reduced total benchmark running time from 370 to 226 seconds on the Ultra-1/170. The standard ctl.in input was replaced with the following:

```
lo dhry.big
cacheoff
g
sd
q
```

Considerable time is spent in two vectorizable functions: killtime and ckbrkpts. The killtime routine contains two loops to update busy time status for the 32 registers and the 5 functional units in the system. The register update loop was explicitly unrolled for the scalar-optimized version. For the vectorized version, both loops were vectorized with vector lengths of 32 and 5.

The cbrkpts routine contains a loop to check on the 16 possible breakpoints, but this loop exits early whenever a breakpoint condition is met. The loop is vectorized by speculatively executing all 16 loop iterations, with the exit iteration determined subsequently. This speculative execution is straightforward to implement in software because the loop does not write memory and the loop bounds are compile-time constants.

11.6.3 compress

The compress benchmark is a modified version of the Unix compress command based on the LZW compression algorithm [Wel84]. To fit into the board's memory, the benchmark was changed to compress and decompress 2,000,000 characters of text. The text was generated by the SPEC-supplied harness code with arguments "2000000 e 2231". The benchmark was split to give separate timings for compression and decompression, with the two components referred to as comp and decomp in the results below.

The scalar-optimized version of the code was almost completely rewritten from the original to provide a clearer structure and faster code. Some of the major changes included removing references to global variables inside loops, handling exceptional conditions, such as code bit size changes and string table clear, outside of the main control flow, and using an array of structures rather than two arrays for the hashed string table to reduce address calculations and cache misses.

Profiling revealed that significant time was spent packing and unpacking codes from the input and output bitstreams. The original compress routine packs each 9- to 16-bit code into the output bitstream one code at a time. A more efficient approach is to keep a buffer of unpacked codes, and then to pack the entire buffer into the output bitstream in one step. In particular, a single check for 16-bit codes can be used to choose a specialized loop which avoids the bitfield manipulations required for shorter code lengths. This is an important optimization because 16-bit codes are the most common for long files. Similarly, the decompress routine can be accelerated by unpacking multiple codes at a time. The scalar-optimized code includes these pack/unpack optimizations.

The original decompression routine reads out strings from the string table one character at a time in reverse order into a temporary string buffer. The characters in the buffer are then copied in correct order into the output stream. The scalar-optimized version of the code puts the characters into the buffer in the correct order and so can use a `memcpy` to move characters into the output stream.

The vectorized code attains speedups by vectorizing the pack/unpack operation and also the standard `memcpy` routine. When 8 N-bit codes are packed, they occupy N bytes of storage. Vectorization is across these independent N-byte sections of packed input or output codes. A buffer of 512 codes is used, giving a vector length of 64.

The execution time for `compress` is very sensitive to code quality in the inner scalar loops. The compiler's assembler output for this inner loop was hand-optimized to overcome problems in `gcc`'s instruction-scheduling and delay-slot filling for both the scalar-optimized and vectorized codes on T0. In addition, the pack/unpack routines of the scalar-optimized code were hand-scheduled, and the `memcpy` call for the vector code was hand-inlined.

11.6.4 `ijpeg`

The `ijpeg` benchmark repeatedly performs JPEG image compression and decompression while varying various compression parameters. It is the most highly vectorizable of the SPECint95 programs.

A number of changes were made to the original scalar code to port to the Spert-II board. The `MULTIPLIER` type was changed from `int` to `short` in `jmorecfg.h` to match the interface with the T0 vector DCT (Discrete Cosine Transform) routines (Section 11.2.4). The `ijpeg` DCT routines have been designed to work with 16-bit integers and so this change doesn't affect the accuracy or runtime of the scalar routine. The T0 forward DCT vector routine first performs 1D DCTs down columns followed by 1D DCTs across rows, whereas the original SPEC code does rows first. Although mathematically equivalent, rounding errors cause a slight change in the coefficients and hence in test output. The original and scalar-optimized versions were modified to work the same way as the vectorized version so that all codes would produce identical output. This resulted in no change to run-time and less than 0.2% change in compressed file sizes for the test image.

The constant `JPEG_BUFFER_SIZE` was reduced to 1 MB in `spec_main.c` to reduce memory consumption by statically allocated arrays. The timings used the `penguin.ppm` reference input image reduced in resolution by a factor of two in both directions to fit into the reduced memory. The reduced size image does change the scalar execution profile slightly, and halves the vector lengths of some routines, but the runs should still be representative of typical JPEG code.

The benchmark command line arguments were:

```
-image_file penguin.ppm -compression.quality 90
-compression.optimize_coding 0 -compression.smoothing_factor 90
-difference.image 1 -difference.x_stride 10
-difference.y_stride 10 -verbose 1 -GO.findoptcomp
```

The `ijpeg` code has been well tuned for scalar execution but a few minor optimizations were added. The original code performs coefficient quantization using an integer divide, with a quick magnitude check to skip cases where the result will be zero. The quantization tables are changed rarely, and so the division can be replaced by a reciprocal multiply and arithmetic shift right. When packing bits into the output stream, the original code calls the `emit_bits` subroutine twice for non-zero AC coefficients, once for the preceding zeros symbol and again for the non-zero coefficient magnitude. These two bit strings can be combined to reduce the number of calls to `emit_bits`, though two calls must still be made if the total number of bits is greater than 24. A further optimization is possible, similar to that in `compress`, where instead of calling `emit_bits` as each symbol is generated, all symbols for a block are first buffered before making a single call to pack all the symbol bitstrings to the output stream.

The vectorized version replaces many routines with vector code. The forward and inverse 8×8 2D DCTs are vectorized across the constituent 1D 8-point DCTs. The forward DCTs are performed with vector length 512, corresponding to the test image width. The inverse DCTs are performed in small groups of 1 or 2 blocks, limiting vector lengths to 8 or 16. The vector length for inverse DCT could also have been increased up to the image width by buffering a scan line of coefficients, but this would have required more than local restructuring of the code. The coefficient quantization and dequantization is also vectorized with vector lengths of 64. The image manipulation routines `rgb_ycc_convert`, `upsample`, and `downsample`, have vector lengths proportional to image width (512), or half of the image width (256) for subsampled components.

The `encode_MCU` routine has two incarnations, one which generates Huffman-coded bitstrings (`encode_one_block`) and one which just calculates the symbol frequencies for optimizing Huffman coding (`htest_one_block`). These routines need to scan the 63 AC forward DCT coefficients to find runs of zeros. A vector routine was written to scan the coefficients and return a packed bit vector. This packed bit vector can then be scanned quickly with scalar code to find the symbols for Huffman coding. The Huffman code lookup can also be vectorized, but it was found that the vector version ran at about the same speed as the scalar version on T0. The scalar code can use an early exit loop to find the coefficient magnitude, but the vector version must use multiple conditional instructions to determine the magnitudes and these consume more than half the vector lookup runtime.

11.6.5 li

The li benchmark is a simple Lisp interpreter. The original SPECint95 benchmark can be run unchanged on the Spert-II board. Profiling revealed that a considerable fraction of time is spent in the mark and sweep garbage collector. The only change made for the scalar-optimized code was to reorder routines in the garbage collector code to allow gcc to inline them.

The mark phase of the li garbage collector was vectorized by converting the depth-first traversal of the live storage nodes into a breadth-first traversal [AB89]. The mark routine dynamically manages a queue of pointers on the run-time stack. The queue is initialized with the root live pointers at the start of garbage collection. A stripmined loop reads pointers from the queue and overwrites it with pointers for the next level of the search tree. Any pointers that should not be followed are set to NULL. Most vector machines provide some form of vector compress instruction to pack together selected elements of a vector [HP96]. T0 lacks a vector compress instruction and so a scalar loop is used to compress out NULL pointers from the queue. The mark phase is finished when the queue is empty after the scalar compress operation. The average vector length is 80 during this phase.

The li interpreter allocates node storage in segments, each of which contains a contiguous array of nodes. The sweep phase was vectorized by partitioning the nodes within a segment into sections, then vectorizing over these sections. The vector code chains together the unmarked nodes within each section into a local free list. A scalar loop then stitches the multiple free lists together to give a single free list, which has the same node ordering as the original scalar code. The vector length is determined by the number of nodes allocated at one time within each segment — 1000 for the li benchmark.

11.6.6 Other SPECint95 benchmarks

The remaining SPECint95 benchmarks could not be appreciably vectorized with small local changes. Although the perl benchmark is dominated by complex control flow, it spends a few percent of its runtime in vectorizable standard C library functions [Cor95] and so might experience a small speedup from vectorization. perl could not be ported to the Spert-II board due to its extensive use of operating system features.

The go benchmark spends significant time in loops but makes extensive use of linked list data structures, which hamper vectorization. It is possible that a substantial rewrite could make use of vectors.

The gcc and vortex benchmarks are dominated by complicated control structures with few loops and it is unlikely that even extensive restructuring could uncover significant levels of vector parallelism.

11.6.7 SPECint95 Vectorization Results

Table 11.10 shows the extent of the code modifications in terms of lines of code in the original sources that were affected. Except for compress, only a small fraction of the source was modified for vectorization. The compress code is a small kernel, and was extensively rewritten for the scalar-optimized version. Although it required the largest percentage of modifications for vectorization, the absolute number

Benchmark	Original (LOC)	Scalar-Optimized Changes (LOC)		Vector Changes (LOC)	
m88ksim	19,915	4	(0.0%)	65	(0.3%)
compress	1,169	1,071	(100.0%)	*80	(7.5%)
ijpeg	31,211	84	(0.3%)	778	(2.5%)
li	7,597	0	(0.0%)	198	(2.6%)

Table 11.10: This table shows the number of lines of code (LOC) in the original benchmark, together with the number of lines of original code changed for the scalar-optimized and vectorized versions. *compress vectorization changes are counted relative to the scalar-optimized version.

Benchmark	Original (s)	Scalar-Optimized (s)	Vector (s)
Ultra-1/170, 167 MHz			
m88ksim	226.5	241.0	N/A
compress	1.5	1.3	N/A
ijpeg	37.6	36.1	N/A
li	403.8	412.1	N/A
Pentium-II, 300 MHz			
m88ksim	133.7	140.2	N/A
compress	1.3	1.1	N/A
ijpeg	24.8	24.3	N/A
li	209.4	191.5	N/A
Spert-II, 40 MHz			
m88ksim	1853.1	1831.5	1300.2
compress	7.4	3.8	3.23
ijpeg	271.5	269.7	63.8
li	2493.0	2279.1	1871.8

Table 11.11: Execution time in seconds for the three versions of each benchmark on each platform. The compress times are the sum of comp and decomp.

of lines changed is small.

Table 11.11 presents the overall timings for the codes on each platform. The scalar optimizations were tuned for Spert-II, and although generally these optimizations improve performance on the other platforms, in a few cases performance is reduced. In particular, the explicit loop unrolling in m88ksim *reduces* performance on both the UltraSPARC and the Pentium-II. The li scalar-optimized code on the UltraSPARC was also slightly (2%) slower than the original code, even though the only modification was to reorder some routine definitions. The scalar-optimized code for compress is twice as fast as the original SPEC source code on T0, but only 15–18% faster on the workstation systems. This difference in speedup might be due to the better compiler and superscalar issue on the Sun system and the out-of-order superscalar scheduling on the Pentium-II, which might give better performance on the original code compared with T0.

Table 11.12 gives a timing breakdown for the benchmarks. The scalar-optimized version was used

Application Routine	Ultra-1/170 167 MHz (s)	(%)	Pentium-II 300 MHz (s)	(%)	T0 Scalar 40 MHz (s)	(%)	T0 Vector 40 MHz (s)	(%)	T0 Vector Speedup
`m88ksim`									
`killtime`	66	(26)	34.8	(26.2)	457.3	(25.0)	57.6	(4.4)	7.93
`ckbrkpts`	41	(16)	15.6	(11.7)	230.1	(12.6)	101.1	(7.8)	2.28
Other	143	(57)	82.5	(62.0)	1144.1	(62.5)	1142.3	(87.8)	1.00
Total	250	(100)	132.8	(100.0)	1831.5	(100.0)	1301.0	(100.0)	1.41
`comp`									
`packcodes`	0.11	(11.0)	0.04	(4.6)	0.22	(9.6)	0.03	(1.2)	8.58
Other	0.85	(89.0)	0.81	(95.4)	2.05	(90.4)	2.12	(98.8)	0.97
Total	0.96	(100.0)	0.85	(100.0)	2.27	(100.0)	2.15	(100.0)	1.07
`decomp`									
`unpackcodes`	0.04	(12.3)	0.04	(10.5)	0.24	(16.3)	0.03	(2.4)	9.46
Other	0.30	(88.6)	0.18	(89.5)	1.23	(83.7)	1.06	(97.6)	1.16
Total	0.34	(100.0)	0.22	(100.0)	1.48	(100.0)	1.08	(100.0)	1.37
`ijpeg`									
`rgbyccconvert`	4.1	(10.4)	3.17	(13.1)	23.1	(8.7)	2.2	(3.6)	10.7
`downsample`	7.1	(18.0)	4.47	(18.5)	62.1	(23.4)	6.5	(11.0)	9.5
`forward_DCT`	12.7	(32.2)	5.95	(24.6)	73.6	(27.8)	5.1	(8.6)	14.3
`encode_MCU`	4.4	(11.2)	3.26	(13.5)	26.3	(9.9)	18.8	(31.6)	1.4
`inverse_DCT`	5.3	(13.5)	3.71	(15.3)	42.4	(16.0)	4.6	(7.7)	9.2
`upsample`	2.2	(5.6)	1.44	(6.0)	14.6	(5.5)	1.1	(1.8)	13.3
Other	3.6	(9.1)	2.20	(9.1)	22.7	(8.6)	21.2	(35.6)	1.1
Total	39.4	(100.0)	24.20	(100.0)	264.8	(100.0)	59.5	(100.0)	4.5
`li`									
`mark`	96.2	(23.3)	57.0	(28.8)	426.6	(18.8)	102.3	(5.5)	4.17
`sweep`	29.2	(7.1)	20.4	(10.3)	129.9	(5.7)	45.6	(2.5)	2.85
Other	286.7	(69.6)	120.3	(60.8)	1712.0	(75.5)	1708.0	(92.0)	1.00
Total	412.1	(100.0)	197.7	(100.0)	2268.4	(100.0)	1855.9	(100.0)	1.22

Table 11.12: Breakdown of runtime for scalar and vector SPECint95 applications. The `compress` benchmark is split into `comp` and `decomp` components. The scalar-optimized profile is given for the scalar systems, except for `m88ksim` on Ultra and Pentium-II, where the faster original version is profiled.

for the scalar profiles, except for `m88ksim` on the workstation systems where the faster original version was used. These profiles were obtained either with statistical profiling or with embedded calls to high-resolution timers, both of which are intrusive methods that slightly affect the timing results compared to Table 11.11.

The vector speedup obtained on whole benchmarks varies widely, with `ijpeg` having the greatest speedup (4.5) confirming the suitability of vectors for these types of multimedia applications. The lowest speedup is for `comp` (1.07) whose runtime is dominated by scalar hash table operations. Using a combined figure of 1.16 for `compress`, and assuming no speedup for the non-vectorized codes, the geometric mean vector speedup for T0 across all eight SPECint95 benchmarks is 1.32.

Individual functions exhibit much higher speedups, with several of the `ijpeg` routines running over 10 times faster in vector mode. The vector unit therefore improves performance by accelerating limited portions of the execution time by a large amount. For `decomp` and `ijpeg`, there is also a more moderate speedup over the whole runtime, but this is due to the vectorized standard C library functions which are not timed individually.

Comparing the profiles of scalar code, we see that all the platforms are broadly similar in their distribution of runtime amongst the vectorizable and non-vectorizable portions of code. The biggest differences occur for the Pentium-II on `comp`, where it spends comparatively less time in vectorizable code, and on `li`, where it spends comparatively more. The workstations are superscalar designs whereas T0 is single-issue, suggesting that speedups from instruction-level parallelism are distributed throughout the benchmark rather than being concentrated in the regions that were vectorized.

11.6.8 Discussion

The above results present clear evidence that significant portions of the SPECint95 benchmark suite can be executed with vector instructions after some minor modifications to the source code, but the magnitude of the vector speedup is affected by several factors. The discussion in this section estimates the impact of these factors, which include fabrication technology, memory hierarchy, vector architecture, and code quality, and also shows how vector speedup can be profitably combined with superscalar speedup.

T0 is fabricated in an older $1.0\,\mu$m two-metal CMOS technology which results in a larger die area and lower clock rate compared with the newer technology used to fabricate the superscalar processors. Because it does not interfere with core processor functions, the addition of a vector unit should have no impact on processor cycle time, and so we can assume that vector microprocessors will have the same clock cycle as vectorless microprocessors when implemented in the same technology. Table 11.13 shows the performance of the benchmarks with all machines scaled to the same clock rate and with speedups measured relative to T0 running scalar-optimized code.

T0 has an unconventional memory hierarchy, with a small 1 KB direct-mapped primary instruction cache (I-cache) but no data cache (D-cache), and a flat three cycle latency main memory system. This memory hierarchy should produce *lower* vector speedups compared with a more conventional memory system. The small I-cache produces more misses, concentrated outside of vectorizable loops which generate few I-cache misses. The overall effect is to reduce the vectorizable fraction of runtime, f_i, by increasing the time spent executing non-vectorizable scalar code. The lack of a primary D-cache has two main effects. First, all scalar accesses have three cycle latencies rather than the two cycles typical for primary D-caches. The tuned scalar code for the vectorizable loops is mostly able to hide this extra cycle of latency, whereas it will likely slow performance on other code. Second, there are no D-cache misses. While both vector and scalar unit will experience approximately the same number of D-cache misses in a conventional memory hierarchy, most of the vectorizable code has relatively long vectors which should help hide miss latencies to the next level of cache. The SPECint95 codes have relatively small working sets and little execution time is spent in misses from typical sizes of external cache, even on fast processors [CD96].

Benchmark/Routine	T0 Scalar	Ultra-1/170	Pentium-II	T0 Vector
`m88ksim`				
`killtime`	1.00	1.66	1.75	7.93
`ckbrkpts`	1.00	1.35	1.97	2.28
Other	1.00	1.92	1.85	1.00
Total	1.00	1.76	1.84	1.41
`comp`				
`packcodes`	1.00	0.48	0.75	8.58
Other	1.00	0.58	0.34	0.97
Total	1.00	0.57	0.36	1.07
`decomp`				
`unpackcodes`	1.00	1.45	0.84	9.46
Other	1.00	0.99	0.91	1.16
Total	1.00	1.04	0.90	1.37
`ijpeg`				
`rgb_ycc_convert`	1.00	1.34	0.97	10.7
`downsample`	1.00	2.10	1.85	9.5
`forward_DCT`	1.00	1.40	1.65	14.3
`encode_MCU`	1.00	1.43	1.08	1.4
`inverse_DCT`	1.00	1.92	1.52	9.2
`upsample`	1.00	1.61	1.35	13.3
Other	1.00	1.53	1.37	1.1
Total	1.00	1.61	1.46	4.5
`li`				
`mark`	1.00	1.06	1.00	4.17
`sweep`	1.00	1.07	0.85	2.85
Other	1.00	1.43	1.90	1.00
Total	1.00	1.32	1.53	1.22

Table 11.13: Relative speed of each component normalized to same clock rate on each platform. The speedups are measured relative to the scalar-optimized code running on T0. The workstation timings for `m88ksim` are for the original code which is faster for those systems.

The vector speedup is also affected by the design of the vector unit. T0 has a simple vector unit that lacks some common vector instructions. For example, the addition of a vector compress instruction would reduce the execution time for the `li` `mark` routine from 102.3 seconds to an estimated 47 seconds, increasing `mark` speedup to a factor of 9. T0 also lacks masked vector memory operations which would further reduce `mark` runtime to around 33 seconds, improving vector speedup to 13. Another example is the `sweep` routine, where masked stores would decrease runtime to around 27 seconds, improving vector speedup to 4.9. The `encode_MCU` routine performance could be increased with the addition of a vector "count leading zeros" instruction to determine AC coefficient magnitudes. These vector unit optimizations require little additional die area, and no increases in main memory bandwidth. Several of the routines are limited by vector strided and indexed memory instructions that transfer only one element per cycle on T0. Additional address ports would improve throughput considerably, but at some additional cost in the memory system. An alternative approach to improve cost/performance would be to reduce the number of parallel lanes in the vector unit; this would reduce area but have limited performance impact for those routines limited by address bandwidth.

The relative timings are also affected by code quality. The vector routines were manually translated into assembly code, likely resulting in higher code quality than automatic compilation and hence greater vector speedup. The `ijpeg` DCT routines were the most difficult to schedule and would probably show the largest improvement over compiled code. But for the `comp`, `decomp`, and `li` benchmarks, and for the `rgb_ycc_convert`, `upsample`, and `downsample` portions of `ijpeg`, performance is primarily limited by the single address port and so there is little opportunity for aggressive assembler hand-tuning to improve performance. The vectorization of the `m88ksim` benchmark is trivial with no stripmine code and little choice of alternative instruction schedules, and so performance should be very close to that with a vectorizing compiler. The speedup provided by the vectorized standard C library routines requires no compiler support beyond function inlining.

Because of the effort involved, the manual vectorization strategy is limited to a few key loops in each benchmark. Compared with an automatically vectorizing compiler, this limits vector coverage and hence reduces vector speedup. Another distortion from hand-tuning is that only the scalar routines compared against vectorized routines were tuned, which reduces vector speedup compared with more careful tuning or higher quality compilation of the non-vectorizable scalar code.

Almost all of these factors act to reduce the vector speedup for T0 compared to a future vector microprocessor with a conventional memory hierarchy and an automatically vectorizing compiler. The exception is the quality of the vector code for each routine, but this only changes the vector speedup, V_i, not the vectorizable fraction, f_i. Because the vector speedups are high and the fraction vectorized is low, the resulting overall speedup is not very sensitive to the values for V_i. As a pessimistic example, consider if the vector speedups of the `ijpeg` DCT routines were reduced by a factor of 2, and other speedups were reduced by a factor of 1.5, except for the standard library routines and `encode_MCU` which remain unchanged. In this case, the resulting geometric mean speedup on SPECint95 would only drop to 1.26.

11.6.9 Combining Vector and Superscalar Speedups

The results in Table 11.13 show that superscalar processors speed up both vectorizable and non-vectorizable code by approximately the same amount, whereas vector units only speed up the smaller vectorizable fraction but to a much greater degree. A particularly attractive design alternative is to combine vector and superscalar techniques, giving a combined speedup of

$$\frac{T_s}{T_{ss+v}} = \frac{1}{(1-f)/S + f/V}$$

where S is the superscalar speedup, and T_{ss+v} is the execution time of the combined superscalar and vector processor.

As an example, consider the MIPS R10000 [Yea96] which is a quad-issue out-of-order superscalar microprocessor. The R10000 achieves approximately 1.7 times speedup over the R5000 on SPECint95 when running at the same clock rate with the same external cache (Table 2.1 on page 18). Based on the previous results, we can estimate that a vector unit would achieve a speedup of around 1.32 over the R5000, by speeding up 28% of the execution time by a factor of 8. From the above equation, we can predict that adding a vector unit to the R10000 would increase its speedup to 2.18, or an additional 1.28 times greater than the superscalar speedup alone. Although the R10000 has the same primary cache configuration as the R5000, the multiple functional units and complex instruction issue management logic inflate the die to 3.4 times the area of the R5000. The full T0 vector unit would only add 7% area to the R10000 die, and could use the existing 128-bit primary and secondary cache interfaces.

These estimates suggest that a vector unit can provide improvements in cost/performance *even for codes with low levels of vectorization*. This is because the vector unit is compact, yet achieves large speedups on the data parallel portions of the code.

Chapter 12

Application Statistics

This chapter presents various statistics from the applications described in the previous chapter. Table 12.1 lists the application kernels measured in this chapter.

Section 12.1 describes the application vector lengths, with typical lengths in the range of 10s to 100s of elements. Section 12.2 presents measurements of vector register usage which show that 32 vector registers would be a reasonable number for a vector microprocessor. Section 12.3 shows how arithmetic precision varies across these kernels, suggesting that variable-width virtual processors should improve cost/performance. Section 12.4 presents the ratio of arithmetic to memory instructions in these codes, showing that these often have much larger compute to memory ratios than those previously measured for scientific vector codes. Adding more arithmetic pipelines in the form of more VAUs is less expensive than adding more memory bandwidth to support more VMUs, and these results suggest that a greater number of VAUs could benefit some codes. Section 12.5 discusses the mix of memory access patterns. The results show that a large fraction of strided and indexed accesses are in the form of rakes and hence can use a rake cache to reduce address bandwidth. Finally, Section 12.6 shows that most of the codes are insensitive to memory latency, and that where memory latency is exposed, it is usually due to scalar operand streams. These streams have predictable access patterns and hence are amenable to scalar decoupling as a technique for latency tolerance.

12.1 Vector Length

Measuring application vector length is important because it indicates the amount of data parallelism present in a program. Table 12.2 tabulates the natural vector lengths observed in the various kernels. These results were either obtained with hand analysis or by instrumenting source code to collect run-time application vector lengths. Where applications contain reductions, a sum ($\sum$) symbol is included. Reduction operations have logarithmically decreasing vector lengths depending on machine vector register length. For these codes, most natural vector lengths are in the range of 10s to 100s of elements. Vector lengths over several thousand are rare.

Application/Kernel	Description
	Linear Algebra
`dot`	Dot product
`vtmmul`	Vector transpose $\times$ matrix multiply
`mvmul:rake`	Matrix $\times$ vector multiply, using rake
`mvmul:unit`	Matrix $\times$ vector multiply, using unit-stride
`mmmul`	Matrix $\times$ matrix multiply
`mmtmul`	Matrix $\times$ matrix transpose multiply
	Image Processing
`box3x3`	Convolution with general 3×3 box filter
`composit8bpp`	Alpha blending two 8-bit per pixel images
`composit32bpp`	Alpha blending 32-bit RGBA image into 16-bit RGB image
	Audio Synthesis
`vaudio`	Additive Audio Synthesis
	Cryptography
`idea:cbcdec`	IDEA CBC-mode decryption
	Neural Net
`quicknet:forward`	Forward pass of speech recognition network
`quicknet:train`	Training speech recognition network
`ksofm:forward`	Forward pass of Kohonen Self-Organizing Feature Map
`ksofm:update`	Weight update of Kohonen Self-Organizing Feature Map
	SPECint95
`m88ksim:killtime`	`killtime` function in `m88ksim`
`m88ksim:ckbrkpts`	`ckbrkpts` routine in `m88ksim`
`comp:pack`	`pack` code during compression
`decomp:unpack`	`unpack` code during decompression
`decomp:copy`	string copy in decompression
`li:mark`	mark phase of garbage collect
`li:sweep`	sweep phase of garbage collect
`ijpeg:rgbycc`	`ijpeg` RGB to YCrCb color conversion
`ijpeg:downsample`	`ijpeg` chrominance subsampling
`ijpeg:fdct`	`ijpeg` forward DCT
`ijpeg:mcu`	`ijpeg` count zero AC coefficients
`ijpeg:idct`	`ijpeg` inverse DCT
`ijpeg:upsample`	`ijpeg` upsample and color convert

Table 12.1: Description of kernels and applications.

Application	Length
Linear Algebra	
`dot`	$N + \sum$
`vtmmul`	N
`mvmul:rake`	M
`mvmul:unit`	$N + \sum$
`mmmul:`$M \times K \times N$	N
`mmtmul:`$M \times K \times N$	N
Image Processing	
`box3x3`	W
`composit8bpp`	$W \times H$
`composit32bpp`	$W \times H$
Audio Synthesis	
`vaudio`	> 600
Cryptography	
`idea:cbcdec`	$B/8$
Neural Net	
`quicknet:forward`	50–4,000
`quicknet:train`	50–4,000
`ksofm:forward`	100–200
`ksofm:update`	10–20
SPECint95	
`m88ksim:killtime`	5, 32
`m88ksim:ckbrkpts`	16
`comp:pack`	512
`decomp:unpack`	512
`decomp:copy`	4
`li:mark`	80
`li:sweep`	1000
`ijpeg:rgbycc`	W
`ijpeg:downsample`	$W/2$
`ijpeg:fdct`	W,$W/2$
`ijpeg:mcu`	63
`ijpeg:idct`	8, 16
`ijpeg:upsample`	$W/2$

Table 12.2: Vector lengths in application codes. 1D vectors are of length N. Matrices are M rows by N columns stored in row-major order (C style). Images are W pixels wide by H pixels high; typical values might range from 128×96 for small video-conferencing up to 1920×1152 for HDTV, and even larger for photographic quality imaging. Data streams are B bytes.

Application	Number of Vector Registers
Linear Algebra	
`dot`	4
`vtmmul`	6
`mvmul:unit`	32 (+5%)
`mmmul`	7
`mmtmul`	19 (+11%)
Image Processing	
`box3x3`	9
`composit8bpp`	8
`composit32bpp`	8
Audio Synthesis	
`vaudio`	19 (+26%)
Cryptography	
`idea:cbcdec`	15
Neural Net	
`quicknet:forward`	6
`quicknet:train`	15
`ksofm:forward`	10
`ksofm:update`	6
SPECint95	
`m88ksim:killtime`	2
`m88ksim:ckbrkpts`	2
`comp:pack`	4
`decomp:unpack`	4
`decomp:copy`	1
`li:mark`	11
`li:sweep`	12
`ijpeg:rgbycc`	9
`ijpeg:downsample`	6
`ijpeg:fdct`	23 (+6%)
`ijpeg:mcu`	1
`ijpeg:idct`	23 (+18%)
`ijpeg:upsample`	10

Table 12.3: Number of vector registers required by application kernels running on T0. Where using more than the available 15 vector registers would have improved performance, the required number of vector registers is given together with the performance improvement in parentheses.

12.2 Number of Vector Registers

Table 12.3 presents the number of vector registers used by these kernels when tuned for T0, which has 15 available vector registers. In cases where more registers would have improved performance, the number of registers required and the resulting speedup is given in parentheses.

T0 stores flag values in vector data registers, and so there is no separate measure for flag registers versus data registers. The number of vector data and flag registers required to run a given loop nest at maximum speed depends on the details of an implementation, including chime length, number of VFUs, and VFU latencies. To achieve maximum performance, machines with short chimes and long latencies or no chaining might require more registers for loop unrolling and software pipelining [MSAD91]. Machines with several VFUs might require more registers to avoid false register dependencies from limiting available inter-instruction parallelism [Lee92].

The results show that several of the codes would experience significant speedups with more than 15 vector registers, but none of these codes would benefit significantly from more than 32 vector registers.

12.3 Data Widths

Table 12.4 lists the data widths used by the applications. The data width is given in bits for the largest type in a vector loop nest. This corresponds to the required virtual processor width setting as described in Section 7.3. For these kernels, 16-bit and 32-bit data types are common, suggesting that support for narrower 16-bit and 32-bit virtual processors would improve cost/performance.

12.4 Vector Compute to Vector Memory Ratio

The ratio of vector compute instructions to vector memory instructions is important to help determine an appropriate mixture of VFUs. Table 12.5 presents numbers for the application codes.

Over half of the codes execute several vector arithmetic instructions for every vector memory instruction. The most extreme example is the IDEA decryption code where over 36 arithmetic operations are performed for every memory operation. This is in contrast to results from studies of compiled scientific vector applications [Vaj91, Esp97b, Was96] where the number of floating point operations is generally approximately equal to or less than the number of memory operations. Apart from the choice of workload, another reason for this difference is that the T0 codes were hand-tuned for a machine with 15 vector registers, whereas the previous results were compiled code for machines with 8 vector registers.

The T0 design has two VAUs and one VMU. The results above suggest that adding more VAUs could improve performance in some cases. But in some of these cases, the VAUs are not fully utilized because of the limited address bandwidth on T0 (e.g., see the DCT codes in Section 11.2.4). Because non-contiguous vector memory instructions run more slowly than vector arithmetic instructions, T0 can execute multiple vector arithmetic instructions in each VAU while a single long-running vector memory instruction occupies the VMU. If more address bandwidth is added to the single VMU, then additional VAUs should improve performance for many of these codes.

Application	Width
Linear Algebra	
`dot`	All
`vtmmul`	All
`mvmul:rake:`	All
`mvmul:unit:`	All
`mmmul:`$M \times K \times N$	All
`mmtmul:`$M \times K \times N$	All
Image Processing	
`box3x3`	16
`composit8bpp`	16
`composit32bpp`	32
Audio Synthesis	
`vaudio`	32
Cryptography	
`idea:cbcdec`	16
Neural Net	
`quicknet:forward`	32
`quicknet:train`	32
`ksofm:forward`	32
`ksofm:update`	16
SPECint95	
`m88ksim:killtime`	32
`m88ksim:ckbrkpts`	32
`comp:pack`	32
`decomp:unpack`	32
`decomp:copy`	8
`li:mark`	A
`li:sweep`	A
`ijpeg:rgbycc`	32
`ijpeg:downsample`	16
`ijpeg:fdct`	32
`ijpeg:mcu`	16
`ijpeg:idct`	32
`ijpeg:upsample`	16

Table 12.4: Data widths and types used by applications. The largest types used in a loop nest are indicated. Entries marked “All” could potentially be used with all data widths. The entries are marked with an A if address arithmetic is the largest type required; addresses are assumed to be either 32 bits or 64 bits.

Application	C/M
Linear Algebra	
`dot`	1.0
`vtmmul`	2.0
`mvmul`	2.0
`mmmul` (8 register tile)	16.0
Image Processing	
`box3x3`	4.3
`composit8bpp`	3.3
`composit32bpp`	7.3
Audio Synthesis	
`vaudio`	4.1
Cryptography	
`idea:cbcdec`	36.3
Neural Net	
`quicknet:forward`	2.0
`quicknet:train`	1.6
`ksofm:forward`	3.0
`ksofm:update`	1.5
SPECint95	
`m88ksim:killtime`	1.0
`m88ksim:ckbrkpts`	1.5
`comp:pack`	0.8
`decomp:unpack`	0.8
`decomp:copy`	0.0
`li:mark`	2.4
`li:sweep`	2.8
`ijpeg:rgbycc`	3.3
`ijpeg:downsample`	1.1
`ijpeg:fdct`	3.9
`ijpeg:mcu`	1.0
`ijpeg:idct`	5.4
`ijpeg:upsample`	3.8

Table 12.5: Average number of vector arithmetic instructions per vector memory instruction.

Application	Unit Stride	Strided Rake	Strided	Indexed Rake	Indexed
Linear Algebra					
`dot`	X*		(1-X)		
`vtmmul`	1.00				
`mvmul:rake:`		1.00			
`mvmul:unit:`	$> \frac{R}{R+1}$		$< \frac{1}{R}$		
`mmmul`	1.00				
`mmtmul`	$\frac{1}{K}$	$\frac{K}{K+1}$			
Image Processing					
`box3x3`	1.00				
`composit8bpp`	1.00				
`composit32bpp`	1.00				
Audio Synthesis					
`vaudio`	1.00				
Cryptography					
`idea:cbcdec`		1.00			
Neural Net					
`quicknet:forward`	1.00				
`quicknet:train`	1.00				
`ksofm:forward`	1.00				
`ksofm:update`		1.00			
SPECint95					
`m88ksim:killtime`	0.86		0.14		
`m88ksim:ckbrkpts`		1.00			
`comp:pack`	0.53	0.47			
`decomp:unpack`	0.53	0.47			
`comp:copy`	1.00				
`li:mark`	0.49			0.51	
`li:sweep`		1.00			
`ijpeg:rgbycc`	1.00				
`ijpeg:downsample`	0.50		0.50		
`ijpeg:fdct`	0.57			0.14	0.29
`ijpeg:mcu`	1.00				
`ijpeg:idct`	0.85			0.15	
`ijpeg:upsample`	0.50	0.50			

Table 12.6: Distribution of memory access patterns. * X is the fraction of times dot product is called with unit-stride arguments.

12.5 Vector Memory Access Patterns

In Table 12.6, I present a breakdown of how the various codes access memory. The results are broken down into unit-stride, strided rake, non-rake strided, indexed rake, and non-rake indexed accesses. Multi-column accesses have been grouped together with unit-stride accesses. These codes have not been rewritten to make use of the histogram cache. As can be seen, many of the strided and indexed accesses are in the form of rakes.

Application	Read	Write	Read/Write	Width	Depth
mvmul:rake:$M \times N$	1			M	N
mmtmul:$M \times K \times N$	1			N	K
idea:cbcdec	1	1		NVP	$B/(8 \times \text{NVP})$
ksofm:update			1		
m88ksim:ckbrkpts	1			16	2
comp:pack	1	1		512	8–16
decomp:unpack	1	1		512	8–16
li:mark			1	80	1–4
li:sweep			1	NVP	1000/NVP
ijpeg:fdct		1		W	2
ijpeg:idct		1		W	2–4
ijpeg:upsample		2		W	2

Table 12.7: Number and types of rake access. Columns labeled R, W, and RW, give the number of read, write, and read-modify-write rakes. The rake widths and depths are in numbers of accesses. For `idea:cbcdec` and `li:sweep`, rake width and depth depends on the number of VPs (NVP).

For those applications where rakes are present, Table 12.7 lists the number of rake entries required, with separate entries for read-only rakes, write-only rakes, and read-modify-write rakes. The width and depth of the rakes are also shown. In all of these cases, a rake cache with two rake entries would be sufficient to capture the spatial locality in active rakes.

For the one dimensional rakes in `idea:cbcdec` and `li:sweep`, rake width and depth are inversely related. As more VPs are used to execute the rake, the rake width increases while the rake depth decreases. The number of VPs, and hence the rake width, should be chosen to be just large enough to provide enough parallelism to saturate the vector unit, so that the rake depth, and hence the amount of spatial locality, is maximized.

12.6 Sensitivity to Memory Latency

Section 4.8 describes the classes of vector instruction that expose memory latencies. Table 12.8 gives a breakdown of the codes that are sensitive to memory latency.

Scalar operand streams are the most common way in which memory latency is exposed. But in all of these cases, the scalar address stream is independent of the data being loaded, and so scalar decoupling (Section 4.8) should be able to hide the memory latency.

The other types of sensitivity to memory latency occur infrequently for this set of codes. For the `ijpeg` DCT kernels, indexed memory accesses are used but the index vector is static over the course of the loop, and so memory latency will only affect the first iteration of a stripmined loop over image blocks.

Overall, for these codes, increases in memory latency are not expected to have a large impact on vector performance.

Application	Scalar Read of Vector State	Indexed Vector Memory Instructions	Masked Vector Memory Instructions	Scalar Operand Streams
`vtmmul`				x
`mvmul:rake`				x
`mvmul:unit`				x
`mmmul`				x
`mmtmul`				x
`idea:cbcdec`				x
`quicknet:forward`				x
`quicknet:train`				x
`ksofm:forward`				x
`ksofm:update`				x
`m88ksim:ckbrkpts`	x			
`li:mark`	x	x	x	
`li:sweep`	x		x	
`ijpeg:fdct`		(static)		
`ijpeg:idct`		(static)		

Table 12.8: How memory latency is exposed to the applications.

Chapter 13

Conclusions and Future Work

13.1 Summary of Contributions

The main contributions presented in this thesis are:

- The design, implementation, and evaluation of T0: the first single-chip vector microprocessor. T0 demonstrates that vector architectures are well suited to implementation in commodity microprocessor technologies. The most significant difference between a vector microprocessor and a vector supercomputer are that the tight integration of a vector microprocessor significantly reduces intra-CPU latencies and allows a machine with short chimes to sustain high efficiency.

- The invention of virtual processor caches, a new type of primary cache for vector units. Virtual processor caches are small, highly parallel caches which can reduce address and data bandwidth demands for a vector processor.

- The invention of a vector flag processing model that supports vector speculative execution to allow the vector execution of data-dependent exit loops ("while" loops) while preserving correct exception behavior.

- The design of full-custom VLSI vector register files. I show that the most area-efficient designs have several banks with several ports, rather than many banks with few ports as used by traditional vector supercomputers, or one bank with many ports as used by superscalar microprocessors.

- An analysis of the vectorizability of SPECint95. The results demonstrate that compact vector units can achieve such large speedups on the data parallel portions of the code that they can improve the cost/performance of microprocessors even for codes with low levels of vectorization.

- An analysis of the vectorization of various multimedia and human-machine interface kernels. These measurements show that future vector processors should have at least 16 and preferably 32 vector registers. They also show compute to memory ratios are significantly higher than previously reported for vector supercomputer applications. This difference can be partly explained because these measurements were for hand-optimized multimedia code for a machine with 15 vector registers compared with automatically compiled scientific code for machines with only 8 vector registers. The results also analyze the ways in which the codes are sensitive to memory latency assuming a decoupled vector pipeline [EV96]. Most of the codes are insensitive to memory latency. Scalar operand streams are the most common form of exposure to memory latency for the remaining codes, but in all cases these are amenable to scalar decoupling to tolerate the latency.

- The design of a decoupled vector pipeline [EV96] that supports demand-paged virtual memory and IEEE floating-point. I also show that only a few classes of vector instruction are exposed to memory latency with a decoupled vector pipeline.

13.2 Related Work

Compared to the flood of literature on scalar microarchitecture, there has been very little published work on vector architectures. Lee's thesis [Lee92] considered several variants of register file design. Vajapeyam's thesis provides a detailed performance data for the Cray Y-MP executing PERFECT benchmark code [Vaj91]. Espasa's thesis [Esp97b] explores a range of microarchitectural techniques, including the decoupled pipeline and out-of-order vector execution. The emphasis was on improving the efficiency of precompiled codes for an existing vector supercomputer architecture (Convex). Several studies have examined vector caching, but only using slight variants of traditional caches [GS92, KSF+94, FP91].

13.3 Future Work

There are several avenues of research which lead on from this work:

Application Vectorization

More vectorized applications taken from future compute-intensive applications are required to help design future vector microprocessors. Many potential applications have never been considered for vectorization.

Advanced Vector Compilation

Although vector compilation is relatively mature, research in this area was mostly abandoned with the introduction of parallel microprocessor-based systems. Vector microprocessors can benefit from further advances in vector compiler technology. Several new vector compilation challenges are introduced by architectural ideas introduced in this work: vector instruction scheduling with short chimes, speculative vector execution, and optimization for virtual processor caches.

Performance Evaluation of Future Vector Microprocessor Architectures

The design work in this thesis describes ways of building future vector microprocessors. Simulation work is required to determine the appropriate combination of features in future vector microprocessors. In particular, vector microprocessor memory systems could potentially contain many different forms of caching. The interaction between these various forms of cache needs research.

Improved Vector Architectures

Advances in vector instruction sets can increase the usefulness of vector hardware. One interesting possibility is to add some degree of autonomy to the virtual processors in the vector unit to allow more loops to make use of vector unit hardware. In a conventional vector unit, all virtual processors execute the same operation at the same time. If we skew the time at which each virtual processor executes its operations, we may be able to use the vector unit hardware to execute DOACROSS loops, where there are loop carried dependences that otherwise inhibit vectorization.

Low-power Vector IRAMs

This work suggests that vectors may be an energy-efficient approach for executing compute-intensive tasks. In particular, vector IRAMs [KPP+97] which integrate a low-power vector unit with a high-bandwidth subsystem on a single die have the potential to form the standard processor for portable computing applications.

Heterogenous Parallel Processor Architectures

Fabrication technology has now advanced sufficiently that highly parallel single-chip microprocessors are possible. Developing viable architectures for these future highly parallel microprocessors is the greatest challenge in high performance computer architecture. Most proposals for highly parallel microprocessor designs [Be97] are based on a *homogenous* array of general purpose processing elements.

But the success of a vector machine is due to its *heterogeneity*. In a vector machine, a powerful master processor (the scalar unit) controls a large array of less capable but highly specialized slave processors (the virtual processors in the vector unit). In the case of a vector unit, the slave processors attain great efficiencies by only targeting highly data parallel tasks. Different forms of slave processor might be able to attain high efficiencies by only targeting highly thread-parallel tasks that are not vectorizable, or by only targeting highly instruction-parallel tasks that are not vectorizable or parallelizable. It is possible that a powerful master scalar processor controlling several different ensembles of specialized slave processors, all sharing the same memory space, could be the most efficient way of exploiting all forms of parallelism in an application.

Bibliography

[AAW+96] S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S. Liao, B. R. Murphy, R. S. French, M. S. Lam, and M. W. Hall. Multiprocessors from a software perspective. *IEEE Micro*, 16(3):52–61, June 1996.

[AB89] A. W. Appel and A. Bendiksen. Vectorized garbage collection. *Journal of Supercomputing*, 3:151–160, 1989.

[AB97] K. Asanović and J. Beck. T0 Engineering Data, Revision 0.14. Technical Report CSD-97-931, Computer Science Division, University of California at Berkeley, 1997.

[ABC+93] K. Asanović, J. Beck, T. Callahan, J. Feldman, B. Irissou, B. Kingsbury, P. Kohn, J. Lazzaro, N. Morgan, D. Stoutamire, and J. Wawrzynek. CNS-1 Architecture Specification. Technical Report TR-93-021, International Computer Science Institute, 1993.

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

[ABI+96] K. Asanović, J. Beck, B. Irissou, B. Kingsbury, and J. Wawrzynek. T0: A single-chip vector microprocessor with reconfigurable pipelines. In H. Grünbacher, editor, *Proc. 22nd European Solid-State Circuits Conf.*, pages 344–347, Neuchâtel, Switzerland, September 1996. Editions Frontières.

[ABJ+98] K. Asanović, J. Beck, D. Johnson, J. Wawrzynek, B. Kingsbury, and N. Morgan. *Parallel Architectures for Artificial Neural Networks — Paradigms and Implementations*, chapter 11, pages 609–641. IEEE Computer Scociety Press, 1998. In Press.

[ABK+92] K. Asanović, J. Beck, B. E. D. Kingsbury, P. Kohn, N. Morgan, and J. Wawrzynek. SPERT: A VLIW/SIMD microprocessor for artificial neural network computations. In *Proc. Application Specific Array Processors 1992*, pages 179–190, Berkeley, USA, August 1992.

[AG96] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[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.

[AKMW90] K. Asanović, B. E. D. Kingsbury, N. Morgan, and J. Wawrzynek. HiPNeT-1: A highly pipelined architecture for neural network training. In *IFIP Workshop on Silicon Architectures for Neural Nets*, pages 217–232, St Paul de Vence, France, November 1990.

[AM91] K. Asanović and N. Morgan. Experimental determination of precision requirements for back-propagation training of artificial neural networks. In *Proc. 2nd Int. Conf. on Microelectronics for Neural Networks*, Munich, October 1991.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conf. Proc.*, pages 483–485, 1967.

[AS90] M. Afghahi and C. Svensson. A unified single-phase clocking scheme for VLSI systems. *IEEE JSSC*, 25(1):225–233, February 1990.

[Asa97] K. Asanović. A fast Kohonen net implementation for Spert-II. In J. Mira, R. Moreno-Dĭaz, and J. Cabestany, editors, *Biological and Artificial Computation: From Neuroscience to Technology.*, volume 1240 of *Lecture Notes in Computer Science*, pages 792–800, Lanzarote, Canary Islands, Spain, June 1997. Springer. Proc. Int. Work-Conf. on Artificial and Natural Neural Networks, IWANN'97.

[ASPF92] K. Asanović, K. E. Schauser, D. A. Patterson, and E. H. Frank. Evaluation of a stall cache: An efficient restricted on-chip instruction cache. In *Proc. 25th HICSS*, pages 405–415, January 1992.

[AT93] M. Awaga and H. Takahashi. The μVP 64-bit vector coprocessor: A new implementation of high-performance numerical computation. *IEEE Micro*, 13(5):24–36, October 1993.

[BAB$^+$95] W. Bowhill, R. L. Allmon, S. L. Bell, E. M. Cooper, D. R. Donchin, J. H. Edmondson, T. C. Fischer, P. E. Gronowski, A. K. Jain, P. L. Kroesen, B. J. Loughlin, R. P. Preston, P. I. Rubinfield, M. J. Smith, S. C. Thierauf, and G. M. Wolrich. A 300MHz quad-issue CMOS RISC microprocessor. In *Proc. ISSCC*, volume 38, pages 182–183, February 1995.

[BB96] T. D. Burd and R. W. Broderson. Processor design for portable systems. *Journal of VLSI Signal Processing*, 13(2/3):203–222, August/September 1996.

[Be97] D. Burger and J. R. Goodman (editors). Special issue on billion-transistor architectures. *IEEE Computer*, 30(9), September 1997.

[BH95] G. T. Byrd and M. A. Holliday. Multithreaded processor architectures. *IEEE Spectrum*, 32(8):38–46, August 1995.

[BKQW95] M. Bass, P. Knebel, D. W. Quint, and W. L. Walker. The PA 7100LC microprocessor: a case study of IC design decisions in a competitive environment. *Hewlett-Packard Journal*, 46(2):12–22, April 1995.

[Bro89] E. Brooks. The attack of the killer micros. In *Teraflop Computing Panel, Supercomputing '89*, Reno, Nevada, 1989.

[Buc86] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[Cas96] B. Case. RM7000 strong on cost/performance. *Microprocessor Report*, 10(14):36–39, October 1996.

[CB94a] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proc. 21st ISCA*, pages 223–232, April 1994.

[CB94b] Z. Cvetanovic and D. Bhandarkar. Characterization of Alpha AXP performance using TP and SPEC workloads. In *Proc. 17th ISCA*, pages 60–70, April 1994.

[CCB92] A. P. Chandrakasan, S. Cheng, and R. W. Broderson. Low-power CMOS digital design. *IEEE JSSC*, 27(4):473–484, April 1992.

[CD96] Z. Cvetanovic and D. D. Donaldson. AlphaServer 4100 performance characterization. *Digital Technical Journal*, 8(4):3–20, 1996.

[CDd+95] A. Charnas, A. Dalal, P. deDood, P. Ferolito, B. Frederick, O. Geva, D. Greenhill, H. Hingarh, J. Kaku, L. Kohn, L. Lev, M. Levitt, R. Melanson, S. Mitra, R. Sundar, M. Tamjidi, P. Wang, D. Wendell, R. Yu, and G. Zyner. A 64b microprocessor with multimedia support. In *Proc. ISSCC*, volume 38, pages 178–179, February 1995.

[CDM+97] R. Crisp, K. Donnelly, A. Moncayo, D. Perino, and J. Zerbe. Development of single-chip multi-GB/s DRAMs. In *Proc. ISSCC*, volume 40, pages 226–227, February 1997.

[CI94] T. Cornu and P. Ienne. Performance of digital neuro-computers. In *Proc. Fourth Int. Conf. on Microelectronics for Neural Networks and Fuzzy Systems*, pages 87–93. IEEE Computer Society Press, September 1994.

[Con94] Convex Computer Corporation, 3000 Waterview Parkway, P.O. Box 833851, Richardson, TX 750833-3851. *Convex Data Sheet, C4/XA Systems*, 1994.

[Cor89] Unisys Corporation. Scientific processor vector file organization. U.S. Patent 4,875,161, October 1989.

[Cor95] Standard Performance Evaluation Corporation. Spec95, 1995.

[Cor96a] Intel Corporation. Color conversion from YUV12 to RGB using Intel MMX technology. Intel MMX Technology Application Note, 1996.

[Cor96b] Intel Corporation. Using MMX instructions in a fast iDCT algorithm for MPEG decoding. Intel MMX Technology Application Note 528, 1996.

[Cor96c] Intel Corporation. Using MMX instructions to convert RGB To YUV color conversion. Intel MMX Technology Application Note, 1996.

[Cor96d] Intel Corporation. Using MMX instructions to implement alpha blending. Intel MMX Technology Application Note 554, 1996.

[Cra93] Cray Research, Incorporated, Chippewa Falls, WI 54729. *Cray Y-MP C90 System Programmer Reference Manual*, 001 edition, June 1993. CSM-0500-001.

[DCDH90] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[DCHH88] J. Dongarra, J. Du Cros, S. Hammarling, and R.J. Hanson. An extended set of FORTRAN basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[DEC90a] DEC. Method and apparatus for executing instructions for a vector processing system. U.S. Patent 4,949,250, August 1990.

[DEC90b] DEC. Vector register system for executing plural read/write commands concurrently and independently routing data to plural read/write ports. U.S. Patent 4,980,817, December 1990.

[DHM+88] T. Diede, C. F. Hagenmaier, G. S. Miranker, J. J. Rubenstein, and Jr. W. S. Worley. The Titan graphics supercomputer architecture. *IEEE Computer*, pages pp13–30, September 1988.

[ERB+95] J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, B. J. Benschneider, D. Bernstein, R. W. Castelino, E. M. Cooper, D. E. Dever, D. R. Donchin, T. C. Fisher, A. K. Jain, S. Mehta, J. E. Meyer, R. P. Preston, V. Rajagopalan, C. Somanathan, S. A. Taylor, and G. M. Wolrich. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

[Esp97a] R. Espasa. Traces supplied by Espasa were measured to obtain distribution of unit-stride, strided, and indexed operations, 1997.

[Esp97b] R. Espasa. *Advanced Vector Architectures*. PhD thesis, Universitat Politècnica de Catalunya, February 1997.

[EV96] R. Espasa and M. Valero. Decoupled vector architectures. In *Proc. 2nd High Performance Computer Architecture Conf.*, pages 281–290. IEEE Computer Society Press, Feb 1996.

[EV97a] R. Espasa and M. Valero. A victim cache for vector registers. In *Proc. Int. Conf. on Supercomputing*. ACM Press, July 1997.

[EV97b] R. Espasa and M. Valero. Exploiting instruction- and data- level parallelism. *IEEE Micro*, pages 20–27, September/October 1997.

[EVS97] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *Proc. Micro-30*, pages 160–170. IEEE Press, December 1997.

[FFG+95] D. M. Fenwick, D. J. Foley, W. B. Gist, S. R. VanDoren, and D. Wissel. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

[Fis83] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *Proc. 10th ISCA*, pages 140–150. Computer Society Press, 1983.

[Fly66] M. J. Flynn. Very high-speed computing systems. *Proc. IEEE*, 54(12):1901–1909, December 1966.

[FP91] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proc. 18th ISCA*, pages 54–63, Toronto, Canada, May 1991.

[FPC+97] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick. The energy efficiency of IRAM architectures. In *Proc. 24th ISCA*, Denver, Colorado, June 1997.

[GH95] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose processors. In *Proc. IEEE Symp. on Low Power Electronics*, pages 12–13, October 1995.

[GHPS93] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, August 1993.

[Gol96] J. Golston. Single-chip H.324 videoconferencing. *IEEE Micro*, 16(4):21–33, August 1996.

[GS92] J. D. Gee and A. J. Smith. Vector processor caches. Technical Report UCB/CSD 92/707, University of California at Berkeley, September 1992.

[Gwe93] L. Gwennap. TFP designed for tremendous floating point. *Microprocessor Report*, pages 9–13, August 1993.

[Gwe94a] L. Gwennap. Digital leads the pack with 21164. *Microprocessor Report*, 8(12):1–10, September 1994.

[Gwe94b] L. Gwennap. PA-7200 enables inexpensive MP systems. *Microprocessor Report*, 8(3):12–15, March 1994.

[Gwe95a] L. Gwenapp. Integrated PA-7300LC powers HP midrange. *Microprocessor Report*, 9(15):12–15, November 1995.

[Gwe95b] L. Gwenapp. PA-8000 stays on track. *Microprocessor Report*, 9(15):14, November 1995.

[Gwe95c] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, February 1995.

[Gwe96a] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):11–16, October 1996.

[Gwe96b] L. Gwennap. Exponential's PowerPC blazes. *Microprocessor Report*, 10(14):1,6–10, October 1996.

[Gwe96c] L. Gwennap. IBM crams POWER2 onto single chip. *Microprocessor Report*, 10(10):14–16, August 1996.

[Gwe96d] L. Gwennap. Intel's MMX speeds multimedia. *Microprocessor Report*, 10(3):1,6–10, March 1996.

[Gwe96e] L. Gwennap. R5000 improves FP for MIPS midrange. *Microprocessor Report*, 10(1):10–12, January 1996.

[Gwe97] L. Gwennap. Klamath extends P6 family. *Microprocessor Report*, 11(2):1,6–8, February 1997.

[Han96] C. Hansen. Microunity's media processor architecture. *IEEE Micro*, 16(4):34–41, August 1996.

[Hew89] Hewlett-Packard Company. *HP Precision Architecture and Instruction Reference Manual*, third edition, 1989. HP part number 09740-90014.

[HHK93] D. Hammerstrom, W. Henry, and M. Kuhn. *Neurocomputer System for Neural-Network Applications*, chapter 4. Prentice Hall, 1993.

[HL96] S. W. Hammond and R. D. Loft. Architecture and application: The performance of NEC SX-4 on the NCAR benchmark suite. In *Proc. Supercomputing '96*, 1996. `http://www.scd.ucar.edu/css/sc96/sc96.html`

[Hod97] T. D. Hodes. Recursive oscillators on a fixed-point vector microprocessor for high performance real-time additive synthesis. Master's thesis, Computer Science Division, EECS Department, University of California at Berkeley, Berkeley, California, CA94720, December 1997.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitative Approach, Second Edition*. Morgan Kaufmann, 1996.

[HS93] W.-C. Hsu and J. E. Smith. Performance of cached DRAM organizations in vector supercomputers. In *Proc. 20th ISCA*, pages 327–336, San Diego, California, May 1993.

[HT72] R. G. Hintz and D. P. Tate. Control Data STAR-100 processor design. In *Proc. COMPCON*, pages 1–4. IEEE, 1972. Also in: IEEE Tutorial on Parallel Processing. R. H. Kuhn and D. A. Padua (editors), 1981, pp36–39.

[IBM87] IBM. Vector processing unit. U.S. Patent 4,791,555, March 1987.

[IBM96] IBM Corporation. *Data Sheet for IBM0316409C, IBM0316809C, IBM0316809C 16Mb Synchronous DRAMs*, January 1996.

[ICK96] P. Ienne, T. Cornu, and G. Kuhn. Special-purpose digital hardware for neural networks: An architectural survey. *Journal of VLSI Signal Processing*, 13(1):5–25, 1996.

[IEE85] IEEE Standard for Binary Floating Point Arithmetic. ANSI/IEEE Std 754-1985, 1985.

[IEE90] IEEE Standard Test Access Port and Boundary-Scan Architecture. IEEE Std 1149.1-1990, 1990.

[Joh91] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall Series in Innovative Technology. P T R Prentice-Hall, 1991.

[Jon89] T. Jones. Engineering design of the Convex C2. *IEEE Computer*, 22(1):36–44, January 1989. A version of this article also appeared in HICSS, 1989.

[Kan89] G. Kane. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1989.

[Kil97] M. J. Kilgard. Realizing OpenGL: Two implementations of one architecture. In *Proc. 1997 SIGGRAPH Eurographics Workshop on Graphics Hardware*, 1997. Available at `http://reality.sgi.com/mjk/twoimps/twoimps.html`

[KIS+94] K. Kitai, T. Isobe, T. Sakakibara, S. Yazawa, Y. Tamaki, T. Tanaka, and K. Ishii. Distributed storage control unit for the Hitachi S-3800 multivector supercomputer. In *Proc. Int. Conf. on Supercomputing*, pages 1–10, Manchester, UK, July 1994.

[Koh82] T. Kohonen. Self-organizing formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, 1982.

[KPP+97] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, K. Keeton, R. Thomas, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[KSF+94] L. Kontothanassis, R. A. Sugumar, G. J. Faanes, J. E. Smith, and M. L. Scott. Cache performance in vector supercomputers. In *Supercomputing '94*, 1994.

[Laz87] C. Lazou. *Supercomputers and their Use*. Clarendon Press, 1987.

[Lee92] C. Lee. *Code optimizers and register organizations for vector architectures*. PhD thesis, University of California at Berkeley, May 1992.

[Lee97] R. Lee. Effectiveness of the MAX-2 multimedia extensions for PA-RISC processors. In *Hot Chips IX, Symposium Record*, pages 135–147, Stanford, California, 1997.

[LH97] G. Lesartre and D. Hunt. PA-8500: The continuing evolution of the PA-8000 family. In *Compcon '97*, 1997.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[Lin82] N. R. Lincoln. Technology and design tradeoffs in the creation of a modern supercomputer. *IEEE Trans. on Computers*, C-31(5):349–362, May 1982. Also in: IEEE Tutorial Supercomputers: Design and Applications. Kai Hwang(editor), pp32–45.

[LL97] J. Laudon and D. Lenoski. "the sgi origin: A ccnuma highly scalable server". In *Proc. 24th ISCA*, June 1997.

[LLM89] C. Loeffler, Adriaan Ligtenberg, and G. S. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *Proc. ICASSP*, pages 988–991, 1989.

[LMM85] O. Lubeck, J. Moore, and R. Mendez. A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2. *IEEE Computer*, December 1985.

[Mar96] R. Martin. A vectorized hash-join. IRAM course project report, University of California at Berkeley. Available from: `http://HTTP.CS.Berkeley.EDU/~rmartin/iram_proj3/report.ps`, May 1996.

[MBK+92] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, and J. Beer. The Ring Array Processor (RAP): A multiprocessing peripheral for connectionist applications. *Journal of Parallel and Distributed Computing*, 14:248–259, April 1992.

[McC] J. D. McCalpin. QG_GYRE and memory bandwidth. Available from: `http://reality.sgi.com/mccalpin/hpc/balance/qgbox/case_study.html`

[McC95] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, December 1995.

[McC97a] J. D. McCalpin. Personal communication, February 1997. STREAM performance of O2 R5K and O2 R10K.

[McC97b] J. D. McCalpin. Memory bandwidth STREAM benchmark results. Available from `http://perelandra.cms.udel.edu/hpc/stream`, August 1997.

[MHM+95] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proc. 22nd ISCA*, pages 138–149, June 1995.

[MS95] G. Myklebust and J. G. Solheim. Parallel self-organizing maps for actual applications. In *Proc. of the IEEE Int. Conf. on Neural Networks*, Perth, 1995.

[MSAD91] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Vector register design for polycyclic vector scheduling. In *Proc. 4th ASPLOS*, pages 154–163, April 1991.

[MU84] K. Miura and K. Uchida. FACOM vector processor system: VP-100/VP-200. In Kawalik, editor, *Proc. of NATO Advanced Research Workshop on High Speed Computing*, volume F7. Springer-Verlag, 1984. Also in: IEEE Tutorial Supercomputers: Design and Applications. Kai Hwang(editor), pp59-73.

[MWA+96] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, A. Fatell, G. W Hoeppner, D. Kruckmeyer, T. H. Lee, P. Lin, L. Madden, D. Murray, M. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf. A 160MHz 32b 0.5W CMOS RISC microprocessor. In *Proc. ISSCC, Slide Supplement*, February 1996.

[NL97] B. Neidecker-Lutz. Usenet posting, March 1997.

[Oed92] W. Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18:947–954, 1992.

[OHO+91] F. Okamoto, Y. Hagihara, C. Ohkubo, N. Nishi, H. Yamada, and T. Enomoto. A 200-MFLOPS 100-MHz 64-b BiCMOS vector-pipelined processor (VPP) ULSI. *IEEE JSSC*, 26(12):1885–1893, December 1991.

[PAC+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM: IRAM. *IEEE Micro*, 1997.

[PK94] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proc. 21st ISCA*, pages 24–33, April 1994.

[PM93] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993. ISBN 0-442-01272-1.

[Prz96] S. Przybylski. *New DRAM Technologies*. Microdesign Resources, 1996.

[Rei95] M. J. Reizenman. The search for better batteries. *IEEE Spectrum*, 32(5):51–56, May 1995.

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

[Rus78] R. M. Russel. The Cray-1 Computer System. *CACM*, 21(1):63–72, January 1978.

[SB95] S. Saini and D. H. Bailey. NAS parallel benchmarks results 3-95. Technical Report NAS-95-011, NASA Ames Research Center, April 1995.

[Sch87] W. Schoenauer. *Scientific Computjng on Vector Supercomputers*. Elsevier Science Publishers B.V., P.O. Box 1991, 1000 BZ Amsterdam, The Netherlands, 1987.

[Sch96] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[Sch97a] R. C. Schumann. Design of the 21174 memory controller for Digital personal workstations. *Digital Technical Journal*, 9(2):57–70, 1997.

[Sch97b] J. Schwarzmeier. Personal communication, 1997. Cray Research.

[Sco96] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. 7th ASPLOS*, pages 26–36, 1996.

[SGH97] V. Santhanam, E. H. Gornish, and W. Hsu. Data prefetching on the HP PA-8000. In *Proc. 24th ISCA*, pages 264–273, June 1997.

[SHH90] J. E. Smith, W.-C. Hsu, and C. Hsiung. Future general purpose supercomputer architectures. In *Proc. Supercomputing '90*, pages 796–804, 1990.

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

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

[Smi94] J. E. Smith. The best way to achieve vector-like performance? Use vectors. In *Proc. 21st ISCA*, April 1994. Keynote talk, slides available at `http://www.ece.wisc.edu/~jes/pitches/vector.ps`

[Smi97] J. E. Smith. Designing Vector ISAs. Talk to IRAM group at University of California at Berkeley, September 17 1997.

[Sun92] Sun Microsystems Computer Corporation, 2550 Garcia Avenue, Mountain View, CA94043, USA. *The SuperSPARC Microprocessor*, May 1992. Technical White Paper.

[Tex97] Texas Instruments Europe. *Implementation of an Image Processing Library for the TMS320C8x (MVP)*, July 1997. Literature Number: BPRA059.

[TONH96] M. Tremblay, J. M. O'Connor, V. Narayanan, and L. He. VIS speeds new media processing. *IEEE Micro*, 16(4):10–20, October 1996.

[Top97] S. Toprani. Rambus consumer technology powers the Nintendo 64. Rambus White Paper, 1997. Available from `www.rambus.com`

[Tru97] L. Truong. The VelociTi architecture of the TMS320C6xxx. In *Hot Chips IX, Symposium Record*, pages 55–63, Stanford, California, 1997.

[UHMB94] M. Upton, T. Huff, T. Mudge, and R. Brown. Resource allocation in high clock rate microprocessors. In *Proc. 6th ASPLOS*, pages 98–109, October 1994.

[UIT94] T. Utsumi, M. Ikeda, and M. Takamura. Architecture of the VPP500 parallel supercomputer. In *Proc. Supercomputing '94*, pages 478–487, November 1994.

[Vaj91] S. Vajapeyam. *Instruction-Level Characterization of the Cray Y-MP Processor*. PhD thesis, University of Wisconsin-Madison, 1991.

[VKY+96] N. Vasseghi, P. Koike, L. Yang, D. Freitas, R. Conrad, A. Bomdica, L. Lee, S. Gupta, M. Wang, R. Chang, W. Chan, C. Lee, F. Lutz, F. Leu, H. Nguyen, and Q. Nasir. 200MHz superscalar RISC processor circuit design issues. In *Proc. ISSCC*, volume 39, pages 356–357, February 1996.

[W.74] P.J. W. *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, Dept. of Applied Mathematics, Harvard University, 1974.

[WAC+92] J. Wade, Z. Abuhamdeh, D. Cassiday, M. Ganmukhi, J. Hill, B. Lordi, M. St. Pierre, S. Smith, M. Wong, S. Yang, B. Zak, D. Bural, D. Steiss, B. Farrel, S. Koval, M. Gill, and K. Cyr. The vector coprocessor unit (VU) for the CM-5. In *Hot Chips IV*, August 1992.

[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.

[Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proc. 4th ASPLOS*, pages 176–188, April 1991.

[WAM93] J. Wawrzynek, K. Asanović, and N. Morgan. The design of a neuro-microprocessor. *IEEE Trans. on Neural Networks*, 4(3):394–399, May 1993.

[Was96] H. J. Wasserman. Benchmark tests on the Digital Equipment Corporation Alpha AXP 21164-based AlphaServer 8400, including a comparison on optimized vector and scalar processing. In *Proc. 1996 Int. Conf. on Supercomputing*, pages 333–340. ACM, May 1996.

[WD93] S. W. White and S. Dhawan. POWER2: Next generation of the RISC System/6000 family. In *IBM RISC System/6000 Technology: Volume II*. IBM, 1993.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[WG94] D. L. Weaver and T. Germond. *The SPARC Architecture Manual/Version 9*. Prentice Hall, February 1994.

[WS94] S. Weiss and J. E. Smith. *POWER and PowerPC: Principles, Architecture, Implementation*. Morgan Kaufmann, 1994.

[Yan93] Q. Yang. Introducing a new cache design into vector computers. *IEEE Transactions on Computers*, 42(12):1411–1424, December 1993.

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

[ZB91] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proc. Supercomputing '91*, 1991.

[ZML95] J. H. Zurawski, J. E. Murray, and P. J. Lemmon. The design and verification of the AlphaStation 600 5-series workstation. *Digital Technical Journal*, 7(1):89–99, 1995.