Low-complexity Vector Microprocessor Extensions

Joseph James Gebis

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-47
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-47.html

May 6, 2008

Low-complexity Vector Microprocessor Extensions

by

Joseph James Gebis

B.S. (University of Illinois at Urbana-Champaign) 1996
M.S. (University of California, Berkeley) 2003

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 David A. Patterson, Chair
Professor Katherine Yelick
Professor Andrea diSessa

Spring 2008

The dissertation of Joseph James Gebis is approved:

Chair	____________________	Date	__________
	____________________	Date	__________
	____________________	Date	__________

University of California, Berkeley

Spring 2008

Abstract

Low-complexity Vector Microprocessor Extensions

by

Joseph James Gebis

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David A. Patterson, Chair

For the last few years, single-thread performance has been improving at a snail's pace. Power limitations, increasing relative memory latency, and the exhaustion of improvement in instruction-level parallelism are forcing microprocessor architects to examine new processor design strategies. In this dissertation, I take a look at a technology that can improve the efficiency of modern microprocessors: vectors. Vectors are a simple, power-efficient way to take advantage of common data-level parallelism in an extensible, easily-programmable manner. My work focuses on the process of transitioning from traditional scalar microprocessors to computers that can take advantage of vectors.

First, I describe a process for extending existing single-instruction, multiple-data instruction sets to support full vector processing, in a way that remains binary compatible with existing applications. Initial implementations can be low cost, but be transparently extended to higher performance later.

I also describe ViVA, the Virtual Vector Architecture. ViVA adds vector-style memory operations to existing microprocessors but does not include arithmetic datapaths; instead, memory instructions work with a new buffer placed between the core and second-level cache. ViVA serves as a low-cost solution to getting much of the performance of full vector memory hierarchies while avoiding the complexity of adding a full vector system.

Finally, I test the performance of ViVA by modifying a cycle-accurate full-system simulator to support ViVA's operation. After extensive calibration, I test the basic performance of ViVA using a series of microbenchmarks. I compare the performance of a variety of ViVA configurations for corner turn, used in processing multidimensional data, and sparse matrix-vector multiplication, used in many scientific applications. Results show that ViVA can give significant benefit for a variety of memory access patterns, without relying on a costly hardware prefetcher.

Professor David Patterson

Dissertation Committee Chair

Dedication

For Kendra

Contents

List of Figures

List of Tables

Acknowledgements

There are a great number of people that, through their advice and support, have made my work possible.

◇

Most importantly, I want to thank Dave Patterson, my advisor. His enthusiasm for research is only exceeded by the quality of his guidance. He gave me great advice, for both academics and beyond. Not only did he help keep me focused, he helped me to keep things in perspective.

I'd also like to thank my Qualifying Exam committee, John Wawrzynek, Kathy Yelick, and Andy diSessa. Their feedback and direction guided my dissertation research and writing. Special thanks to Kathy and Andy for agreeing to continue helping me on my dissertation committee.

◇

Thanks also to other members of the Berkeley academic community and LBL that have offered me guidance, assistance, and support. John Shalf, Lenny Oliker, Krste Asanović, and Kurt Keutzer have given me incredibly useful feedback and suggestions that helped direct my research and investigations.

A number of administrative personnel have given me huge amounts of assistance in working through the bureaucracy and regulations that can be so frustrating to deal with. In particular, La Shana Porlaris helped decode the rules for me many times.

I've frequently gone to Jon Kuroda and Mike Howard for help with setting up a computer, getting access to a server, and many other things. They've always gone out of their way to help me get what I need.

I've been lucky enough to perform research with many other people that have helped me extensively. I was able to work on interesting web research — quite a change of pace, for me — at IBM Almaden, and received support and guidance on simulator work from IBM Austin. Thank you to Jim Beck and Steve Pope for their work and collaboration on VIRAM1, and thank you to everyone in the Berkeley View and Parlab groups that helped me.

My first introduction to teaching was through Michael C. Loui at the University of Illinois. I appreciate his guidance and advice. I was fortunate to be able to continue teaching here at Berkeley and through Cal VIEW.

I could not imagine having a better group of peers to work with than the ones I've met here. Christos Kozyrakis and Sam Williams worked with me for many years, and I'm grateful to have had a chance to collaborate with them.

◇

My family has always been available for me, even when I've been tough to deal with. Thank you.

Kendra has given me immeasurable support.

All of my friends and a huge number of others have helped me get through my graduate student experience by helping to provide diversions outside of academics, especially the crew at Gilman.

◇

Thanks also to Maude. RIP.

Chapter 1

Introduction

For the last few years, single-thread performance has been improving at a snail's pace. A "perfect storm" of trends have come together to reduce, or even halt, the rate at which microprocessors get faster: the memory wall of increasing relative memory latency; the power wall, where processors are generating as much heat as can possibly be dissipated without moving to liquid cooling; and the ILP wall, where huge amounts of resources are used to try to extract smaller and smaller benefits in the number of instructions that can be run at once. On top of those limits, the "free lunch" of cooler, faster transistors being delivered every 18 months by a new silicon technology generation is now over: when a new generation arrives, it comes with drawbacks, such as greatly increased leakage that can limit clock speed [SHK$^+$05]. The combination of these walls (which have collectively been called a "brick wall" for increasing single-thread performance [Pat07]) and the reduction in benefits from new silicon technology has forced microprocessor manufacturers to consider significantly new designs, such as multicore [Paw07] and simultaneous multithreading [EEL$^+$97].

In this dissertation, I take a look at a different technology that can improve the efficiency of microprocessors: vectors. The first observation is that there is a large amount of data-level parallelism available in important programs, and vectors are a flexible and efficient way to take advantage of it. Other people have made this point before, but it is an important idea that is especially relevant today, as the increasing challenges of silicon technology are forcing the industry to search for efficient computational techniques.

The second main idea that this work is predicated upon is that vectors are a rich tapestry: it is not a single implementation idea, it is a family that covers many different instantiations. Because of that, this thesis takes a look at the process of transitioning from traditional microprocessors to computers that can take advantage of vectors. Not only are there many possible end systems, there are a number of approaches to get to those targets. In this thesis, I study and evaluate two targets, with a variety of configurations for each. Specifically, I am looking at a range of simple, low-cost approaches for moving traditional microprocessors towards vector processors, but those techniques are just a few of the systems that are possible.

1.1 Thesis Overview

In the rest of this chapter, I will describe my motivation and the specific contributions that I am making with this work. I will also cover some background on vectors and the implications of using vector computers. For a longer discussion of vector processor operation, see [AHP06] and [Asa87]. My contributions build on prior projects of others and mine; I describe those projects and additional related work on vector processors in Chapter 2.

In Chapter 3, I describe the details of my approaches of extending microprocessors. First, I describe ViVA (Virtual Vector Architecture), which adds vector-style memory operations but does not include vector arithmetic or logical datapaths. It serves as a low-cost solution to getting much of the performance of vector memory systems. I will also describe a process for extending existing ISAs to support full vector processing.

Chapter 4 contains the explanation of the software techniques used for programming my proposed vector systems, as well as a description of the benchmarks that I use to evaluate them and the reason that those benchmarks were chosen. The chapter ends with a description of my simulation environment.

In Chapter 6, I examine the results of the simulations, and discuss conclusions that I can draw from that work. Finally, in Chapter 7, I present my overall conclusions, and discuss the future directions that my work may take.

1.2 Motivation

A few years ago, a group of Berkeley computer science researchers got together to have meetings to discuss a dramatic change in the processor landscape. The group, which included domain experts from the embedded, desktop, and supercomputing fields, discussed recent trends, where they would lead, and how the computing industry should respond. Eventually, we published our conclusions in a report called the Berkeley View on Parallelism [ABC+06].

The Berkeley View includes a number of important ideas that are particularly relevant to my work. First and foremost is that computer chip manufacturers can no longer proceed along the same path that they have followed in the past. Indeed, it seems that manufacturers are realizing this. Most desktop machines — and laptops, as well — now include at least two processor cores, and manufacturers have begun to look at other techniques to take advantage of parallelism.

Another important idea is that software will have to change, as well. Microprocessor architectures are being forced to adapt. Part of that adaptation will allow them to take advantage of parallelism more effectively, but software has to expose the parallelism that is available in a particular algorithm or application: relying on hardware to extract it, by itself, is inefficient in both the amount of parallelism that can be extracted, and the power required to find it.

Finally, the Berkeley View presents a case that architects should design cores to be energy-efficient. That means that they should enable programs to express parallelism easily, and that they should take advantage of that parallelism effectively.

Even though processors should be designed with parallelism in mind, single-thread performance continues to remain important. The constraints of Amdahl's Law [HP06] will continue to apply: any part of an application that cannot be parallelized will limit overall performance.

The previous work of others and mine, described more in Chapter 2, has shown that vectors are an effective tool to achieve the above goals: good single-thread performance and an effective, power-efficient way to use parallelism. VIRAM1 [GWKP04] was a low-power media-oriented vector microprocessor; my experiences working on it led me to ask the questions investigated here: How do vectors work with more modern microprocessors? Is it possible to get some of the advantages of vectors with a small hardware investment? How can scalar processors be extended to work with vectors?

1.3 Contributions

The main contributions I present in this thesis are:

- The design of ViVA, the Virtual Vector Architecture, which is a new extension to traditional microprocessors that allows them to take advantage of vector-style memory instructions. Vector transfers occur between a new on-chip buffer and DRAM; individual elements from the buffer can be transferred to the processor core, which can operate on them.
- A comparison showing the similarities and differences between SIMD extensions and true vector processing, and a discussion of the implications of those differences.
- The design and description of a technique that allows existing microprocessors with SIMD extensions to convert their instruction sets to true vector ISAs in a simple manner. My approach details the steps needed to extend SIMD instructions in a way that allows initial vector implementations to be low cost — virtually the same cost

as the original SIMD implementation — but easily extendable to higher performance and greater latency tolerance for future implementations.

- The modification of a cycle-accurate full-system simulator to allow it to be easily extended at runtime. The modifications allow new instructions to be added, new physical state to be created and associated with existing system objects, and simulator execution flow to be observed and modified.
- The evaluation of ViVA's performance in a number of different configurations, using the modified simulator to execute code that represents the dwarfs that are most relevant to my work.

1.4 Overview of Vectors

The key idea behind vectors is to collect a set of data elements in memory, place them into large register files, operate on them, and then store them back. Vector register files act as software-controlled buffers, which are large enough to allow many memory operations in flight. Vector instructions are a compact way of capturing data-level parallelism, and are a simple, scalable way to organize large amounts of computation [LD97].

Vector processing gets its name because, at its heart, it is designed to work on long one-dimensional arrays, or vectors, of data. The register file is designed so that a single logical named register holds a number of elements: 64 double-precision (eight byte) values is typical, although shorter and longer vectors have been used. The entire architecture is designed to take advantage of the vector style in organizing data.

In a scalar computer, a single instruction specifies an operation on individual words, as well as input and output registers that hold those words. An instruction will typically have two sources and a single destination. Vector instructions also specify only a single operation, but the operation is applied to many elements: the sources and destination are now vectors, and the operation is applied to successive elements of each, as shown in

Figure 1.1. In a vector add operation, the first element of each source is added and stored in the first element of the destination; then the second element of each source is added and stored in the second element of the destination, and so on, for the full length of the vectors. Figure 1.2 shows the vector and scalar code to perform a series of 64 adds. The scalar processor will have to fetch, decode, and issue hundreds of instructions to execute the code; the vector processor will have to issue just five instructions.

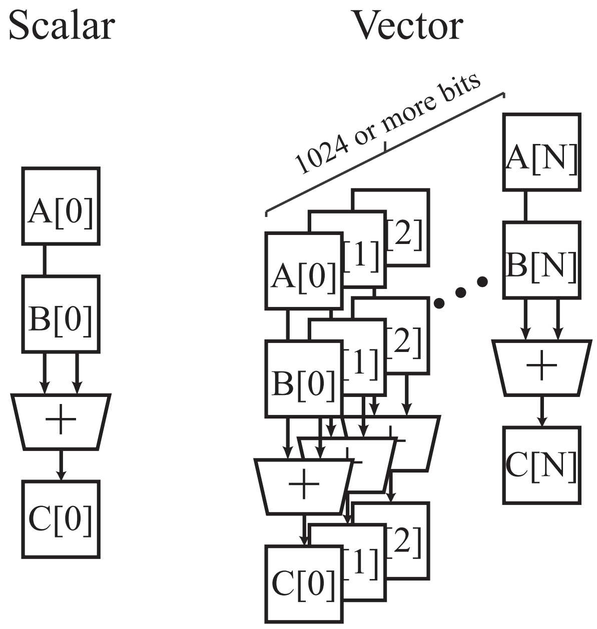


Figure 1.1: Operation of a scalar and vector “add” instruction.

Because vector processors are designed to operate on vectors, they typically include only data processing (load, store, and arithmetic) operations; control operations are executed in a separate scalar processor. Thus, vector computers typically contain at least two processors, a scalar and a vector processor, that operate as co-processors.

Vector instruction sets contain typical operations, but the fact that the instructions specify multiple operations necessitates some changes and differences from scalar instruction sets. First, programs need to know the length of the physical vector registers. It would be possible to fix the length in the vector ISA, but that is inflexible and unnecessary; instead, vector processors typically include a control register called “Maximum Vector Length”

(MVL), which is a read-only register that contains the greatest number of elements that can be stored in a single vector. Because programs read hardware vector lengths from a register instead of relying on a fixed value, processor designers have the flexibility of using different hardware vector register lengths across processor generations without changing the ISA. In fact, the same binary executable can run on the different implementations, transparently taking advantage of the greater hardware resources. MVL is set by the hardware; its software counterpart is the Vector Length (VL) register, which allows programs to specify a logical vector length no longer than MVL for which the processor should operate on vectors. By writing a shorter value to VL, vector programs can execute programs with short vectors. Vector ISAs usually guarantee a minimum MVL, so that applications using vectors shorter than that value can execute without checking hardware vector lengths.

```
       mtvcr  $vl, 64         # set VL
       vld    $v0, A          # load values
       vld    $v1, B          # load values
       vadd   $v2, $v0, $v1   # add
       vst    $v2, C          # store values
```

(a)

```
       addi   $r1, $r0, 0     # reset counter
loop:  ld     $r2, $r1, A     # load value
       ld     $r3, $r1, B     # load value
       add    $r4, $r2, $r3   # add
       st     $r4, $r1, C     # store value
       addi   $r1, $r1, 1     # increment counter
       cmpi   $r1, 64         # compare
       bne    loop            # loop back
```

(b)

Figure 1.2: Code to perform a series of adds. (a) shows the vector form of the code; "mtvcr" sets a vector control register value, in this case, VL. (b) shows the scalar code. Strip mining, loop unrolling, and other techniques omitted.

Because applications may want to execute on vectors that are longer than MVL, compilers vectorize loops using a technique called strip mining. In strip mining, multiple loop iterations of full-MVL instructions are executed, and the remaining amount is compared to MVL; once the remaining length is below the length of physical registers, VL is set to the remaining amount and a final loop is executed. There are a number of techniques that can simplify strip mining, including saturating VL registers or instructions.

Another important difference between scalar and vector processors is in memory operations. Vector computers can load a block of contiguous elements — called a unit-stride load — and they can accesses elements in two other ways. They can perform strided operations, which are loads or stores of a series of elements with a constant inter-element displacement, as well as indexed (sometimes called scatter-gather) operations, where an individual index is given for each element to be accessed. Figure 1.3 summarizes the memory access patterns. Generally, the locations to access in an indexed operation are given with a base memory location specified in a scalar or control register, and offsets given in a vector register. For strided operations, the stride is generally given as a value in a control or scalar register.

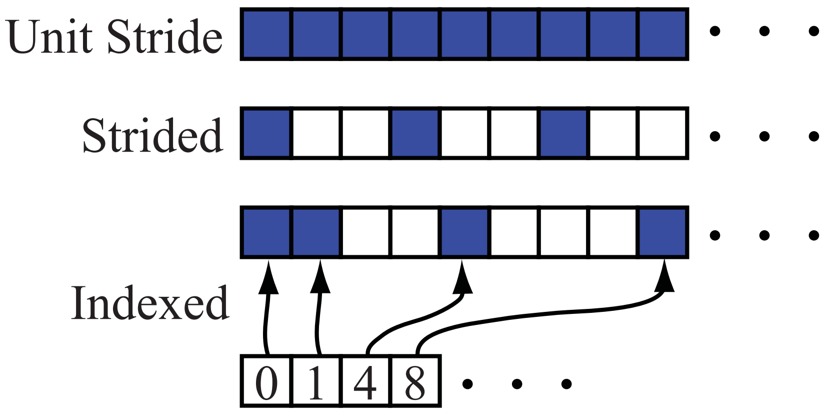


Figure 1.3: Illustration of different vector memory operations. Dark spaces represent accessed locations.

Unit-stride accesses are useful for operating on large chunks of data, or streaming accesses. Often, vector processors have different instructions for unit-stride and strided accesses, even though a strided instruction using a stride of one can perform unit-stride operations. Identifying unit-stride operations separately sometimes allows the processor to perform optimizations specific to that instruction.

Generally, vector memory operations operate under a weak memory consistency model, which means that the scalar and vector processors may observe loads and stores from the other unit happening out of order. Vector processors are generally designed to optimize memory system performance; offering few automatic consistency guarantees allows the memory system to be simpler, and take advantage of multiple memory units. A memory barrier (or fence) is usually required to guarantee ordering of any sort between scalar and vector operations, including loads and stores. The exception is if there is a true data dependency carried through registers, that will force the processor to delay executing the later instruction until the data is available from the earlier instruction.

Vector execution is designed to operate efficiently by pipelining a series of operations. Conditional statements — statements that depend on the result of an "if" — can cause problems for pipelined vector operations. One way that vector processors can deal with conditional statements in loops is to use masked execution. The result of the conditional statement is stored in a flag register, which contains one bit of information per element. When the processor executes conditional statements, the flag is interpreted as a mask — elements with a corresponding "1" bit in the flag register are execution, and elements with a "0" bit are not.

Flag registers can serve other purposes, as well. Much as in scalar processors, flags can show status or characteristics — for example, sticky floating-point exception or subnormal flag registers can record elements that might need extra processing. Vector architectures can have general-purpose flag registers, used for conditional execution, as well as purposed flag registers, that record specific conditions or exception types. Multiple general-purpose flag registers are useful for temporarily storing the results of multiple comparisons, or for building up a complex condition from multiple simpler conditional statements [SFS00].

Since a vector instruction executes the same type of operation many times, there is a simple way for vector processors to improve execution performance: put multiple pipelines in each functional unit, so that a single functional unit can begin execution of multiple operations per cycle. Multi-pipeline datapath organizations lend themselves to a simple way of designing scalable performance: designers create a small chunk of the full datapath that contains part of the vector register file and a simple pipeline for each functional unit. That chunk, called a “lane” or “pipe”, can be replicated with minimal changes in order to scale performance. During instruction execution, a pipeline in each lane will begin execution on another element; all of the elements from a vector instruction that begin execution together are called an element group. Low-cost implementations can have a single lane; larger implementations can replicate the lane multiple times to achieve the desired performance. Figure 1.4 shows a vector datapath, and highlights a lane, functional unit, and pipeline.

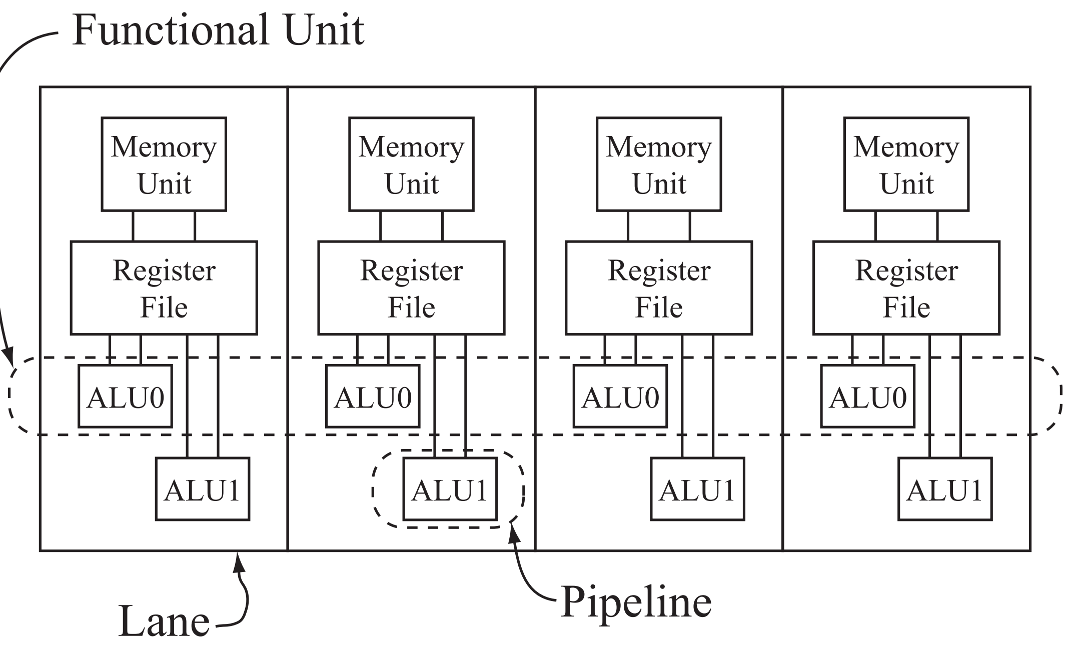


Figure 1.4: Vector datapath, with components labeled.

1.5 Benefits of Vectors

Vectors have been used for a long time, in many different situations. The reason that vector technology has survived for decades in an ever-changing field is that vectors offer many benefits over other alternatives. Specifically, vectors are:

- compact, describing many operation in a single instruction;

- expressive, allowing application to describe useful characteristics about a group of operations;
- scalable, in both processing and latency tolerance;
- latency tolerant;
- easily programmable; and
- power-efficient.

Each of these benefits is described more below.

1.5.1 Compact

The basic design of vector instructions, that a single instruction specifies many operations, leads directly to many benefits on its own. For one, less issue bandwidth is required: when performing a fixed number of operations, fewer instructions will have to be issued on a vector computer than on a scalar computer. Additionally, vector programs can operate on a number of elements without much of the extra overhead that is typically required in scalar programs: vectors allow the application to describe a looping operation implicitly. Originally, the reduced instruction bandwidth was a huge benefit for systems with small or no instruction caches; nowadays, the actual fetching of instructions is not necessarily a large impediment to performance. Yet, the same feature has become important for another reason: fewer instructions to fetch, issue, and decode can lead to significant power savings. Several studies reporting measured or simulated power consumption state that instruction fetching and decoding can range from 18 to 30 percent of dynamic power on superscalar processors [MKG98, BTM00, TM05, SMR$^+$03, GBJ98].

There are a number of other benefits from the way vector instructions encode multiple operations. Separate operations within a vector instruction all share the same source and destination registers, and are all the same sort of operation; therefore, dependency checking, required for interlocking in a pipelined processor and for proper dispatch in an out-of-order processor, only has to be done once for the vector instruction, not once per operation. If a program is mainly composed of vector instruction, each of which consists of n operations, the total number of dependency checks that has to be done is reduced by n^2 (assuming that the total number of operations in flight at any time is the same).

Vector instructions' ability to pack multiple operations in a single instruction gives a potential benefit to performance, as well. Out-of-order processors rely on being able to search through a number of upcoming instructions, and executing them as early as possible. If the processor is not able to find instructions that are able to run, performance is diminished significantly. Consequently, instruction window size — that is, the number of upcoming instructions that the processor is able to examine — can be an important performance limitation [LKL$^+$02]. Unfortunately, instruction window size cannot be arbitrarily increased: it is limited by the size of queues of upcoming instructions, which are limited by power and chip area. As relative memory access time increases, the average lifetime of instructions (measured in processor clock cycles) increases, further putting pressure on instruction windows [KS02]. Since vector instructions need only take a single slot in the instruction window, a window of a certain size can represent more vector operations than scalar operations.

1.5.2 Expressive

Vector operations are expressive in many ways that scalar operations are not.

Vector ISAs include a number of limitations about what sort of operations can be encoded in a single vector instruction. In practice, those limitations do not pose much difficulty for producing vectorized code; on the other hand, they provide a number of guarantees that the processor is able to use.

A processor knows that a single vector instruction is going to encode many operations; it knows that they will all be the same sort of operation, which simplifies instruction issue and datapath organization, and it knows that all of the operations encoded in the instruction will be independent, which simplifies the execution of the program.

Vector memory instructions allow a program to describe its access patterns explicitly: on a unit-stride or strided load, the processor knows precisely which elements will be needed, and can use that information to begin address translation or memory accesses early. Scalar processors try to take advantage of memory access patterns with hardware prefetching units, but they are forced to try to discern future access patterns of a single stream by looking at all recent past accesses, an error-prone, power-hungry, complex proposition. Scalar prefetch instructions generally allow an application to express a need for particular memory elements in the future, but usually do not encode information about the general access pattern of multiple elements.

Finally, processors know that vector instructions will access the register file in simple, fixed patterns. An implementation can take advantage of the access pattern by partitioning a full vector register file, so that a datapath in a vector processor with multiple pipelines per functional unit only has to store the particular elements it will be accessing. Furthermore, the vector processor can take advantage of a regular access pattern in order to simplify the design of the register file: instead of requiring multiple access ports, a register file can instead have a single wide access port. Figure 1.5 shows an example. On any register file access, the register file transfers the requested element and a number of neighboring elements to a smaller fast buffer. On subsequent accesses to the neighboring elements — which are very likely to occur, because of the vector instruction register file access pattern — elements can be transferred from the buffer, freeing up use of the register file access port. In this way, a single access port is multiplexed between all of the functional units that need it. On a conflict, when functional units need multiple elements during the same cycle that are not in the fast buffer, one access can stall for a cycle.

1.5.3 Scalable

As mentioned above, vector computers can naturally scale in two dimensions: in the length of the physical vector registers, and in the number of lanes or pipelines in a functional unit.

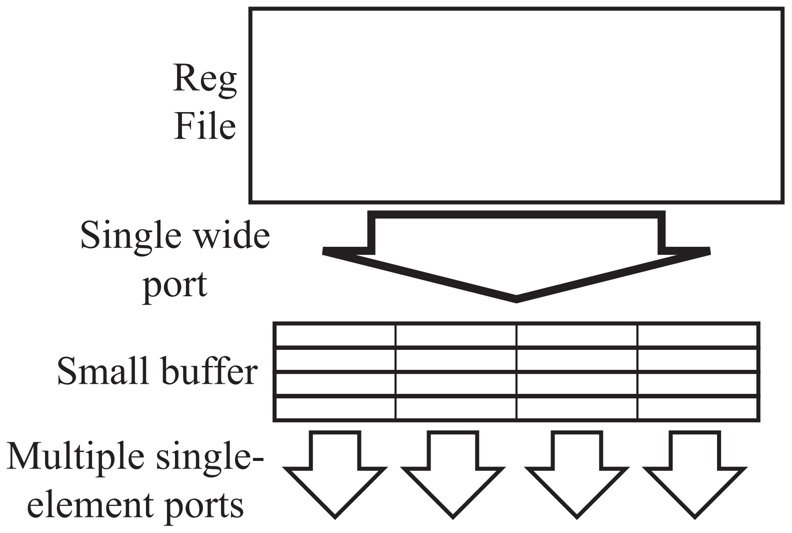


Figure 1.5: The larger register file has a single port that transfers four consecutive elements to a small buffer each cycle. Functional units access one of the smaller single-element ports of the small buffer. Keeping the large register file to a single port reduces complexity and area.

Scaling the length of vector registers gives a benefit in that any cost that is amortized over the length of a vector will then be spread over a greater number of elements. For example, if load instructions have to wait for memory latency to get the first result, but then get a result every cycle after that, the effective per-element latency cost will be less for longer registers. Increasing the lengths of vector registers also means that the number of operations represented by a single instruction increases, resulting in greater benefits from reduced number of instructions to fetch, decode, and issue.

On the other hand, longer vector registers only give a benefit if programs can take advantage of them. If vectors become too long, programs might never set the VL to a full MVL length, and the extra storage will be wasted.

Scaling lanes can lead to a similar problem: if the number of lanes is greater than the typical vector, the extra lanes will go to waste. In the case of lanes, what is wasted can be very costly: the chip area dedicated to a full lane is quite a bit more than the area dedicated to longer vector registers, and the associated power consumption will likely be quite a bit more, as well. Having a greater number of lanes can cause additional waste on longer vectors, as well: if vector lengths in a program are not an integer multiple of the number of lanes, the final element group will not use all of the lanes. With a smaller number of lanes, there is more of a chance that they will all be used, and during the cases when there are unused lanes in an element group, there is a better chance that the number of wasted lanes will be smaller.

One nice feature about both of the types of scaling that are possible with vector computers is that they are transparent to applications. A single binary is able to run on a vector computer with short vector registers and a single lane, as well as on a computer with very long registers and multiple lanes. The lanes are purely a performance feature, with no directly visible implications for the application; the length of vector register is exposed through the MVL control register, but applications usually do not care about the particular value, they simply use the number in order to control loop indices.

1.5.4 Latency tolerant

Vectors tolerate latency. That is critically important, especially as relative latency to memory keeps increasing. Vectors tolerate latency in two primary ways: first, by amortizing costs over the entire length of a vector, and secondly, by providing enough state to facilitate significant concurrency.

Latency amortization is straightforward: any per-instruction costs will have a lower per-element cost, as the number of elements per instruction increases.

The latency tolerance afforded by extra state is related to Little's Law [Lit61], which is a queuing theory law that relates to single-server queues with customers arriving, being served, and leaving. It is usually given as: $N = AT$, where N is the average number of waiting customers, A is the average arrival rate, and T is the average wait time. A consequence of Little's Law [Bai97] is that concurrency in a system is equal to bandwidth times latency. This means that if we want to maintain a certain bandwidth, the amount of concurrency we need is proportional to the latency in the system. Another way to look at it is that, for a given latency, the amount of bandwidth we can achieve is limited by the amount of concurrency we can exhibit. Vectors are a simple way to keep many operations in flight, since a relatively small number of instructions corresponds to many operations, and thus vectors are generally more able to tolerate memory latency than scalar computers.

1.5.5 Easily programmable

Vector computers have been around since the early 1970's [EVS98], which has given them ample time to be well-analyzed and understood. Vector compiler technology is mature; as opposed to many architectural innovations, where it takes years for compilers to be able to deliver performance anywhere near what is theoretically possible, vector compilers are able to produce high-quality vector code now.

Vectors also work well with programmers. The vector model is a natural way to think about data-oriented loops: instead of a single element being processed in the program, it is generally simple for programmers to think of the same algorithm working on chunk of data. In addition, vectors give a benefit when attempting to achieve good performance: vector compilers describe which loops vectorized, and are often able to given feedback about the loops that did not. Programmers then just look at the report, and know to focus their attention on the loops that the compiler was not able to handle.

Vectors are also portable, in the sense that a program that vectorizes well on one vector computer will likely vectorize well on another vector computer.

1.5.6 Power-Efficient

Vectors derive performance from the way they process multiple elements of data: the vector paradigm amortizes costs and allows designers to optimize power. The key idea is that vectors are an excellent match to take advantage of data-level parallelism.

There are many different forms of parallelism. Recent out-of-order superscalar microprocessors focus on taking advantage of the least-restrictive type: instruction-level parallelism. Machines such as Sun's Niagara processor are designed to use a slightly more restrictive form of parallelism, thread-level parallelism, and vector machines are a good example of taking advantage of the most restrictive form of parallelism, data-level parallelism.

As the form of parallelism becomes more restrictive, from instruction-level through data-level, it becomes less flexible: a machine that takes advantage of instruction-level parallelism can also take advantage of parallelism expressed in the other forms. Data-level parallelism is the most restrictive of all the forms.

But, with restrictions comes simplicity. Data-level parallelism is the simplest form of parallelism to take advantage of, and instruction-level parallelism is the most complex.

In the era where power consumption did not matter, the only limitation on parallelism was the effort that designers were willing to put into the processors, and instruction-level parallelism was king. Now that power is an important constraint, the situation is different: we are forced to confront an engineering tradeoff, and evaluate the benefits of moving to more flexible types of parallelism and the costs of doing so.

As it turns out, the additional benefits offered by the more flexible forms of parallelism are often not all that much more than what is offered by data-level parallelism. Put another way, data-level parallelism is an effective way for programs to express parallelism, and a simple form of parallelism for processors to use.

A good example of the efficiency of data-level parallelism as compared to instruction-level parallelism is in the organization of datapaths and pipelines. In an out-of-order superscalar processor, the processor must check for dependencies between every operation and every other operation in flight. This limits the number of operations that can be in flight, so these processors usually are limited to a relatively small number of functional units and operations that can be launched per cycle. To achieve performance, they generally focus on clocking the processor as fast as possible, to achieve an overall high processing rate. Vector pipelines are usually organized with multiple lanes, and can be thought of as wide, slow execution units. Since dependency checks only have to be performed per instruction, not per element, the number of elements launched per cycle can be large, while still maintaining a simple datapath organization.

1.5.7 Other Benefits

In addition to the above, there are a number of other benefits that vector computers usually offer.

Vector register files are designed to hold multiple elements, so they tend to be more flexible than scalar register files: an eight-byte block of register storage can be interpreted as a double-precision floating point value, two single-precision values, four 2-byte integer values, or 8 byte values (depending on exactly what the processor allows).

The vector register file is a block of storage that the program has control over: as opposed to a cache, which gives no guarantees as to how long it will retain elements, the vector register file acts as a software-controlled buffer, and offers the same benefits of explicit control.

1.6 Misconceptions about Vectors

Because vector computers have been around for such a long time, there are a number of common misconceptions about them. Some of the misconceptions were true for early vector machines, and have been addressed; some of them were never true.

1.6.1 Misconception: Vector context switch time is too large

Vector register files do contain more data than scalar register file; that is a big part of why they are able to tolerate memory latency. Consequently, it can take longer to store and re-load all of the data in their register files, as you may have to do on a context switch. Nevertheless, it turns out that the reality is not that bad in practice.

First of all, vector computers perform some optimizations to reduce the amount of unnecessary data saved. A simple approach, used in the VAX Vector Architecture [BB90], is to keep track of which applications use the vector unit. When a vector application swaps out, the vector unit is disabled, but none of its state is immediately saved. If another application attempts to use the vector unit, it throws an exception; at that time, the old vector state is saved. Thus, if no other vector application runs between the time that a vector application swaps out and when it swaps back in, no vector state will have to be stored and restored.

Another optimization, used by the IBM System/370 Vector Architecture [PMSB88], uses "Vector In-Use" bits to reduce the number of vector registers that have to be stored and restored. Each vector register has a single associated bit that is set to "0" on program startup. As the program uses a register, its in-use bit is set to "1"; bits can be reset by the application to declare that a register is not currently needed. On a context switch, only registers with a "1" in-use bit are saved and restored.

Finally, the system can keep track of the largest value of Vector Length that the application has used. Only register elements up to that length need to be saved [Koz99].

Beyond specific optimizations for reducing vector context switch times, though, there is a larger point: all programs have a working set that they need to access to perform well. In vector applications, much of that set will be stored in vector registers; in scalar applications, much of the working set will be accessed in the L1 cache. Until that set is loaded — vector registers or L1 cache — the program will not be able to execute at high speed. Since L1 caches are typically about the same size as vector register files, context switch times — including the time before the program is able to execute at high speed — will likely not be extremely different between scalar and vector computers. Additionally, vector computers are generally tolerant of latency, so they are typically better able to make program progress while executing saves and restores than scalar computers. Furthermore, bandwidth is expected to improve faster than latency [Pat04], so it makes sense to use a technology that may be bandwidth hungry on context switches, but latency tolerant for most processing.

1.6.2 Misconception: Vector register file takes up too much chip area

Vector register files are much larger than scalar register files, but in a modern chip, the size of the vector register file will typically be tiny. A typical vector register file might be 16 kilobytes; on-chip L2 caches on many microprocessors are multiple megabytes. For simpler cores with small or no caches, the relative size of the vector register file will be more. But, in order to achieve tolerance of memory latency, a certain amount of storage is required. If vector register files were smaller (without a commensurate increase in storage elsewhere), the system would not be able to tolerate as much latency.

Vector register file complexity can be much lower than the complexity of scalar register files on modern superscalar microprocessors. Vector register files can be partitioned, and can use wide, slower access ports. Scalar register files may be very complex, to support fast access to multiple arbitrary elements.

1.6.3 Misconception: Vectors are only good for scientific applications

Vectors traditionally have been extremely popular for scientific applications, but in more recent years a number of other areas have turned to vector processing. In particular, media and DSP applications, two areas increasing in popularity, are good matches for vector technology [EVS98, KP02].

1.6.4 Misconception: Vector computers only work well if applications have extremely long vectors

The earliest vector computers had an important limitation in processing vectors: a vector instruction that used the result of a previous vector instruction could not begin processing until the earlier instruction was completely finished. The lack of forwarding meant that vector operations could have a large start-up time, and it only made sense to use vector operations for long vectors.

Starting with the Cray-1 [Rus78], vector computers could forward data from one functional unit immediately to another — known as “chaining” for vectors. Chaining significantly reduces vector start-up time, and can make it more efficient to use vectors, rather than scalar operations, for loops as small as a few elements.

1.6.5 Misconception: Vectors memory operations only work with contiguous blocks of memory

Vector computers work on large chunks of data, but the data does not just have to be allocated in memory as a single large chunk. Vectors use strided accesses — where the elements are spaced apart a constant displacement in memory — and indexed (scatter-gather) accesses — where the each individual element location in memory is given by an index vector — to take advantage of other memory access patterns. Strided and indexed accesses are usually not as efficient as unit-stride accesses, but they are typically more efficient on vector computers than on scalar computers because a vector program is able to describe its access patterns to the processor explicitly.

1.7 Summary

Now is a time of change in the microprocessor industry. The previous trend of boosting single-thread performance by increasing clock rate has come to an end: processors were getting increasingly small amounts of benefit and consuming ever greater amounts of power.

In this dissertation, I investigate a technology that can greatly improve the efficiency of microprocessors: vectors. Vectors are an established idea with mature compilers that important applications can use to take advantage of data-level parallelism effectively. I focus on techniques that manufacturers can use to easily add vectors. The first is an evolutionary technique that describes how to use SIMD extensions for true vector processing, and the second adds a data buffer that allows processors to get the advantage of vector memory operations without a full vector implementation.

In the next chapter, I cover related work that led to my research.

Chapter 2

Background and Related Work on Vectors

In many ways, this work is an extension of previous work and projects. Many of the ideas developed here came through investigating questions raised in earlier work, in particular: T0 [ABI$^+$96], VIRAM [Koz99], and Code [Koz02], all described below. These, in turn, are based on previous work done at other institutions.

This chapter describes the previous work. It describes the projects, and includes lessons learned. Furthermore, it describes why those projects motivate the current work: what questions were raised that led to the ideas that I explore? Additionally, this chapter reviews some of the core ideas that my work uses.

2.1 SIMD

SIMD technology is often called "vectors", but there are many important differences between microprocessor SIMD extensions and true vector implementations. Some of the basic ideas are shared, but SIMD extensions lack some critical features, and have quantitative shortcomings in the ways that they are similar to vector computing.

The term "SIMD" was first coined by Flynn [Fly66], as part of a taxonomy categorizing computers by style of processing instructions and data elements. Table 2.1 shows the full table, along with descriptions and examples. The original intent and usage was for categorizing computers: a SIMD computer uses a single instruction to operate on multiple pieces of data, thus exploiting data-level parallelism to amortize instruction processing costs over multiple elements. By the early 1990's, microprocessors begin adding extensions to existing microprocessors that allowed execution of new SIMD instructions [PW96], generally to improve performance on media applications. Eventually, most microprocessor manufacturers had created SIMD extensions [SJV04].

		Data	
		Single	Multiple
Instructions	Single	SISD Most simple microprocessors	MIMD Multiprocessors
	Multiple	MISD Rare; used to describe shared functional units [AFR67]	SIMD Classically: ILLIAC IV [BDM$^+$72]; Modern usage: Intel MMX extensions [PW96]

Table 2.1: Flynn's taxonomy.

Microprocessor SIMD extensions usually involve adding a new set of register that are either 64 or 128 bits wide, as well as a new set of instructions that operate on the registers. The instructions typically operate on integer data of shorter (8- or 16-bit) lengths, although some are more restricted and many operate on longer data. Some are also able to operate on floating point elements.

The instructions often focus on media applications, and so include a number of features that are typical of DSPs: saturating arithmetic, processing media data types, and so on. Generally, SIMD instructions operate on two input registers, element-pairwise: an operation is applied to the first element of the input registers and produces a result in the first element of the output register, and then the same operation is applied to the second element of the input registers, and so on. Some SIMD extensions have included instructions that include non-pairwise operations, such as a random shuffle that allows elements of a register to be placed at an arbitrary location in the output register; some SIMD extensions have even included operations between elements in a single SIMD register.

SIMD extensions and vectors share some similarities. Both are models of computation that attempt to take advantage of data-level parallelism in the same way: a single instruction specifies one type of operation that is applied to multiple independent elements, logically stored in a single long register. In fact, SIMD execution can be a way to achieve significant performance improvements over regular scalar execution, particularly on some media applications.

Yet, SIMD extensions are not true vector processing. In fact, SIMD extensions tend to have a number of shortcomings in the ways that they are similar to vector processing, and vectors have additional features that offer significant benefits over SIMD:

- SIMD extensions are too short to attain a significant benefit from amortizing long-latency operations;
- SIMD extensions typically have restrictive alignment requirements;
- SIMD instructions cannot operate on partial vectors: programs can only operate on the entire register length (with, possibly, the option to operate on a single element), instead of being able to use VL to specify arbitrary lengths;
- SIMD extensions typically have a limited means of conditional execution, or no means at all;
- SIMD ISAs often do not exhibit orthogonality, in terms of which operations can be applied to what type and size of data elements;

- SIMD operations are not scalable to greater amounts of data-level parallelism: physical vector lengths are implicitly fixed, and there is no equivalent to MVL to allow them to grow;
- the only way to increase data-level parallelism in SIMD ISAs is to add new instructions with every increase.

Chapter 3 explores ways to improve existing SIMD extensions, or design new extensions, that do not suffer from these drawbacks, and give the benefits of full vector execution.

2.2 Cray X1

The Cray X1 is a recent vector supercomputer. It builds on the design of traditional vector computers, but it has many new features as well. Figure 2.1 shows a diagram of the basic processor that the X1 is made from, the MSP (Multi-Streaming Processor). [BCBY04]

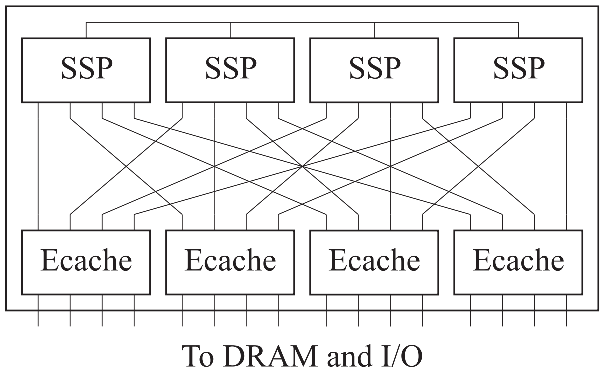


Figure 2.1: Cray X1 MSP (Multi-Streaming Processor). Shown are the four SSPs (Single-Streaming Processors) and a total of 2 MB of cache that make up a single MSP.

A single node consists of four MSPs and 16 GB of DRAM. The MSP contains four SSPs (Single-Streaming Processors), each of which is contains two vector lanes and a scalar processor. [DFWW03] Additionally, the MSP contains 2 MB of cache [AAA$^+$04]. The X1 illustrates that vector computers can work well with caches; in fact, much of the computer's design reflects the influence of the cache. For example, the full amount of memory bandwidth in each node is 200 GB/s [Tan02], while the peak processing rate of 12.8 GFlop/sec on 64-bit operations per MSP would require over 1200 GB/s; the reason for the apparent design imbalance is that, in contrast to traditional vector computer, the caches are expected to take advantage of reuse and deliver higher bandwidth to vector applications.

2.3 IRAM

IRAM stands for "Intelligent Ram" [PAC$^+$97]. IRAM is based on the idea of addressing a critical problem, the processor-memory performance gap, by using embedded DRAM.

Processor performance has been increasing at a much faster rate than memory performance. This situation has been called the "memory wall" [WM95, McK04]. As the relative performance of DRAM, compared to processing speed, increases, average memory access time continues to take more cycles. Overcoming memory latency becomes more difficult, until the point where the processor is spending most of its time waiting for memory, instead of actually doing useful work.

IRAM addresses the processor-memory performance gap by including DRAM on the processor die — in fact, as the amount of on-chip memory increases, it overwhelms the amount of processing, and the chip looks more like a memory chip with a small area dedicated to processing, instead of the other way around.

Placing memory and processing on the same die can have a number of advantages. Since signals stay on-die, there are no pin or board trace constraints, and bandwidth can be much larger. Additionally, it is easy to add on-chip DRAM modules in parallel, and use the bandwidth from all of them. Latency also improves, simply because the processor does not have to drive signals through high-capacitance off-chip drivers or board traces. Physical lengths are also reduced, further cutting back latency.

Locating processing and memory on the same die also gives energy benefits. Driving signals off chip and through board traces requires a relatively large amount of energy, which can be avoided if the signals never leave the chip. Off-chip memory typically takes the form of standard packages; on-chip memory can be more flexible, and directly take advantage of any module configuration. Finally, reducing external memory can reduce the total board space, which can be a significant cost advantage, and can be critical for small (hand-held or portable) devices.

Traditional microprocessors are designed to take advantage of traditional memory organizations: they will work in an IRAM process, but they might not fully take advantage of the potential benefits of embedded DRAM. Additionally, traditional microprocessors spend a huge amount of power and area dealing with drawbacks of conventional memory — drawbacks that are no longer as critical with embedded DRAM. By reconsidering the organization of a microprocessor in an IRAM process, we can design something that is a better match for what embedded DRAM offers.

One example of a microprocessor organization that is designed specifically for IRAM is VIRAM, a design approach that combines vector microprocessors with embedded DRAM. We implemented a VIRAM processor that takes advantage of a characteristic that is a good match for embedded DRAM and vectors: low-power execution. Embedded DRAM can save significant power over off-chip DRAM, and vector processing requires only simple (and therefore low-power) control logic [LSCJ06].

The design of VIRAM1 started in earnest in the summer of 2000, and taped out in late 2002 [GWKP04]. Table 2.2 shows details of the design.

We developed VIRAM1 using a combination of different techniques: hard macros (complete blocks, containing physical representations of transistors and wires, all placed appropriately) for the DRAM, soft macros (logical descriptions in Verilog) for the MIPS M5KC core, macros created by a generator (a tool designed to customize logic blocks) for SRAM, custom logic, and hand layout. We were able to combine so many different sources because we used an industrial-strength design process, combining standard synthesis, layout, extraction, and place-and-route tools.

The EEMBC benchmarks [EEM07] include suites targeted towards automotive appli-

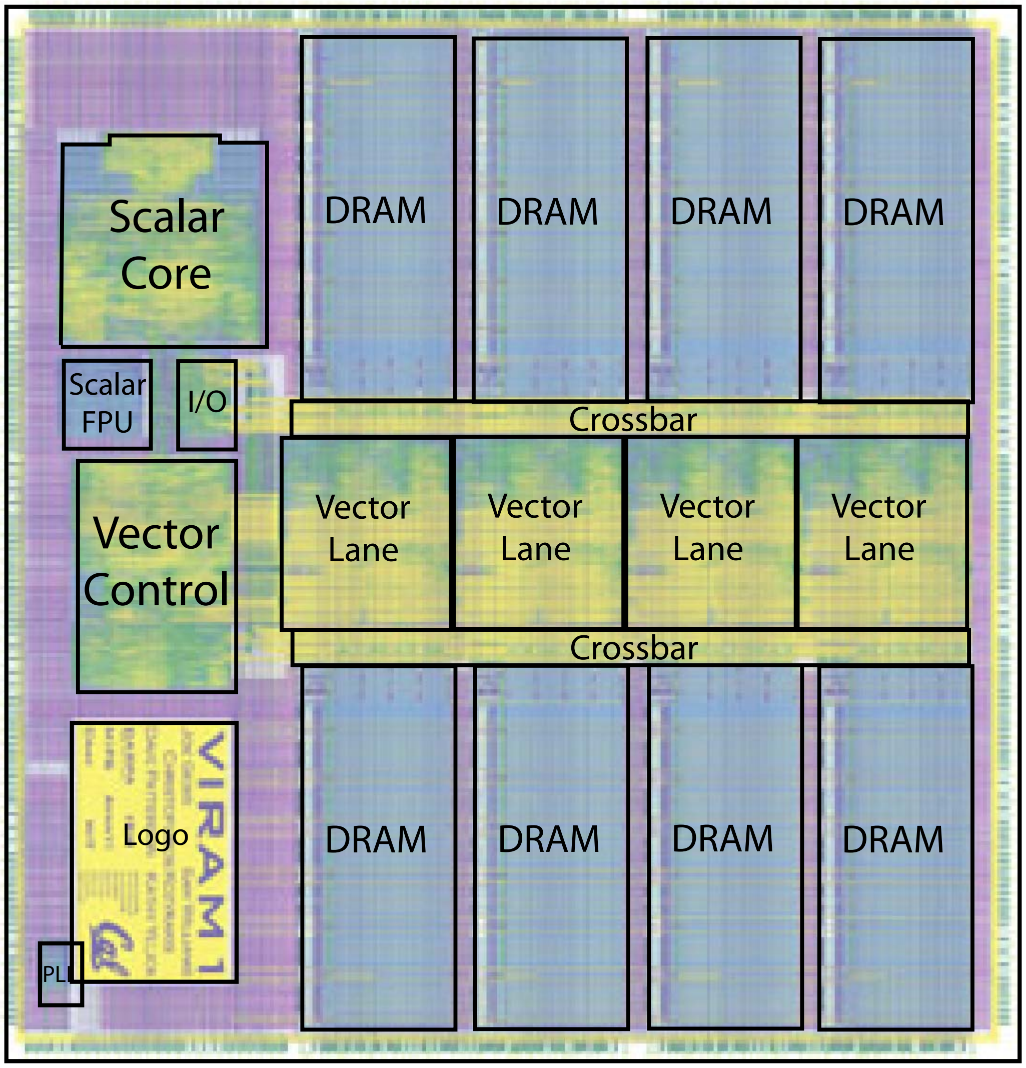


Figure 2.2: Floorplan of VIRAM1.

cations, telecom areas, office automation, and so on. Two of the areas are particularly well-suited to evaluating performance of VIRAM1: the Consumer category and the Telecommunications category, both of which contained media applications of the sort the VIRAM1 was designed to run efficiently [Lev00].

Both the consumer and telecommunications benchmark suites use only fixed-point data, and generally use data at narrow widths (8 or 16 bits). EEMBC measures the performance for an individual application by recording repeated iterations per second; an overall score

Clock speed	200 MHz
Total area	332.6 mm^2
Number of transistors	$>$ 125 million
Power consumption	$\sim$2 Watts
Integer performance: 16b	6.4 GOPS
Integer performance: 32b	3.2 GOPS
Integer performance: 64b	1.6 GOPS
Floating-point (multiply-add) performance: 32b	3.2 GFLOPS
Floating-point (multiply-add) performance: 64b	0.8 GFLOPS
Floating-point (multiply) performance: 32b	0.4 GFLOPS
Floating-point (multiply) performance: 64b	1.6 GFLOPS
Fixed-point performance: 16b	9.6 GOPS
Fixed-point performance: 32b	4.8 GOPS
Fixed-point performance: 64b	2.4 GOPS

Table 2.2: VIRAM1 statistics. Data from [GWKP04].

(ConsumerMarks or TeleMarks) is generated by calculating a geometric mean of the individual scores. Performance can be reported in an out-of-the-box mode, where binaries are generated by compiling the original benchmark code, or in an optimized mode, where the source code (but not algorithm) can be changed.

We compared the performance of VIRAM1 on the Consumer and Telecommunications suites to a number of other processors. Table 2.3 gives details of the different processors, and Table 2.4 gives results as reported in [Koz02] and [KP02].

Processor	Architecture	Processor	Issue Width	Execution Style	Clock Freq.	Power
VIRAM1	Vector	VIRAM	1	in order	200 MHz	2.0 W
x86	CISC	K6-III+	2 (6)	out of order	550 MHz	21.6 W
PowerPC	RISC	MPC7455	4	out of order	1000 MHz	21.3 W
MIPS	RISC	VR5000	2	in order	250 MHz	5.0 W

Table 2.3: Details of processors used to compare to VIRAM1 performance. The K6-III+ is able to issue two CISC instructions simultaneously, which get broken into a maximum of six microoperations.

VIRAM1 was a successful demonstration. It validated our claims that vectors and embedded DRAM could be combined to produce a simple, low power media processor capable of significant processing power. We showed that a small team of university graduate students was capable of using the VIRAM design style to produce an interesting portable

Chip	ConsumerMark Score	TeleMark Score
VIRAM1 (cc)	81.2	12.4
VIRAM1 (as)	201.4	61.7
K6-III+	34.2	8.7
MPC7455	122.6	27.2
VR5000	14.5	2.0

Table 2.4: IRAM performance comparison for EEMBC Consumer and Telecommunications suites. VIRAM1 performance shown for both compiled (using re-targeted existing vector compiler) and hand-optimized code. TM1300 performance shown for both direct compilation and hand optimization. Data from [Koz02].

media processor.

We also discovered some limitations. For the applications we looked at, the vector lengths could not be increased much beyond the VIRAM1 maximum vector length: we would not see much performance benefit by simply increasing the number of vector lanes. The cross-lane control signals limited our clock rate, which meant that more or larger lanes would result in a slower clock cycle.

VIRAM1 was a useful exercise. In many ways, though, VIRAM1 raised more questions than it answered. VIRAM1 was designed for low-power media applications; how applicable were the ideas to other design targets? Were the limitations inherent in the VIRAM design style, or is it possible that other implementations would not suffer from them? What could we change to reduce the effect of those limitations?

2.4 T0 and CODE

T0 and CODE are two vector microprocessor research projects related to VIRAM. T0 was developed before IRAM, and helped develop many of the ideas that were implemented in VIRAM1. CODE was created after VIRAM1, and addressed several of the issues that were discovered during the creation and testing of the IRAM chip.

T0, or Torrent 0, was born out of a series of projects designed to be used for speech recognition [Asa87]. The goal of the project was to develop a workstation accelerator card that would provide a cheap way to add significant amounts of processing for performing speech recognition.

T0 was designed only to support fixed-point operations; the primary intended application is speech processing, which does not rely on floating-point calculations. Operand sizes are limited to 16 and 32 bits. T0 only used three flag registers, as status bits for the results of comparisons, overflow, and saturation; conditional moves used flags held in regular vector registers.

T0 had a single memory functional unit, and two arithmetic functional units, one of which was capable of multiplies. It was organized with eight lanes, or parallel datapaths. The arithmetic units included some DSP functionality, in the form of a clipper that could perform saturation. T0's vector register file held 16 registers, each of which comprised 32 32-bit elements. The register file had a total of five read ports and three write ports. T0 supported chaining on all operations. The peak performance was 4.3 billion 32-bit operations per second, running at 45 MHz [ABI$^+$96]. T0 performed excellently at its target application, remaining in use for speech recognition for years.

The experience with T0, and the desire to create a microprocessor that could harness large amounts of bandwidth with a simple architecture, led to the development of VIRAM. In a similar fashion, the knowledge gained from VIRAM1 led to another architecture: CODE.

CODE was designed to specifically address a number of inefficiencies discovered during the creation of VIRAM1:

- short vectors;
- vector register file complexity;
- delayed pipeline inefficiency for long latency;
- imprecise virtual memory exceptions.

CODE combines two main ideas: composite organization and decoupled execution. Both are attempts to improve capacity for scaling, over what VIRAM1 could deal with.

Composite organization refers to the basic approach of designing the vector hardware as a collection of interconnected cores, as opposed to the style used in VIRAM1 of having a single vector coprocessor accessing a centralized register file. The vector register file in CODE is not a single structure: the logical register file defined in the ISA is mapped to a distributed structure. Each part of the distributed register file is associated with only a single functional unit. Thus, only a small number of access ports is required, and the logical and physical control and interconnect is very simple. Furthermore, each individual part of the distributed structure has only part of the total register file, so the structure is smaller and can therefore be simpler, faster, or lower power than the full structure.

Scaling the vector register file in VIRAM1 is difficult, because the structure is quite large to begin with. Increasing the size of the register file would probably require moving to a slower clock cycle, or require more clock cycles per access. In CODE, the vector register file can be scaled with more functional units simply by replicating the basic structure additional times. The interconnection between the decentralized structure is simple; scaling up might incur additional latency, but this does not cause problems in practice because vector execution is inherently tolerant of latency.

VIRAM1 was organized around the model of a delayed pipeline. Since memory access to the embedded DRAM took a fixed length of time, the memory pipeline was designed to account for that time. The main benefit of this organization is that it is simple to implement. Unfortunately, it is also inflexible: changes to the relative memory access time require a change in the pipeline. Additionally, the delayed pipeline cannot achieve any benefit in cases where memory accesses return quicker than the scheduled delay.

An alternative to the delayed pipeline is the decoupled pipeline [Smi84]. The decoupled pipeline separates memory access and arithmetic execution into two logically distinct components. Those components are then connected with queues, which allow instruction execution in each component to slip relative to the other. Greater latency can be tolerated by increasing the size of the queues, and the architecture can take advantage of accesses with differing latencies. Decoupled pipelines have been shown to give advantages in vector processors, particularly for short vector instructions [EV96].

CODE uses a decoupled pipeline to overcome the limitations of VIRAM1's delayed pipeline, and because decoupled pipelines facilitate simple communication and control between the decentralized cores. There is another reason that they work well together: CODE mitigates one of the drawbacks that is generally associated with decoupled execution, the need for extra storage dedicated to decoupling queues. In CODE, each core already has registers that can be used for those queues: local vector registers. Thus, CODE can take advantage of the benefits of decoupled execution, without paying a large penalty for its instantiation.

CODE addressed some of the questions that were raised during the development of VIRAM1, but there are a still a number of interesting related areas that deserve to be explored. T0, VIRAM1, and CODE were all aimed at low-power media applications. Can similar ideas be applied to desktop computing, or other application areas? How well would the ideas work when combined with more modern microprocessors? How else can vector processing be combined with scalar processors? The rest of this thesis helps to explore those issues.

2.5 Impulse memory controller and NMP

There are a number of other attempts to improve microprocessor memory access efficiency. Two of the more interesting ones, both related to aspects of my work, are the Impulse memory controller and the Near-Memory Processor (NMP).

The Impulse memory controller attempts to improve memory efficiency in two ways: remapping non-unit-stride memory streams into unit-stride streams, and remapping multiple non-adjacent pages into a single larger page [ZFP+01].

Impulse works by adding new mechanisms to a memory controller:

- shadow descriptors, that store the remapping configuration;
- an ALU to generate addresses;
- a translation lookaside buffer, for accesses to unmapped physical addresses;
- and buffers to store partially-assembled cache lines.

Software and operating system support are required to make use of Impulse. First, an application makes an Impulse system call. The operating system responds by allocating a range of contiguous virtual addresses, and then interacts with the memory controller to map physical elements to the new addresses. The memory controller requires a function that describes how addresses should be mapped, and page table entries that allow it to translate addresses.

Once the mapping is complete, the application can begin to access the mapped elements. When the memory controller sees a request in the mapped address range, it gathers the corresponding physical elements together into a dense cache line, and returns that to the processor.

Because Impulse can pack non-contiguous ranges of memory into dense cache lines, it is able to deliver an immediate improvement in the efficiency of memory accesses. For example, a processor requesting single bytes at a large stride might only use one byte out of a 128-byte cache line; with Impulse, it would be able to make use of all of the data in the cache line.

The other way that Impulse is able to improve memory efficiency is through facilitating the use of large memory pages. A typical memory page is 4 KB. The first time that a value in the page is accessed, a memory translation that requires a page table walk will be required; if the address translation gets dropped from the TLB (because it is evicted as other translations are cached), then another page walk will be required. For memory access patterns that exhibit a large amount of temporal and physical locality, the address translation overhead might be small. In other cases, where the memory access pattern does not exhibit a lot of locality, the translation overhead can be significant.

Large pages can reduce the effective overhead of address translations by letting a single translation cover more data. Most TLBs support large pages, but have certain requirements: the regular-size pages that make up a large page must be aligned properly, and must be adjacent. In some cases, data from multiple regular-size pages is copied to a large page. The costs of copying can be outweighed by the eventual benefits of reduced address translation cost, but Impulse offers a way to avoid paying the copy costs in the first place: it allows regular pages to be mapped into an effective large page, without requiring any copying. The mechanism is similar to mapping non-unit-stride streams, described above.

Impulse is an interesting way to deal with greater memory efficiency, but it has some important limitations. The software programming model is unclear: examples are given in pseudocode that resembles a modified version of C [CHS+99], but it is not clear how much the compiler would be able to generate automatically, and how much direct involvement from the programmer would be required. There are a number of additional issues that need to be addressed: how Impulse handles multiple applications; efficiency of communicating address translation information between the processor and an external memory controller; how pin bandwidth limitations affect Impulse, particularly once memory controllers are integrated on the same die as the processor; and how memory coherence and page table synchronization is handled.

A second approach to improving memory efficiency is the Near-Memory Processor (NMP) [WSTT05b]. NMP combines a number of ideas: scratchpad memory, vector processing, multithreading, DSP operations, and conditional execution.

NMP, rather than being an extension to existing microprocessor cores, is more accurately a heterogeneous system architecture that includes traditional microprocessors, as well as an additional processor that is dedicated to memory functions. The additional processor is quite ambitious. The designers claim that vector processing, multithreading, and streaming hardware are all needed to achieve good performance. Additionally, the near-memory processor runs at 4 GHz, is able to issue two instructions per cycle, has support for four multithreaded contexts, and can have a total of 256 outstanding memory requests.

The NMP designers use a combination of vector instructions, streaming hardware, bit manipulation, and multithreading on all of the kernels used to measure the performance of the system [WSTT05a]. It is not clear exactly how the different features interact in software — for example, much of the state is not saved on a context switch.

The designers suggest that different features can add up to give good performance, but a few critical questions are not answered. First of all, how much do each of the features contribute to the overall performance? Secondly, how is the system programmed — and how easy is it for compilers and programmers to take advantage of those features? Finally, what is the cost, in terms of area, complexity, or power?

2.6 Hitachi SR8000

The Hitachi SR8000 [TSI+99] was designed to derive some of the benefits of vector execution in a scalar microprocessor. It combined a number of different features, including parallel processing with fast custom networks, large numbers of registers to increase memory concurrency, and specialized synchronization instructions.

The overall architecture of the SR8000 is multiple nodes connected with a multi-dimensional crossbar. Each node contains multiple scalar processors [SKH+99], a custom crossbar, and local DRAM.

One way that the SR8000 attempts to achieve vector benefits is through processing multiple iterations of a loop in parallel, attempting to amortize loop costs over the amount of physical parallelism, much as a vector computer can efficiently execute multiple elements of a single vector in parallel. Because the SR8000 is not a true vector computer, it relies on what they call “DO-loops”: loops in the code that can be effectively parallelized to run on physically independent processors. The loops are executed in a “fork-join” model, where code is executed in a serial fashion on a single processor (in the parent thread) until it encounters a fork point: at that time, the parent thread sends a signal to threads running on the other scalar processors (the child threads). The signal wakes the child threads and instructs them as to the location of the loop to execute. As each child thread finishes execution of the parallel section of code, it signals to the parent thread; once the parallel execution is complete on all child threads, the parent thread can resume serial execution of code.

There are many potential pitfalls to executing a single loop effectively on separate physical processors. Besides requiring enough communication capability, the hardware and operating system have to cooperate to start and end loops quickly — otherwise, only very long loops (or no loops at all) will provide a speedup over execution on a single processor. The SR8000 uses a number of techniques to ensure that parallel loop overhead remains small.

To keep loop startup overhead low, the SR8000 uses a feature called "COMPAS" (CO-operative MicroProcessors in a single Address Space). COMPAS provides a means for synchronizing cache tags before a fork point and after a join point, so that data stored by one of the scalar processors can be accessed by a different scalar processor at a later point. The SR8000 accomplishes fast tag synchronization by including a special storage controller between the scalar processors and the node DRAM. The storage controller contains a crossbar, used by the scalar processors to communicate to each other and DRAM, as well as a copy of the cache tags of each of the scalar processors. When the parent thread needs to synchronize tags, it executes a sync instruction that synchronizes its cache tags to the shared copy. At the end of a parallel section of code, each child processor executes a sync instruction that synchronizes its tags to the copy. Once the storage controller sees that all child processors have synchronized their cache tags, it signals to the parent thread that the parallel loop has completed execution.

DO-loops on the SR8000 effectively provide a fast means of barrier synchronization among the scalar processors. Because the storage controller communicates directly with each processor, COMPAS is able to provide the synchronization much faster than if it was performed through DRAM.

Another key component to making DO-loop execution fast is avoiding operating system scheduling of the parallel threads on each loop execution. Organizing multiple threads to execute a single block of code at the same time is known as “gang scheduling” [FR92], and was originally developed for applications running on multiprocessor systems that need to execute certain portions of code concurrently on multiple processors [Ous82]. Effective gang scheduling in the general case can be complicated, so operating systems generally want to make the best use of available hardware and allow a program to execute on many threads, but do not want to stall programs to wait for resources to become available [FR90]. The SR8000 avoids the potential problems by assigning all node resources to a single thread at any time.

Finally, the SR8000 has a feature that tries to allow scalar processors to tolerate latency, much as a vector processor does. Part of the reason that vector computers are able to tolerate memory latency is that they are able to keep many memory operations in flight at the same time, and the only way to keep many memory operations in flight is if you have somewhere to put that data when it is returned from memory. For vector machines, that storage location is the vector registers. A typical vector machine might have 32 registers, each of which can hold 64 double-precision floating-point values, for a total of 16 KB of data storage. A scalar machine will typically have much less named space available for returned values from memory: 32 single-element registers is only 256 bytes. Caching and out-of-order execution can increase the temporary storage available, but the SR8000 relies upon a technique developed earlier, called “Pseudo Vector Processing” (PVP) [NIY$^+$94].

The basic idea in PVP is to increase the amount of named register storage available for memory operations. In order to do this while remaining compatible with existing microprocessor ISAs, they rely on windowed register files. Windowed register files are usually used for fast procedure calls, by creating a simple way for calling procedures to pass argument values: the registers designated as output registers are automatically mapped as input registers to a called procedure.

Since PVP requires that more named storage be available, they introduce new instructions for accessing register windows beyond the current one: values can be loaded into the next register window, and stored from the previous register window. Those windows are not accessible to arithmetic instructions, but that does not cause any problems. Since the register window shifts between loop iterations, values that were loaded during the last iteration (where they were inaccessible to arithmetic instructions) get mapped to locations that are accessible. Values that were involved in calculations during the last iteration become mapped to an area that is no longer accessible to arithmetic instructions, but they only need to be available to storage instructions. In this way, PVP acts as a hardware accelerator for software pipelining, while giving the additional benefit of allowing access to more total storage.

The combination of features used in the SR8000 can deliver good performance, but the processors in the SR8000 are not true vector processors. Individual scalar cores get some latency tolerance benefit from PVP, but the total amount of named storage at any point is limited to a small multiple of the named storage of a regular scalar processor: the current window, plus the next and previous windows. Furthermore, access to the additional storage is constrained, and therefore not applicable to all situations where a more general access method, such as true vector registers, could be used.

The Hitachi SR8000 provided the initial inspiration for a proposed joint project between IBM and Lawrence Berkeley National Laboratory to allow multiple Power microprocessors to operate together in a vector fashion, similar to the way that PVP works. The initial proposal never materialized, but the project name, ViVA, was used to describe two different features that were implemented in later Power chips: a fast synchronization capability, and a new software prefetch mechanism. My work at LBL, which led to the research described in this dissertation, did not use the same approach as previous ViVA work, but it shares the name because it falls under the project goal of adding vector-style capabilities to commodity microprocessors.

2.7 Cray T3E E-registers

The Cray T3E used the idea of extra storage space to help tolerate memory latency. The T3E is a large multiprocessor built with Alpha 21164 processors and additional custom logic. Each node contains a single processor, local memory, a router to connect to the network, and a control chip. The control chip contains a set of registers called "E-registers", which are 640 (512 user, 128 system) 64-bit registers that are used for a number of messaging, memory access, and synchronization purposes [Sco96].

One of the largest benefits of the E-registers is that they provide more temporary storage to allow for greater concurrency in access to memory. The greater amount of storage allows a greater number of memory requests to be in flight at the same time, which allows a greater total bandwidth to be delivered. E-registers can also be used for strided block access.

The E-registers serve a number of other purposes, including address space expansion (addresses from the processor are combined with segment addresses stored in E-registers to form a larger global address), atomic memory operations (natively supporting fetch & inc, fetch & add, compare & swap, and masked swap), fast interprocessor communication, and interprocessor synchronization.

The E-registers are a useful addition to the Cray T3E, but they are limited in the memory access usefulness. Block transfers are limited to 8 words, in a strided pattern; longer transfers, or indexed accesses, are not supported.

2.8 Other Vector Machines

Vector computers have a rich history. Many different variations on the basic design have been implemented and tested. The basic architectural ideas have survived for decades, even as the motivation for using those ideas has evolved in response to changing technology and application demands.

In their early history, vector machines generally were developed with the purpose of producing the fastest computers available. In practice, this meant that they had to provide a simple way to access large amounts of bandwidth, and organize a large number of functional units to operate on that bandwidth. Vectors provided a useable framework that could be scaled up in performance.

Eventually, vector computers transitioned to machines that were able to deliver a high percentage of their peak performance. Other architectures and networks of computers — particularly networks made up of microprocessors whose development costs were subsidized by a large market outside of high-performance computing — were able to provide higher theoretical peak performance, but vector computers were able to provide greater delivered performance in a more manageable system.

Finally, the meaning of vector computers has changed over time. In recent years, the vector processing name has been given to simple SIMD microprocessor ISA extensions. Currently, graphics processing units (GPUs) and even multicore processors are starting to re-examine vector architectures, as a way to capture data parallelism effectively, organize large amounts of processing to take advantage of large bandwidth, and keep architectures simple and low-power.

2.9 Vectors and Multithreading

There have been a few different processors that attempt to combine vector processing and thread processing. The Espasa-Valero architecture [EV97] is a relatively straightforward combination of a vector processor and multithreading. They analyze performance for 2–4 hardware contexts, and switch contexts on a blocking operation. They report good performance benefits from adding multithreading to a vector processor.

The Vector-Thread (VT) [KBH$^+$04] paper from MIT analyzes SCALE, one particular implementation of a VT architecture. VT is more flexible than the model presented in the Espasa-Valero paper: individual virtual processors (the hardware contexts) can execute in a free-running mode, where they operate as a traditional multithreaded architecture, or they can operate together in a more organized manner.

To facilitate coordinated operation between separate virtual processors, VT includes a control processor that communicates to all of the virtual processors. The control processors issues two sorts of commands to the virtual processors: thread-fetch and vector-fetch. Both commands instruct the processor to fetch and execute a block of instructions; for thread-fetch commands, different virtual processors operate independently, and for vector-fetch commands, the processors act as virtual processors in a single vector processor.

VT is supposed to flexibly give the benefits of both multithreading and vector processing: the processor can act as a regular multiprocessor, or if programs make use of vector-fetch operations, the system gets the benefit of vector operations. Better yet, it is able to mix both types of computation together.

The combination of multithreading and vector computation seems promising. It will be interesting to see if designers are able to deliver on their claims of getting the benefits of both systems, while not losing ground due to overhead in dealing with trying to harness two different types of parallelism.

2.10 Summary

Vectors have been applied in many different circumstances. This chapter described some of the features and projects that most directly influenced the direction of my research.

The common thread running among these projects is that they all take the basic idea of vectors and explore different ways to build on its strengths: using data-level parallelism, cheaply and effectively. The Cray X1 combines vectors with caches and virtual memory;

VIRAM1 uses vectors to take advantage of high on-chip bandwidth available from embedded DRAM; other machines examined here include vector features delivered by memory controllers, windowed register files, and multithreaded computers.

In the next chapter, I present the ideas behind my research, which looks at different simple, low-cost ways to add vectors to existing microprocessors.

Chapter 3

Extending Scalar ISAs to Support Vectors

Most vector computers are really two computers in one: a regular scalar computer, and a separate vector computer. Each component has its own register file, functional units, and control; additionally, much of the memory hierarchy is duplicated. The separate approach results in high synchronization and data communication costs. More importantly, the idea of vector computers as a separate unit added on to the main scalar computer imposes a barrier — real and psychological — to embracing vector operations in modern microprocessors. Designers think that the only option is to spend a significant amount of time and energy designing a large vector add-on, which will be difficult for software to use. In reality, vector operations can be added to designs for relatively little cost, and software can take advantage of vectors easily — sometimes, without recompiling.

In this chapter, I describe an array of options for integrating vectors and vector-style operations into modern microprocessors. I describe evolutionary methods that allow restricted SIMD extensions to be used as full vector ISAs, as well as low-cost methods for achieving most of the performance of a full vector implementation, with much lower complexity and cost.

In Section 3.1, I address the question of why it is important to look specifically at integrated solutions: why the traditional approach has some important drawbacks, and what an integrated solution can bring to the table.

One of the central ideas I present in this dissertation is that vector operations do not have to be an all-or-nothing proposition: there is a vast array of options that can give some of the benefits of a full vector implementation at a lower cost. The rest of this chapter considers a few interesting points on the spectrum, and examines them in detail.

The first implementation I talk about, in Section 3.2, is ViVA: Virtual Vector Architecture. ViVA uses vector-style memory operations, but does not include any vector arithmetic processing: the result is that with only very minor hardware changes, a processor can achieve significant benefits on memory operations.

In the last section, 3.3, I describe an approach to extending existing microprocessors and traditional scalar ISAs to include full vector processing. My approach allows for the possibility of incremental change, and even allows existing SIMD programs to take advantage of vector instructions without recompiling.

3.1 Examining Integrated Solutions

The traditional approach to vector processors, described below, has been around for many years — why should we consider different approaches now? This section describes the reasons that it makes sense to look at different ways to combine scalar and vector processing.

3.1.1 Traditional Vector Design Drawbacks

The traditional approach to vector processor design is to include a scalar core that can stand on its own, and a completely separate vector coprocessor that is loosely connected to the scalar unit.

The scalar core in traditional vector computers is required because vector coprocessors only execute data-processing commands: data loads and stores, and arithmetic and logical operations. All other operations, including control comparisons, loops, branches, and I/O, are executed on the scalar processor. A good mental model for traditional vector computers is that the scalar core executes the program; when it encounters a vector instruction, it does no processing on it beyond delivering it to the vector coprocessor. The instruction is then decoded and executed there.

The vector coprocessor has most of the parts of a full vector standalone core — the ability to decode, issue, and execute instructions — but cannot be considered a full standalone processor, because it effectively implements a partial ISA, that does not contain operations that are critical for a full core.

There are a number of drawbacks to the traditional approach of having a separate scalar core and a vector coprocessor. The first is that the computer really contains two separate computers: one that initially processes programs and executes control statements, and one designed for fast arithmetic processing. By separating the cores, the computer has to deal with all of the difficult issues of multiprocessing — even for a single vector computer. Latency between the scalar core and vector coprocessor is usually quite large, and synchronization between the two components is difficult.

The extra latency between the cores is important: often, the extra processing time means that some otherwise-convenient programming patterns just do not make sense. For example, scalar cores are often good at processing irregular data patterns — shuffles, transposes, highly control-dominated sequences — and vector cores are usually efficient for processing regular data patterns. Very rarely does a program only contain one pattern or the other: control-dominated programs often contain sequences of pure data processing, and data-heavy programs often need to do irregular processing around boundary edges and interfaces. If there is a way for the two cores to communicate to each other quickly, both cores can process data; if the data communication between the cores takes many cycles, the programmer may be forced to use one core to execute both parts, in a slower, more complex fashion.

Synchronization is similar. The basic issue is the same, since synchronization is simply communication of control, rather than data. Synchronization is especially important if the cores are able to communicate data quickly; in that case, they will need to be able to synchronize more often. If the programming model does not include the capability for fast communication between the scalar and vector units, then synchronization is not as important — but it is still needed. Often, there are times when data will have to be touched by both units, even if that is a relatively slow operation. For example, program initialization typically requires the scalar core to read in data from disk, or set up data structures; if that data is touched by the vector unit, there will most likely need to be some sort of barrier. If the scalar and vector core are designed as two separate computers, barriers and synchronization will introduce more latency into program execution.

There are also a number of drawbacks in terms of design. The scalar core and vector core are both able to perform many of the same sorts of operations: fetching instructions, decoding, issuing to datapaths, and so on. By treating the two units as separate cores, there is less of a chance to share overhead costs, both in terms of chip design (including verification) as well as chip area.

3.1.2 Benefits of an Integrated Solution

Integrated solutions avoid many of the problems associated with proposing a solution that is radically different from existing approaches. The benefits go beyond simply avoiding problems, however: an integrated approach to combining scalar and vector execution has a number of innate advantages.

One important benefit to an integrated solution is that a combined approach is inherently more flexible than a separate coprocessor approach. Often, applications work in the general model of loading data into registers, operating on that data, and then writing those values back. If the act of loading data implicitly sequesters it to one core or the other, without an efficient way to operate on that data in the other core, the resources for processing that data are limited to what is available in only a single core.

The problem with having two separate cores and no quick, efficient synchronization mechanism is greater than individual data element processing being limited to one of those cores. More importantly, the data block as a whole is effectively limited to a single core. Flexibility for larger data blocks is a large plus: often, even though the majority of an application will be spent on either data-intensive or control-intensive work, there will still be a significant amount of the other sort of processing. With fast synchronization, it is easy for a vector core and scalar core to both be applied to the problem.

3.2 ViVA

The first integrated solution to combining scalar and vector processing is ViVA, the Virtual Vector Architecture [GOS$^+$08]. ViVA is an attempt to provide a low-cost extension to standard scalar microprocessors that gives them many benefits of vector memory accesses.

A key idea is that ViVA is designed to be a low-cost, low-effort way to add support for vector memory operations. In the next section, I will take a look at a more complete solution; in some cases, the simpler approach I describe here might be more appropriate.

This section describes the hardware associated with ViVA. Chapter 4 describes the applications that I use to evaluate ViVA, and Chapter 6 talks about the results and performance implications.

3.2.1 Introduction to ViVA

ViVA adds a software-controlled memory buffer to traditional microprocessors. The new buffer logically sits between the L2 cache and the microprocessor core, in parallel with the L1 cache, as shown in Figure 3.1.

ViVA adds a small number of new instructions that perform block memory transfers. The transfers move data between DRAM and the ViVA buffer; scalar operations move

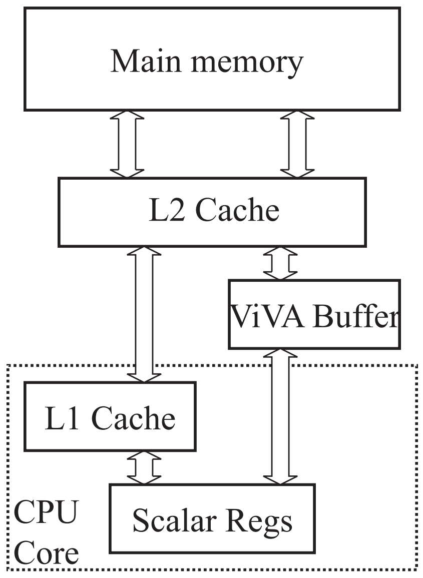


Figure 3.1: Simplified memory hierarchy, showing the relative position of the ViVA buffer and caches.

individual elements between the new buffer and scalar registers in the core, where regular scalar instructions can operate on them.

3.2.2 ViVA Architecture and Implementation

Adding ViVA to a microprocessor involves a few changes to the ISA, as well as adding some additional hardware to the system.

Programming model

Vector computers are designed to operate on large blocks of data collected from memory. ViVA implements vector-style memory operations, and so the program model is largely the same. Block transfers are performed between DRAM and the ViVA buffer using instructions that have regular vector semantics: unit-stride, strided, and indexed (scatter-gather) transfers are all supported.

One important difference between the regular vector programming model and the ViVA programming model is that ViVA does not support any vector arithmetic operations. Since

ViVA only adds vector-style memory operations, all arithmetic and logical operations have to be performed in the regular scalar registers. Thus, data is transferred to the ViVA buffer using vector-style loads; individual elements are transferred to the scalar core, operated on, and transferred back to the ViVA buffer; finally, vector-style stores return values to memory. Figure 3.2 shows the basic programming model. Real code would typically add in a number of optimizations, such as double buffering.

```
do_vector_loads;
for(all_vector_elements) {
   transfer_element_to_scalar_reg;
   operate_on_element;
   transfer_element_to_buffer;
}
do_vector_stores;
```

Figure 3.2: Pseudocode showing simplified ViVA programming model.

A large advantage of ViVA is that the hardware is simpler than on a typical full vector machine, but it can use the same compiler technology. ViVA memory operations respect the same semantics as regular vector memory operations, and the basic vectorizing techniques can be the same. One difference is required: since ViVA does not have vector arithmetic or logical operations, any of those operations that would be produced in the vector code have to be replaced with a scalar loop that iterates over the elements of the vector, as in the pseudocode in Figure 3.2. Since vector compilers are a mature technology, real applications could benefit from ViVA almost immediately.

One downside of ViVA is that it does not offer the benefits of reduced instruction bandwidth, or the ability to use multiple lanes, that is available from a full vector architecture as described in Chapter 1.

ViVA's scalar execution model — the extra loop that replaces vector arithmetic instructions — is simple for vector compilers to adapt to, and it can actually provide benefits for some applications. The first advantage is that ViVA does not need long execution vectors to get good performance. Since ViVA does not execute arithmetic or logical operations on vectors, obviously their length does not matter. While ViVA does get better performance with longer memory vectors, there are still a number of applications that have regular memory access streams — amenable to encoding in long vectors — and irregular execution vectors, which can only take advantage of short vectors. In other cases, one memory access — say, loads — will have a regular access pattern, and another — stores — will not. ViVA is able to perform the long-stream accesses using vector loads, and perform the irregular accesses using scalar stores. The sparse matrix-vector application, described in Chapter 4, takes advantage of this optimization to get a large performance boost.

Often, vector machines are very slow at moving data between vector and scalar registers — sometimes, the only way to move data between the different register files is through memory. The limitation can make sense if good performance depends on executing long vectors in the vector coprocessor, but it limits the flexibility of the system. Since ViVA is designed to move data between the ViVA buffer and the scalar register files, the transfers are quick. Sometimes, the ability to move data quickly allows the system to use different algorithms: a program that needs to shuffle data in an odd pattern can move elements from any location in the ViVA buffer to a scalar register, and back to a possibly different register. Any application that can use that ability to perform arbitrary shuffles quickly, without going back to memory, will derive a large benefit from ViVA.

Hardware details

The main hardware changes required for ViVA can be broken into three categories, each of which is described in detail below. The first is a software-controlled buffer, logically in the memory hierarchy between the core and L2 cache, which supports vector-style transfers to and from memory, and scalar transfers to and from scalar registers. The second change is the addition of a few new instructions to the ISA to take advantage of the buffer. Finally, a small number of control registers are added to help manage the ViVA buffer.

ViVA buffer

The first major new component of ViVA is the added buffer. The ViVA buffer shares characteristics of a vector register file, and a scratchpad memory.

The ViVA buffer acts as a set of vector registers, but it has no associated datapaths. Because it operates as a vector register file, it exhibits many of the same characteristics. For example, the ISA does not fix the length of the vectors stored in the ViVA buffer — there is a maximum length, but applications are free to set a shorter length that will be used for vector instructions. In my experiments, I study lengths from 16 64-bit words through 256 words. Most experiments use 64 words, which is typical of many real vector computers. With a length of 64 words, the ViVA buffer holds a total of 16 KB of data — about the same size as an L1 data cache. Thus, the total chip area consumed by the ViVA buffer is relatively small. Section 3.2.4 explores the costs of ViVA in more details, including the chip area it consumes.

The ViVA buffer is logically situated between the core and the L2 data cache. No extra ports are needed to communicate with the L2 cache; instead, the ViVA buffer communicates with an L2 arbitration unit, as shown in Figure 3.3. The arbitration unit has input queues for L1 data and instruction caches, as well as the ViVA buffer. It may also have separate queues for demand and prefetch requests from caches. The arbitration unit prioritizes requests from the different sources and sends them to the appropriate port on the L2 cache.

Figure 3.4 shows typical data flow for both scalar and ViVA memory operations. In the diagram, the wider arrows represent data and control flow, and the thin arrows represent only control flow (for example, sending a request for a particular word from memory). The load diagram, part (a), is very similar for both the scalar and ViVA case: in both cases, the request has to go all the way to main memory (steps 1 – 3 for scalar, 1 – 2 for ViVA) before the data can be returned (steps 4 – 6 for scalar, 3 – 4 for ViVA). In ViVA's case, the actual transfer of data to scalar registers (step 5), where is can be used, is performed using individual scalar transfer instructions.

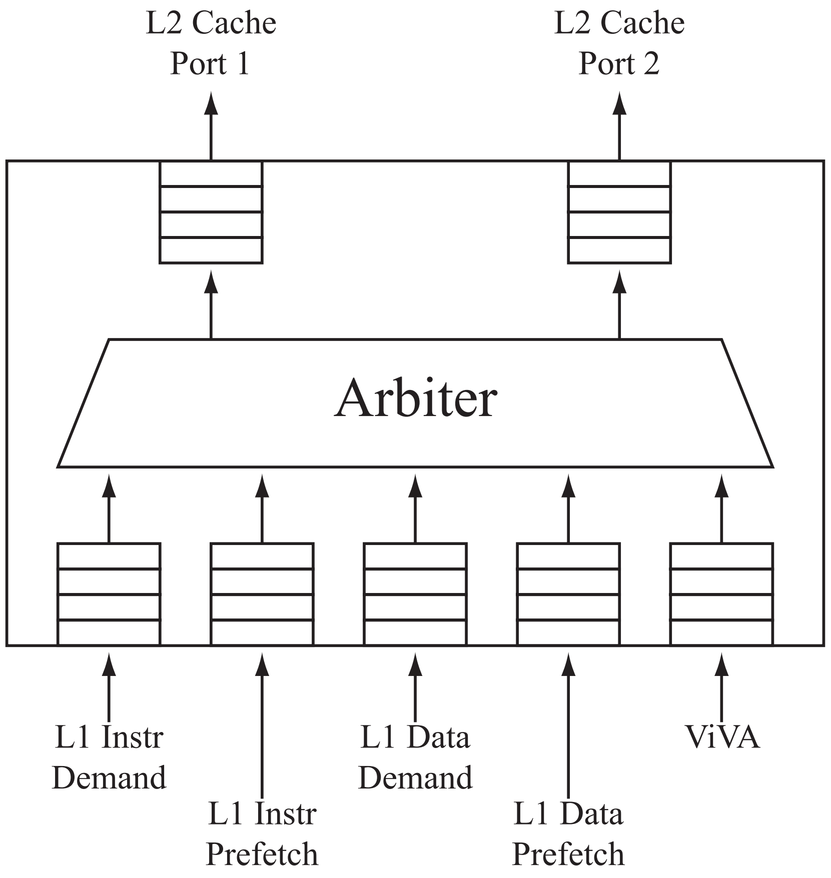


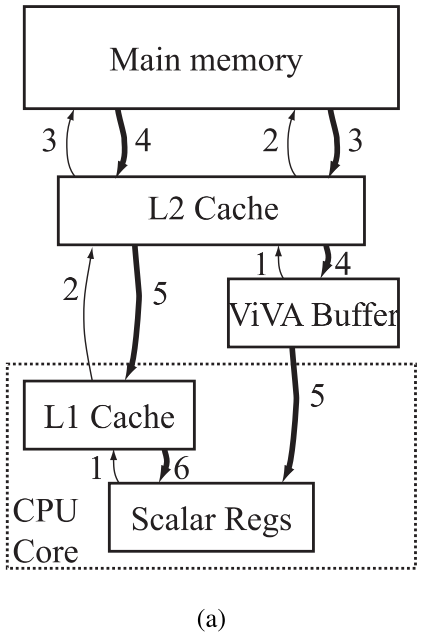
Figure 3.3: Simplified arbitration unit, prioritizing requests from L1 caches and ViVA buffer to send to the two L2 ports. Queues shown for incoming and outgoing requests.

The store diagram, part (b), shows one advantage that ViVA has over scalar requests: the ViVA operation can avoid a fill operation. The diagram depicts a write-allocate L1 cache. In that sort of cache, when a value is written that is smaller than a full cache line, the remaining parts of the cache line have to be filled in with valid data. That valid data is requested from memory in a fill operation. In the diagram, scalar data is first written in step 1; then a fill is performed, first requesting the cache line from memory (steps 2 – 3) and then returning it (steps 4 – 5). Eventually, the cache line is evicted, and memory must be updated (steps 6 – 7).

It is possible for the fill to be avoided if the full cache line is written at once. In ViVA, full cache line writes are common: elements are typically collected in a ViVA register, which is then written to the cache. Since ViVA is designed around vector operations, programs can often make use of that data layout. In the ViVA case, if a full cache line's worth of data is first written to the ViVA buffer (step 1), then the write to memory (steps 2 – 3) does not need to perform a fill.

Scalar operations can theoretically avoid a fill operation in some cases, but in practice they typically need to perform it. Since a single instruction only stores a single word, a single instruction can never write a full cache line of data. Store buffers may be able to merge writes: if a buffer merges an entire cache line's worth of writes, it can avoid a fill operation. Often, though, the number of writes that can be merged together is limited (to simplify store queues), preventing the possibility for a full cache-line write. A cache line written from a lower level of cache may be the same size as a cache line in a higher level of cache, meaning that no fill will be required for the higher level of cache. But, a fill may already have been performed for the lower level of cache, meaning that the only thing saved is a second fill for the same data. Additionally, the only time a cache line from a lower level of cache will have to be written to the higher level of cache are in cases where the higher level of cache is not inclusive of the lower level of cache (a useful feature, so that the higher level of cache can perform all of the inter-core coherency operations). Even in that case, writes from the lower level of cache to the higher level of cache might be relatively rare if the lower level of cache is not write-allocate.

The ViVA buffer acts a register file. Locations in the ViVA buffer are named and accessed as in a regular register file. As a result, no automatic coherence mechanisms are needed for the ViVA buffer. A value that has been loaded in the ViVA buffer will not automatically be updated or evicted by other memory operations, just as a value loaded into a scalar register file will not be updated by other loads or stores. As soon as values are stored from the ViVA buffer into the L2 cache, regular coherence mechanisms apply. No consistency orderings are guaranteed between ViVA and scalar memory operations: memory fences are required to enforce a particular order.



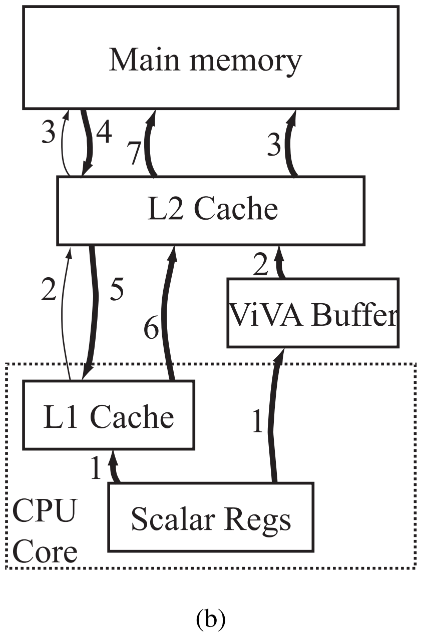


Figure 3.4: Comparison of scalar and ViVA data flow for (a) loads and (b) stores. Thick lines show data transfer, thin lines show control flow. Detailed description of individual steps is given in the text.

In practice, the lack of automatic coherence is not a problem. Vector compilers can manage coherence in ViVA the same way they do with traditional vector computers. Generally, programs do not frequently access the same locations in memory with both scalar and vector operations, so coherence is not an issue. In cases where different types of operations are mixed, memory fences still might not be needed: as long as no single memory location is written by one type of operation (say, a scalar store) and then read with the other type (say, a vector load), a memory fence is not needed.

By avoiding automatic coherence and giving no consistency ordering guarantees, ViVA memory access processing can be simpler (and more power-efficient) than scalar memory accesses. The hardware offers a tradeoff: the stricter consistency model is available for scalar accesses, which may simplify some programs, but it comes with the cost of extra energy per access.

Generally, in-order instruction processing is an excellent match for vectors because the simpler design of in-order processors can save significant energy, and vectors give latency tolerance that out-of-order execution would provide. It might seem that either vectors or out-of-order execution could provide the necessary tolerance of latency, and that using both techniques would provide no additional gains. In fact, each method provides benefits that the other cannot. Out-of-order machines are limited by the size of their instruction windows, which have to be constrained to keep design complexity and power consumption to reasonable amounts. Vectors machines allow for a large amount of concurrency, but certain instruction sequences can reduce performance to levels below the processor's potential. My simulations, presented in this thesis, and the work of others [EVS97], show that the increased latency tolerance and concurrency of vectors gives additional improvement to an out-of-order processor. The benefits of using both methods are greater than the benefits of using either by itself.

Out-of-order processing in the ViVA buffer is similar to out-of-order processing in scalar registers, with a few small differences. As in the scalar registers, there is a larger set of physical registers than named registers. The system keeps track of the committed state, and the current status of operations in flight; as a new register is needed, control logic temporarily maps an unused register to the address of the needed register. On an exception, the processor performs the necessary actions, then recovers the previously committed state and begins re-processing from that point.

One important difference between the out-of-order handling for ViVA and scalar registers is that a single ViVA instruction refers to many operations, and a single register corresponds to many elements. That means that any per-instruction or per-register processing cost is amortized over the full vector length. Actions such as inter-instruction dependency checking can be performed once, and applied to many operations: that means that for a certain amount of work (and associated overhead state), a greater number of operations can execute out of order, and more concurrency is obtained. Put another way, the power and instruction state costs of concurrently processing a given number of vector operations are less than the costs of concurrently processing the same number of scalar operations.

The ViVA buffer needs to be able to provide access to multiple elements every cycle. Most vector loads will need to be able to write one register file element to the buffer for each element in the vector (as a load transfers elements from memory to the ViVA buffer), and vector stores will need to read at the same rate. Indexed accesses need an additional register file read per element, to access the offset value. Scalar transfers need to access the ViVA buffer as well. On one hand, concurrency gives us good performance; on the other hand, it also increases demands on the hardware.

Fortunately, there are some ways that the ViVA buffer can be simplified. In addition to traditional multi-ported register file simplification techniques such as time division, replication, and banking, [RTDA97, BDA01] ViVA is able to take advantage of the fact that accesses generally proceed in regular patterns. Instead of using a port that is the same size as data elements, the ViVA buffer can have wider ports that are used to access multiple elements less frequently. [Asa87]

For example, consider a case where the port is four elements wide. If a functional unit tries to read element 0, the port actually grabs elements 0 through 3, and puts them in a small, fast buffer, returning element 0 to the requesting unit. If the functional unit next reads element 1, its value is returned from the small buffer, bypassing the need to access the full register file.

Wider ports can introduce additional latency in the case of conflicts. In that case, the additional latency is typically only a few cycles: some of the initial requests will access the register file on the first cycle, and grab multiple elements; on the next cycle, the remaining requests will proceed. After that, for the full lengths of the vectors, requests will proceed without any extra delay. A worse case is when accesses to the register file are not regular, some or all of the extra values read into the small buffer are not used, and conflicts are more frequent.

New instructions

The second major change required to support ViVA is the addition of a small set of new instructions added to the processor's ISA. The first class of new instructions includes the vector instructions that transfer blocks of data between DRAM and the scalar buffer; the second class includes the transfers between the buffer and scalar registers; the final class is control operations.

Table 3.1 shows the vector transfers supported in ViVA. They include stores and loads, in a unit-stride, strided, or indexed pattern. The instruction mnemonic also includes the size of elements transferred. No difference is required for floating-point and scalar values: the ViVA buffer handles both types the same way, so no differentiation is required.

Vector instruction operands include a single vector register, and two general-purpose registers, which serve as a base and offset. The exception to this pattern is the indexed instruction, which includes a single general-purpose register to serve as the base address, a vector register that is the source or destination of the data, and another vector register that provides per-element offsets from the base address.

Table 3.2 shows scalar transfer instructions. Scalar instructions include the direction of the transfer and whether the transfer involves the floating-point or fixed-point (general-purpose) registers. As with the vector operations, the size of individual elements is specified within the instruction. Scalar instructions include the source and destination registers, as well as an additional general-purpose register that contains the index of the element within the vector register.

Table 3.2 shows the third and final class of new instructions, which contains only two operations: one that stores a value from a general-purpose register to a ViVA control register, and a complementary instruction that reads values from control registers.

`vivalub`	`%v_dest,`	`%s_base,`	`%s_offset`	ViVA Load Unit-stride Byte
`vivaluh`	`%v_dest,`	`%s_base,`	`%s_offset`	ViVA Load Unit-stride Halfword
`vivaluw`	`%v_dest,`	`%s_base,`	`%s_offset`	ViVA Load Unit-stride Word
`vivalud`	`%v_dest,`	`%s_base,`	`%s_offset`	ViVA Load Unit-stride Doubleword
`vivalsb`	`%v_dest,`	`%s_base,`	`%s_offset`	ViVA Load Strided Byte
`vivalsh`	`%v_dest,`	`%s_base,`	`%s_offset`	ViVA Load Strided Halfword
`vivalsw`	`%v_dest,`	`%s_base,`	`%s_offset`	ViVA Load Strided Word
`vivalsd`	`%v_dest,`	`%s_base,`	`%s_offset`	ViVA Load Strided Doubleword
`vivalib`	`%v_dest,`	`%s_base,`	`%v_offsets`	ViVA Load Indexed Byte
`vivalih`	`%v_dest,`	`%s_base,`	`%v_offsets`	ViVA Load Indexed Halfword
`vivaliw`	`%v_dest,`	`%s_base,`	`%v_offsets`	ViVA Load Indexed Word
`vivalid`	`%v_dest,`	`%s_base,`	`%v_offsets`	ViVA Load Indexed Doubleword
`vivasub`	`%v_src,`	`%s_base,`	`%s_offset`	ViVA Store Unit-stride Byte
`vivasuh`	`%v_src,`	`%s_base,`	`%s_offset`	ViVA Store Unit-stride Halfword
`vivasuw`	`%v_src,`	`%s_base,`	`%s_offset`	ViVA Store Unit-stride Word
`vivasud`	`%v_src,`	`%s_base,`	`%s_offset`	ViVA Store Unit-stride Doubleword
`vivassb`	`%v_src,`	`%s_base,`	`%s_offset`	ViVA Store Strided Byte
`vivassh`	`%v_src,`	`%s_base,`	`%s_offset`	ViVA Store Strided Halfword
`vivassw`	`%v_src,`	`%s_base,`	`%s_offset`	ViVA Store Strided Word
`vivassd`	`%v_src,`	`%s_base,`	`%s_offset`	ViVA Store Strided Doubleword
`vivasib`	`%v_src,`	`%s_base,`	`%v_offsets`	ViVA Store Index Byte
`vivasih`	`%v_src,`	`%s_base,`	`%v_offsets`	ViVA Store Indexed Halfword
`vivasiw`	`%v_src,`	`%s_base,`	`%v_offsets`	ViVA Store Indexed Word
`vivasid`	`%v_src,`	`%s_base,`	`%v_offsets`	ViVA Store Indexed Doubleword

Table 3.1: ViVA vector operations and mnemonics.

`mvsb`	`%s_dest,`	`%v_src,`	`%s_offset`	Move ViVA to Scalar Byte
`mvsh`	`%s_dest,`	`%v_src,`	`%s_offset`	Move ViVA to Scalar Halfword
`mvsw`	`%s_dest,`	`%v_src,`	`%s_offset`	Move ViVA to Scalar Word
`mvsd`	`%s_dest,`	`%v_src,`	`%s_offset`	Move ViVA to Scalar Doubleword
`msvb`	`%v_dest,`	`%s_offset,`	`%s_src`	Move Scalar to ViVA Byte
`msvh`	`%v_dest,`	`%s_offset,`	`%s_src`	Move Scalar to ViVA Halfword
`msvw`	`%v_dest,`	`%s_offset,`	`%s_src`	Move Scalar to ViVA Word
`msvd`	`%v_dest,`	`%s_offset,`	`%s_src`	Move Scalar to ViVA Doubleword
`mtvcr`	`%s_dest,`	`%s_src`		Move To ViVA Control Reg
`mfvcr`	`%s_dest,`	`%s_src`		Move From ViVA Control Reg

Table 3.2: ViVA non-vector operations and mnemonics.

Control registers

ViVA has a small number of control registers, shown in Table 3.3: two that control lengths of registers, and one for strides. The first length register is the maximum vector length

(MVL) and the second is vector length (VL). Both operate as in traditional vector machines: MVL is a hardwired read-only control register that contains the physical length of vector registers on the current machine. VL is used by programs to specify a length (up to MVL) to use for vector operations; instructions are only processed on elements up to that length. By writing a smaller value to VL, a program can run instructions on vectors that are shorter than physical vector registers. The last control register is the stride register, which allows programs to specify their desired stride — the distance between consecutive elements for strided memory operations.

Number	Name	Purpose
0	MVL	Maximum Vector Length
1	VL	Vector Length
2	VSTR	Vector Stride

Table 3.3: ViVA control registers.

3.2.3 ViVA Programming

In a full system implementation, ViVA would be programmed much as vector computers are programmed today: a vector compiler would vectorize the loops and structure the code appropriately using strip mining and so forth. The ViVA compiler would require the additional straightforward step — beyond regular vector compilation — of adding a scalar loop to replace traditional vector arithmetic instructions.

Since modifying or creating a vector compiler is outside the scope of my work, and to maintain full control over generated and tested code, I use assembly language programming to conduct my experiments. Individual functions are written in assembly, and then called from the main C code.

Figure 3.5 shows example code to perform a unit-stride triad: `z[j] = x[j] × factor + y[j]`. The first part of the code executes two vector loads (using `vivalud`, ViVA Load Unit-stride Double instructions); each of those instructions executes many operations, transferring an entire vector's worth of data into the ViVA buffers. The next part of the code is the scalar loop; the loop iterates over a single vector length, transferring individual elements to the scalar registers (using `mvfd`, Move Vector to Floating-point Double instructions), then performing a multiply-add operation, and finally transferring elements back. The last part of the code executes a vector store instruction, to transfer an entire vector's worth of elements back to memory. Optimized assembly code could take advantage of the entire ViVA buffer and schedule instructions more efficiently, but this segment of code illustrates basic ViVA programming concepts.

```
begin_vloop:
   vivalud %v0,  %r1,  %r2         # get x[j]               } Vector loads
   vivalud %v1,  %r1,  %r3         # get y[j]

   xor     %r30, %r30, %r30        # r30 = 0
   xor     %r31, %r31, %r31        # r31 = 0
begin_sloop:
   mvfd    %f2,  %v0,  %r31        # move from vec reg      ⎫
   mvfd    %f3,  %v1,  %r31        # move from vec reg      ⎪
   fmadd   %f4,  %f1,  %f2,  %f3   # perform calc           ⎪
   mfvd    %v2,  %r31, %f4         # move to vec reg        ⎬ Scalar loop
   addi    %r30, %r30, 8           # increment count        ⎪
   addi    %r31, %r31, 1           # increment pointer      ⎪
   cmpl    0,    1,    %r30, %r10  # are we done?           ⎭
   blt     begin_sloop

done_sloop:
   vivasud %v2,  %r28, %r3                                  } Vector store
```

Figure 3.5: Hand written assembly code for triad microbenchmark `z[j] = x[j]` $\times$ `factor + y[j]`.

Chapter 4 goes into more detail on ViVA programming, including specific techniques used to obtain the most performance benefit.

3.2.4 Cost of ViVA

ViVA is an additional feature that can be added to existing microprocessors. It can supply benefits, but it has additional costs, as well. Those costs can be split into three types: complexity of both design and verification; chip area; and power consumption.

Complexity

ViVA contains a relatively small amount of new hardware, and the basic components of that hardware are similar to existing components. The buffer itself is similar to existing register files, and interfaces to other components similarly to the way that caches do. There are only a few new instructions needed, and those instructions are not extremely complex. Because there are no automatic consistency ordering guarantees, the relationship between existing scalar accesses and ViVA accesses is simple.

Because ViVA is an additional component, it does increase overall chip complexity. The increase, however, is not large.

Area

ViVA adds new area to the chip, in the form of the ViVA buffer and interface queues. ViVA also has some minor secondary effects on area, in the form of the few additional instructions and control registers, but those effects are extremely small.

ViVA has not been implemented in real hardware, so true measurements of its area cannot be made. In order to obtain an approximate value of the area required for ViVA, I compare to existing structures implemented in real chips, and I obtain an estimate from a specialized modeling tool, CACTI [TMJ07].

Table 3.5 lists the microprocessors used to estimate the size of the ViVA buffer. All of the microprocessors were fabricated in 65 nm technology. While all three are flagship microprocessors, they encompass a relatively wide variety of approaches: the number of cores varies from two to eight, the clock rates vary from 1.4 GHz to greater than triple that amount, some of the machines include simultaneous multithreading, and the machines represent both in-order and out-of-order designs.

Table 3.6 shows the characteristics of the level one instruction and data caches of the microprocessors listed in Table 3.5. Just as the microprocessors exhibit a wide variety of features, so do the individual caches. The size of the caches varies from half of a square

Processor	Area	Clock rate	Cores	Type
Niagara 2	342 mm^2	1.4 GHz	8	8-way SMT, in order
Opteron	285 mm^2	2.8 GHz	4	out of order
POWER6	341 mm^2	4.7 GHz	2	2-way SMT, in order

Table 3.5: Microprocessors used for comparison for ViVA size. All processors fabricated in a 65 nm process.

millimeter, to over four square millimeters. The reason for the variability can be seen in the other characteristics: the caches have different amounts of set associativity, ports, and other features.

Processor	Cache	State	Dimensions (mm)	Area (mm^2)	Set Assoc.	Line Size	Banks	Ports
Niagara 2	Instr	16 KB	1.15x0.56	0.64	8	32B	1	1 rw
	Data	8 KB	0.95x0.52	0.50	4	16B	1	1 rw
Opteron	Instr	64 KB	2.96x1.09	3.21	2	16B	1	1r, 1w
	Data	64 KB	2.96x1.47	4.35	2	64B	8	2 rw
POWER6	Instr	64 KB	1.26x1.27	1.60	4	32B	4	1 rw
	Data	64 KB	1.29x2.01	2.59	8	32B	4	1r, 1rw

Table 3.6: Characteristics of L1 caches from recent microprocessors, used to estimate ViVA size. Data from [Nia07, Kon04, Gro06, Phi07, DSC$^+$07, dV03, LSF$^+$07, PC07]; sizes and some features estimated from die microphotographs or inferred from sources.

None of the caches can provide an exact match for ViVA, but there are some useful comparisons we can make. First of all, notice that all of the caches are relatively small: even the largest is barely over 1.5% of the total area of its chip. We can get a better idea of where ViVA's buffer would fall along the range by looking at the characteristics its buffer would have. Image that we want to support, simultaneously, two operations between the ViVA buffer and DRAM (say, two loads, or a load and a store), as well as two arithmetic operations, each of which requires two sources and a destination. That sum total is eight accesses to the ViVA buffer per cycle. A default implementation might have two wide ports, each of which can read or write four elements per access, as described in Section 1.5.2. Since the ViVA buffer has software-controlled storage, it effectively has a set associativity of one. The total storage of the ViVA buffer depends on the implementation; a typical implementation might have 32 registers, each of which can store sixty-four 64-bit words, for a total of 16 KB of storage. Most of those characteristics place the ViVA buffer on the small side of the comparison caches; even allowing for more area for longer registers or more ports, the total area should not be much larger than a few square millimeters in a 65 nm process.

CACTI seems to verify that conclusion. Table 3.7 shows the parameters used to estimate the ViVA buffer characteristics, and Table 3.8 shows the results. The area estimated is less than a square millimeter.

Parameter	Value
Cache Size (bytes)	16384
Line Size (bytes)	32
Associativity	1
Nr. of Banks	1
Technology	SRAM
Technology Node (nm)	65
Read/Write Ports	2
Read Ports	0
Write Ports	0
Single Ended Read Ports	0
Nr. of Bits Read Out	256
Change Tag	No
Nr. of Bits per Tag	0
Type of Cache	Normal
Temperature (K)	400
SRAM cell and wordline transistor type	ITRS-HP
Peripheral and global circuitry transistor type	ITRS-HP
Interconnect projection type	Aggressive
Type of wire outside mat	Semi-global

Table 3.7: Parameters used to estimate ViVA buffer characteristics in CACTI.

Parameter	Value
Total area (mm^2)	0.725
Access time (ns)	0.771
Random cycle time (ns)	0.442
Total read dynamic energy per read port (nJ)	0.0496
Total read dynamic power per read port at max freq (W)	0.112
Total standby leakage power per bank (W)	0.0597

Table 3.8: CACTI results.

3.2.5 Access Time

Unlike chip area, access time is not a cost that is paid when ViVA is added to a system; rather, it is a cost associated with every access. As with area, the best measurement would be derived from the process of actually designing a chip that used ViVA. Fortunately, the same methodology that provides an estimate of area can also give us an idea of the time needed to access the ViVA buffer.

CACTI includes an estimate of total access time, as well as random cycle time. Table 3.8 shows the estimated values. The total access time corresponds to the time it takes one access to complete, from the time it reaches the interface of the buffer until the value is returned. Random cycle time corresponds to the minimum cycle time for accesses to the buffer; the corresponding clock rate is over 2.2 GHz, with each access taking two clock cycles.

3.2.6 Power

ViVA also has a cost in terms of additional power consumed. Again, since ViVA has not been implemented in a real chip, power consumption cannot be measured. I look at estimates provided by CACTI.

One important difference from the area cost is that the true power cost depends upon how ViVA is used. ViVA's area is fixed, regardless of how much it is used (or not). On the other hand, ViVA's power consumption will increase if it is used more — but, the power consumption of the scalar memory path will decrease, as ViVA accesses are used to replace scalar memory accesses. Therefore, the power consumption is much more difficult to analyze.

There are two basic ways that ViVA accesses can save power over scalar accesses: first, because ViVA instructions are vector instructions and use vector processing, and second, because ViVA memory accesses replace scalar memory accesses. Vector processing can save power over scalar processing in general because of the reduced number of instructions fetched, decoded, and issued. Additionally, the actual processing of individual elements is simpler: things like dependency checks, setting up operands sources and destinations, and so on, can be executed once for many elements, instead of once for every element.

ViVA gives an additional benefit over scalar execution: the ViVA memory model is simpler than the scalar model, and so the hardware can be simpler and lower-power, as well. The scalar unit maintains the appearance of sequential ordering of memory instructions, even as it executes them out of order. The hardware required to support the ordering includes reorder queues and checks for each level of the memory hierarchy. In comparison, ViVA can process memory elements while performing many fewer checks.

Besides having a simpler coherency model, ViVA can save power by allowing the programmer to use a software-controlled scratchpad, instead of a system-controlled cache with prefetch. First of all, a scratchpad can avoid cache hardware overhead, such as tags and the checks that are performed with them. More importantly, though, a software-controlled scratchpad allows the programmer to completely control its contents, which means that the program can ensure that elements that will be used in the future remain in closer storage, allowing the system to avoid a later power-hungry fetch from the next cache level (or, perhaps, from DRAM). One example of this that will be explored in more detail later is unnecessary cache prefetch accesses: a hardware cache prefetcher attempts to load values before they are requested, but sometimes those values are not used. Hardware prefetchers deliver better performance if they fetch large amounts of data as soon as they think they recognize a stream; unfortunately, that is precisely the behavior that maximizes useless prefetches. Useless prefetches waste power in their own right, and they evict other data that might actually be used, leading to even greater power waste.

Finally, the addition of ViVA could possibly save power by allowing an implementation to achieve a certain level of performance, while using a slower clock or simpler scalar design. Obviously, the exact amount of power saved in those cases depends on the specific design chosen; the power saved cannot be estimated without knowing the specific design used.

We can get an idea of one aspect of ViVA's impact on overall power by looking at an estimate of its power consumption — ignoring the potential savings that we might also get. Once again, CACTI provides us with an estimate, shown in Table 3.8. For two ports, the reported value is equal to a maximum of less than a quarter watt.

3.2.7 Summary of ViVA Advantages

ViVA provides a number of advantages over a system that uses only traditional scalar processing. One of the most important is that ViVA provides many of the advantages of using vectors: memory instructions are compact, expressive, scalable, tolerant of latency, and easily programmable. ViVA is even easier to program than traditional vector computers, since vector-scalar transfers do not incur a large penalty. Vectors are a well-established technique, having mature compilers in widespread use. ViVA is slightly different from full vector implementations, but in practice the differences are not difficult to handle. Only the memory operations in ViVA are vector operations, as opposed to a full vector system that would include arithmetic and logical vector operations, but memory operations are often the most critical feature in processors, consuming large amount of power and limiting performance.

ViVA is simple to implement. The required hardware is simple, and most of the changes are located outside of the processor's core. ViVA needs only a small amount of ISA changes.

The ViVA programming model allows for mixed vector and scalar accesses, which simplifies the programming for many applications. In some cases, fences are needed to guarantee proper ordering, but often they are not.

While ViVA does not mean that the more expensive scalar hardware can be removed from a system, it does allow a program to make better use of it. ViVA works best with regular accesses that have a known or predictable access pattern. By using vector instructions to perform those transfers, the scalar hardware is left free to perform irregular accesses. Prefetch analysis, coherency analysis, and other associated power-hungry hardware such as reorder queues can still be used when there is no simpler alternative that will work well, but ViVA provides a cheaper alternative for a large number of operations.

3.2.8 Summary of ViVA Disadvantages

As with most engineering solutions, ViVA is not a perfect answer — it has some costs and tradeoffs that must be considered. One obvious disadvantage as compared to full vector computers is that ViVA only provides vector memory operations — arithmetic and logical operations must be performed in a scalar fashion. Not only is there no vector advantage, there is the additional cost of a transfer from the ViVA buffer to scalar registers, and back, that must be performed before non-memory operations are executed on data. This means that, as opposed to the reduction in instruction count that is typically observed with vector programs, ViVA applications will typically execute more instructions than a scalar implementation. (There will be a large reduction in long, expensive memory instructions; they are made up for by scalar transfer operations, which are shorter and simpler.)

There is actually a case where ViVA can help arithmetic performance: since ViVA can improve achieved memory bandwidth, a memory-bound application can make better use of its arithmetic units if it can use ViVA to improve memory performance.

The ViVA buffer comes with costs of its own. It is additional hardware, which has design, verification, area, and power costs. Fortunately, the buffer is relatively simple, and located outside of the processor core, so the costs are not large.

3.3 Extending SIMD to Full Vectors

ViVA improves performance by vectorizing memory operations, but is not a full vector implementation. This section explores full vector microprocessors; in particular, it describes the justification and process of moving from an existing scalar-only microprocessor ISA to a full vector computer.

3.3.1 Why Extend Existing ISAs?

Microprocessor development is undergoing reconsideration. The diminishing benefits of increasing instruction-level processing, combined with the limits of power dissipation, increasing relative memory access latency, and decreasing benefits of new semiconductor technology generations, described in Chapter 1, are causing designers to embrace new designs that can continue to deliver better performance. Vector processing is a natural solution for allowing processors to take advantage of data-level parallelism in a scalable, efficient manner.

On the other hand, there are some good reasons to take advantage of existing ISAs. Established systems have large support bases of applications, compilers, and users. Besides the obvious business interests of maintaining an existing ISA, adding on to existing systems means that the software development work that has been put into the system is not lost.

Adding vectors to an out-of-order microprocessor can give benefits, as well. Vectors work well with data-level parallelism, but cannot use other forms of parallelism such as thread- or instruction-level parallelism. Other techniques can capture more parallelism from applications, but require more power and are not efficient when used as the only parallelism technique. Combining existing microprocessor designs with vectors will allow the resulting processor to use vectors to efficiently take advantage of available data-level parallelism, and use multithreading or out-of-order processing to capture some of the remaining parallelism, with less total power consumption. Previous work has shown that vectors works well with both out-of-order [EVS97, VEV98, QCEV99, EAE$^+$02] and multithreading [EV97, Chi91, Jes01, KBH$^+$04] technologies.

By building upon an existing microarchitecture, designers can avoid a problem that has plagued some vector computers: poor overall performance due to scalar limitations.

Amdahl's Law [Amd67] says that performance improvement is limited by the fraction being improved; even if vector instructions make up the majority of a program, poor scalar performance will limit the overall speed. A microarchitecture that achieves good scalar performance will reduce the limiting effect.

The techniques that are used to extending existing ISAs can also be used to help design ISAs that are more amenable to extension in the future. By looking at the issues that cause difficulty or inefficiency for existing ISAs, a designer can avoid or reduce including those features. Often, designing with future extension in mind does not even involve any real negatives (or, perhaps, only very small costs), but can save lots of difficulties later.

Finally, existing ISAs are here to stay. The 80x86 ISA, first introduced in 1978 as a 16-bit microprocessor ISA, has seen many modifications, and continues to remain popular. It was extended to 32- and 64-bits, and has been modified to include support for things such as SIMD [PW96, RPK00] and virtualization [AA06]. PowerPC, another long-lasting microprocessor ISA, has been around since 1990 and also experienced many modifications. Figure 3.6 shows the number of instructions in x86 and PowerPC chips. Although the microprocessors have remained backwards compatible — able to execute programs written on the very first implementations — they have added an average of more than one instruction every month since they were first introduced. If ISAs are going to stick around for 30 year or more, it is worthwhile to consider how to continue to improve them.

3.3.2 Adding Vectors

There are a number of ways that vector instructions can be added to existing processors. One method is to add instructions to available space in the existing ISA, using either unallocated space (for example, as was done with the IBM System/370 vector extension [Buc86] and Vax vector facility [BB90]) or pre-defined extension space (as how the VIRAM1 ISA was treated as a MIPS coprocessor extension [Koz99]). My approach is slightly different: it re-uses existing instructions in a way that lets them operate as vector instructions, but

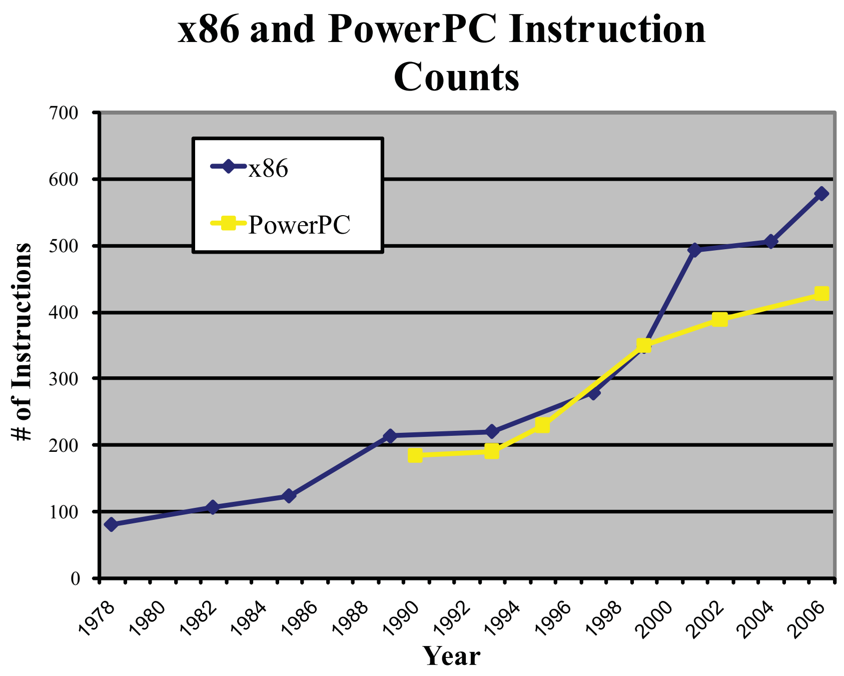


Figure 3.6: Instruction counts of x86 and PowerPC ISAs. Details in Table 3.9.

remain compatible with their existing use in non-vector programs. By building on an existing SIMD ISA extension, we can take advantage of an existing set of rich instructions, and ease transition from SIMD to vector programming.

There are four main changes required to change SIMD extensions to vector instructions, described in more detail below:

- turn SIMD registers into vector registers;
- add new memory instructions;
- add flag registers;
- finally, add a few new and missing operations.

Figure 3.7 shows the existing and added state.

Vector registers

The first step towards a vector extension is increasing the length of the existing SIMD registers, and adding the control registers necessary to take advantage of that increase. The

Year	Instruction Counts: x86	PowerPC	Notes
1978	80		8086 (without FPU)
1979			
1980			
1981			
1982	106		80286 (without FPU)
1983			
1984			
1985	123		80386 (without FPU)
1986			
1987			
1988			
1989	213		80486DX
1990		184	32-bit POWER ISA
1991			
1992			
1993	220	190	Pentium (with FPU), 32-bit PowerPC
1994			
1995		229	64-bit POWER ISA
1996			
1997	278		Pentium MMX
1998			
1999	348	349	SSE, 32-bit PowerPC with Altivec
2000			
2001	493		SSE2
2002		388	64-bit PowerPC
2003			
2004	506		SSE3
2005			
2006	578	426	SSSE3 and SSE4, 64-bit PowerPC with Altivec

Table 3.9: Evolution of Intel x86 and PowerPC ISAs. Excludes x86 extensions from companies besides Intel, including SSE4a and SSE5, and virtualization instructions. SSE4 includes SSE4.1 and SSE4.2.

current length of SIMD registers in 80x86 and PowerPC is 128, which are used to represent a variety of different-size integer and floating-point values.

The length of the vector register is a tradeoff that depends on the target size of the chip, market area, and cost. Instruction overheads such as fetching, decoding, issuing, and dependency checking are amortized over the length of a vector, so longer vectors have a

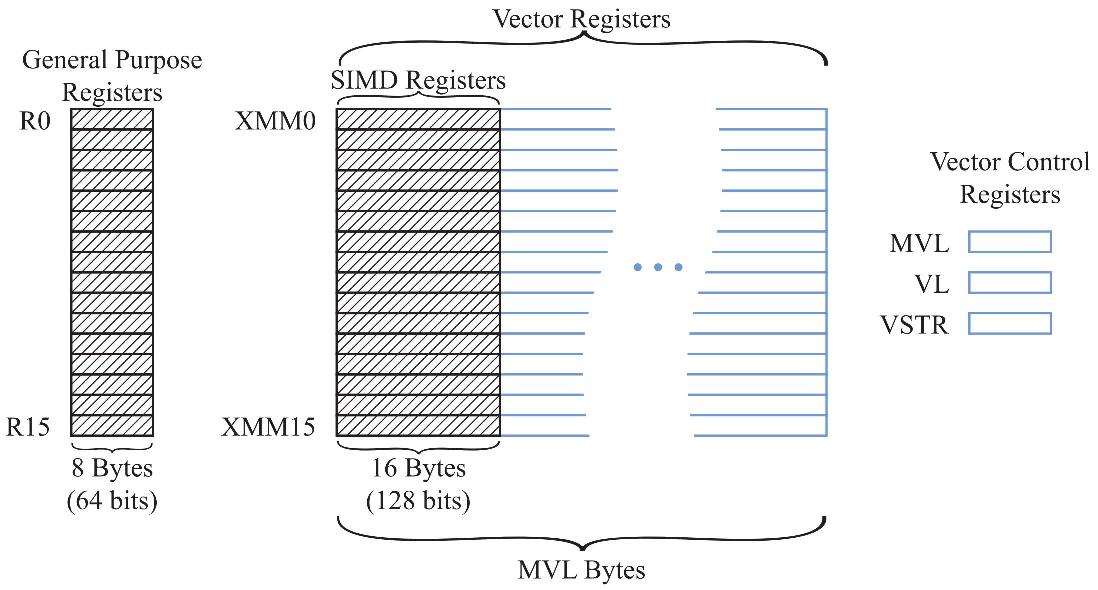


Figure 3.7: Current architectural state (shaded) and added state (unshaded).

lower per-element cost. If the system can handle the exposed concurrency, longer vectors are able to tolerate more latency, as well. On the other hand, longer vectors consume more area. To allow for flexibility, the architecture does not fix vector register length, so initial or low-cost chips can have shorter registers while high-performance implementations of the same architecture can have longer registers. The same binary executables will be able to run on both.

Large register files can present a number of challenges. Fortunately, all of the techniques used to mitigate those problems described in Section 3.2.2 work. There is one other technique that can work for full vector architectures: splitting the full vector register file into multiple lanes, as described in Section 1.4. The full register file is broken into blocks, which are distributed across the lanes; parallel accesses to the blocks provide additional concurrency. Since arithmetic operations generally occur between elements within the same lane, there is no need to provide a full crossbar between functional units and vector registers in different lanes.

It is possible to leave the registers short — at 128 bits — but at that length, the system will not be able to derive much benefit from vector semantics. On the other hand, it may make sense to have short vector registers in a low-cost microcontroller that does not need to be very fast, but is still binary compatible with full vector applications.

Once the SIMD registers and treated as vector registers, the processor needs to have a Maximum Vector Length (MVL) control register to communicate the hardware register length to programs, and a Vector Length (VL) control register for programs to communicate the number of elements on which to operate.

The traditional interpretation of MVL and VL is that they specify the number of elements on which to operate. By slightly changing the meaning of those registers, it is possible to add vector functionality and keep compatibility with existing applications. The problem arises because the default operation for existing SIMD instructions is to process 128 bits of data, regardless of the number of elements that corresponds to. That means that an instruction might operate on two double-precision floating-point values, or eight 16-bit integer values: the length is constant, but the number of elements is not. The solution is to re-interpret MVL and VL to refer to lengths, not number of elements; then, a default value can be chosen that matches the length of existing SIMD instructions. Old programs will continue to operate correctly, and new programs can change the value in VL to operate on more data.

Once MVL and VL are added to the processor, we can change SIMD instructions to respect their values. For most operations, it is very straightforward to modify them: for example, a SIMD single-precision floating-point add instruction normally operations on four elements (for a total of 128 bits); if the vector register length was 64 bytes (512 bits), the same instruction would simply perform an add on sixteen elements.

A few instructions are not as straightforward to extend. For example, Intel's SSE SIMD extension has a "SHUFPS" shuffle instruction. That instruction operates on 32-bit data: each of the four elements in the destination register can be filled with any of four input elements. A fixed field of eight bits (two bits for each of the four destinations) is included in the instruction. Extending the operation so that any destination element could be filled with any source element, as in a true shuffle, would obviously not work: the number of bits to specify the shuffle pattern would grow as the register lengths increase, and they would not fit in the fixed instruction field. The solution is to treat these instructions as operating on multiple 128-bit segments of data. The shuffle operation then works on each segment independently; put another way, the original 128-bit shuffle as a whole is replicated for the length of the new vector registers.

New memory instructions

After extending the register file and providing MVL and VL control registers, the next step is to provide vector memory instructions. Currently, SIMD instructions can only reference multiple memory elements as unit-stride accesses. That is not a large problem for existing systems: SIMD registers are short, so there is not much space to store more complicated access patterns, and unit stride accesses are common anyway. Once we have the capability of exposing more concurrency with longer vector registers, it makes sense to add the capability to take advantage of more complex, but still frequently used, memory access patterns.

As described in Section 3.2, memory accesses are critical to overall system performance. Vector memory accesses can improve performance while reducing power consumption. Vector memory accesses allow a program to describe its memory access pattern explicitly: a single instruction lets the hardware know that it can concurrently fetch many elements that the program will use. Scalar accesses do not contain any information about overall patterns; hardware prefetchers can try to find patterns, but they are forced to choose between being aggressive (and thereby wasting energy and reducing performance by fetching unneeded data, and evicting useful data) and being conservative (and not getting much performance improvement). Hardware prefetchers have to spend energy analyzing every memory access, to try to determine if it is part of a larger stream or simply an independent access.

Adding six new instructions: unit-stride, strided, and indexed accesses, for both loads and stores, lets more applications get the benefit of vectorization. Strided memory accesses are frequently used for accessing fields in arrays of structures. That includes scientific applications, which use structures representing physical objects and environments, and media applications, where the fields might represent the red, green, or blue values in an array of RGB-encoded pixels. Indexed accesses are useful for sparse matrices and other arbitrary patterns.

It would be possible to extend the existing SIMD memory access instructions into unit-stride vector instructions, but it is useful to keep the existing versions and add new instructions. By keeping the old version, the new memory instructions are free to use a different memory coherence model, or to be treated semantically differently — for example, SIMD memory accesses could be cached in the regular scalar cache, and vector memory references could bypass small L1 caches and use a separate, custom-purpose vector cache.

In addition to expressing memory access patterns, vector memory instructions can describe source and destination element size. Many media applications operate on data at one size and store it at another — say, loading 16-bit values and operating on them as 32-bit values. Similarly, many scientific applications operate on double-precision values that are stored as single-precision values. Vector instructions can either encode the element source and data sizes in instruction fields, or refer to a control register that describes the necessary conversion; data alignment can then be performed automatically on loads and stores.

Flag registers

The last piece of new state needed to support vectors are flag registers. Existing SIMD implementations have only basic, if any, means of providing conditional execution. It is much simpler for a compiler to vectorize conditional statements (such as if-then-else) in loops if the processor has the capability to perform conditional execution. There are various ways to vectorize conditional statements, but using masks and flag registers is one of the simplest and most flexible [SFS00]. Each flag register contains a single bit corresponding to each element in a regular vector register: a ‘1’ bit enables execution for the corresponding element, and a ‘0’ bit inhibits execution.

Because vector registers can store different size elements, one bit is needed for each of the smallest size elements that the system can handle; if it is possible to execute directly on bytes, one bit of storage in the flag register per byte in the vector register is needed.

As with the MVL and VL registers, an implementation can allow for full vector functionality, while remaining compatible with existing code. Existing instructions do not have a way to specify a mask, and do not currently have a form of conditional execution. To remain compatible with that model, the new implementation would execute every vector instruction under a mask (so that no mask need be specified), and have that mask default to enable operation on all elements.

Some vector implementations have a rich set of instructions to operate directly on mask registers. Those instructions can simplify flag processing, but they are not absolutely necessary: the same operations can be performed in the regular vector registers, with the results transferred to the flag registers. By performing most of the flag processing in regular vector registers, the number of new instructions needed can be kept small. New instructions to transfer flags between the regular vector registers and the flag registers are needed, and it can be useful to add in a small set of operations that do work directly on the flag registers. Instructions to clear and set a full flag register directly, as well as move the entire contents of one flag register to another, can speed up flag processing dramatically.

One simple way to allow mask processing in the regular vector registers is to treat any non-zero value as a logical ‘1’ in the mask. When a mask is moved from a flag register to a vector register, a ‘1’ bit gets stored as all 1’s in the vector register; when a mask is moved from a vector register to a flag register, any non-zero element is stored as a 1.

While only one default mask would be used to provide conditional execution, it is useful to have more mask registers available. Multiple flag registers allow masks to be stored, and allow different complex masks to be built out of multiple subexpressions. Another useful reason to have multiple flag registers is to encode information about exceptions. A dedicated set of flag registers, one for each exception type, can be used to maintain per-element records of exceptions. As vector instructions are processed, any element that causes an exception that does not demand immediate action (for example, a floating-point underflow, as opposed to a page fault) can set the corresponding bit in the appropriate flag register. The resulting mask can then be used to process floating-point exceptions in a vector manner. To continue with the example of floating-point underflow: a processor can execute a block of code, and then check at the end to see if any underflows occurred. If so, the processor can use the underflow flag register as a mask to perform fixup (say, scaling elements) using vector instructions that affect only the appropriate elements.

New and missing operations

The final component needed to support vector execution is to add ISA support for any "missing" operations. Current SIMD instruction sets are often not very orthogonal, offering some operations only for certain sizes or data types. It would be possible to extend only existing instructions, but it is easier to vectorize more programs with operations such as double-precision floating-point, which is not currently supported on PowerPC's SIMD extension.

There are some vector-specific instructions that are not strictly necessary, but can speed performance considerably in some cases. A good example is adding insert and extract instructions: without some mechanism to support cross-lane data transfer, certain operations become extremely difficult, or impossible to execute without resorting to shuffling data in DRAM — a slow, power-hungry process. Insert and extract operations are limited primitives and thus relatively easy to implement, but they can be used to build useful solutions to a number of common problems. An extract operation moves values from an arbitrary start position in one vector register to the beginning of another vector register; an insert operation moves values from the beginning of one vector register to an arbitrary start position in another.

An example of a common operation that can be built from inserts and extracts is a vector reduction: often, a single result — such as a sum — has to be generated from a large amount of data. Typically, the initial steps can proceed in the regular vector pattern: a chunk of memory is loaded, and added to an accumulator register. Eventually, though, the intermediate results to be added are only as long as a single vector register. At that point, a reduction operation is needed to add those values. An extract operation can move the second half of the accumulator vector register to a different register; VL can be set for the shorter length, and the two registers can be added. That process can be repeated to eventually produce a single result.

Another useful operation that can be built from inserts and extracts is rotations. If an extract operation specifies a starting location, a length that is a full vector register, and the extract wraps around the end of the source register, the extract instruction will actually perform a left rotation. Similarly, an insert operation will perform a right rotation.

3.3.3 Drawbacks

The process of starting with an existing ISA and adding an extension to it has some drawbacks, compared to starting over with a completely new ISA. The biggest problem, in general, is that you are stuck with what has been called “cruft”: all of the parts of the ISA that are no longer needed or appropriate, but must still be supported. Hopefully, those parts at worst serve as a nuisance; in some cases, they can actually interfere with the new purposes of the ISA.

An example of a previous limitation is the short length of the existing SIMD instructions, which dictates the default length of VL. Fortunately, the only real effect that this has is that programs have to set VL upon startup, instead of relying on VL being set to MVL.

Perhaps a bigger issue with the technique of extending SIMD instructions into full vector instructions is that it is difficult to tell how well applications would be able to be modified. Is it possible that an individual application could be "ported" to the new fully vectorized system without a lot of difficulty, or would there be no advantage over starting from scratch? It seems as though there would be an advantage to having most of the system remaining identical; unfortunately, to get a true understanding, it would be necessary to build a system and implement a mature vector compiler, a feat outside the scope of my work.

3.4 Summary

This chapter laid out the basic ideas in my research. I take a look at two different systems for combining vectors and existing microprocessors in a simple, low-cost way. The first is ViVA, which adds a memory buffer that acts a vector register file, and vector memory instructions that can transfer data between DRAM and the new buffer. In keeping with the goal of simplicity, no vector datapaths are used: individual elements are transferred from the ViVA buffer to the existing scalar core, and operated on there.

The second system I take a look at is a method for converting existing SIMD ISAs to true vector ISAs. The technique converts SIMD instructions to vector instructions, but allows existing software to remain compatible.

In the next chapters, I take a look at the infrastructure used to evaluate ViVA. First, I describe the benchmarks used to judge its performance, describing their behavior and why they were chosen. Next, I describe the full-system simulator that I used to execute the applications, including the modifications that I made to model ViVA.

Chapter 4

Software Techniques, Benchmarks, and Dwarfs

Vectors offer a different programming model than scalar computers, and so they have different programming and optimization techniques. ViVA has vector memory instructions, but does not perform vector arithmetic operations. This chapter goes over the software techniques used on ViVA, focusing on how they differ from standard scalar methods and regular vector models.

This chapter also covers the benchmarks and software that use those programming techniques. In Chapter 6, I evaluate ViVA's performance on a number of small applications. In this chapter, I describe those applications, and why they were chosen. I first describe the simple microbenchmarks that are used to measure the basic features of the test system, such as maximum bandwidth and latency, and then describe the reasons for selecting my particular benchmarks.

4.1 ViVA Software Techniques

Many of the software programming techniques used with ViVA are common to other architectures. After reviewing those methods, this section describes techniques that work particularly well on ViVA, and techniques that are specific to ViVA's combination of vector memory references and scalar execution programming model.

4.1.1 Regular Vector Techniques

Vector computers typically require some programming techniques that are slightly different from the techniques used in scalar programming. This section covers a few basic ideas that the benchmarks use, or are important for understanding how ViVA operates; for a more general overview of vector techniques, see [AHP06] or [Wol95].

Basic vectoriziation

In the simplest vector code example, a loop iterates over a block of data, performing arithmetic calculations. The vector equivalent of that code simply turns the arithmetic statements inside the loop into vector statements. Figure 4.1 shows an example.

```
for(i = 0; i < 64; i++) {
       z[i] = x[i] × fac + y[i];
}
```

(a)

```
vload   %v1,   x
vmuls   %v1,   %v1,   fac
vload   %v2,   y
vadd    %v1,   %v1,   %v2
vstore  %v1,   z
```

(b)

Figure 4.1: Simple vectorization example. (a) shows C code for a triad loop; (b) shows the vectorized version.

Unfortunately, most of the time, real code is not nearly as simple as the example.

Strip mining

One issue that slightly complicates vectorization is the limited length of physical vector registers. Real applications use vectors of various lengths. Any vectors that are the same length as physical registers, or shorter, are easy to deal with: the program simply sets VL to be equal to the vector length, and executes the instructions. If the vectors are longer than the register lengths, the vector will have to be processed in multiple parts. The process of executing a vector loop in multiple parts is called “strip mining”.

Vector architectures normally do not have a maximum architectural vector length, but they often have a guaranteed minimum length: programs are then able to vectorize short loops with a single statement, without having to resort to strip mining.

Other vector techniques

There are a number of other techniques that are used for vector programs. Most of the other techniques deal with code that is not as ideal as the example given here, in which some feature of the code prevents direct vectorization. Often, these techniques allow vector programs to run efficiently in cases where the original code looks like it might not be amenable to running well on a vector architecture.

Many of the techniques allow the processor to handle loop-carried data dependences. In general, the difficulty arises when a value is written in one iteration of the loop, and used in a later iteration. Compilers will move scalars out of loops, split a single loop into multiple loops, or vectorize part of a loop to obtain better performance.

Another case that can cause problems for direct vectorization is nested loops. One simple approach that can work well is simply to vectorize the inner loop. Often, a vector computer will give better results if a slightly different approach is used: simply interchanging the inner and outer loops will sometimes allow the vector compiler to use longer vectors, or obtain a better stride. In some cases, the outer loop itself can be vectorized.

4.1.2 Basic ViVA Approach

The basic ViVA programming model is largely based on vector programming. The most straightforward version can be summarized easily:

- vectorize as normal, keeping vector loads together at the top of a basic block, and stores together near the bottom;
- for each basic block, insert a scalar loop between the loads and stores. Inside the loop, iterate over a vector's length of elements, first transferring elements from the ViVA buffer to scalar registers, then performing arithmetic or logical operation, then transferring back.

The basic method can usually be improved quite easily. Often, defaulting to a double-buffered approach is not more difficult and can improve performance. Double-buffering logically splits the ViVA buffer into two parts: one that the processor uses to perform long-latency loads, and another part that the processor uses for arithmetic calculations. Double buffering helps performance by overlapping memory transfer latency with calculations.

4.1.3 ViVA-Specific Techniques

ViVA's differences from traditional vector processors cause its programming model to be different, as well. The differences also create some opportunities; that is, new ways to use ViVA that are not possible in most vector systems.

ViVA's main structural difference from traditional vector processors is that it includes a buffer as opposed to vector registers. Programs transfer elements from the buffer to scalar registers, where arithmetic and logical operations are processed. Since ViVA does not have vector datapaths associated with partitions of a register file, processing is not as constrained.

Processing of sparse matrices is an example that illustrates the combination of regular memory access streams and irregular calculation: the matrix stores elements in a dense vector, allowing the use of unit-stride loads, but row boundaries occur at irregular locations. Thus, a row may contain just a few non-zero elements, or it may contain more elements than fit in a vector register. Section 4.5 goes into more detail, illustrating ViVA's operation on a Sparse Matrix - Vector multiplication kernel.

The same idea can be applied to the case where memory has to be shuffled in an irregular pattern. The processor can transfer a block of memory to the ViVA buffer using unit-stride transfers; from there, it can move individual elements to a scalar register, and then back to an arbitrary location in a vector register.

While ViVA does allow irregular execution patterns, they can incur a cost. As described in Section 3.2.2, most accesses to the vector register file follow a regular pattern: usually, each successive element is accessed in order. The register file can then multiplex multiple sources onto fewer wide ports. If accesses actually occur in an irregular pattern, more port conflicts occur, and the program proceeds at a slower rate. The number of conflicts depends on the particular access pattern, as well as the width of the multiplexed ports.

ViVA gives another advantage over traditional vector processors that is related to its use of scalar processing: the ability to combine both vector and scalar memory accesses. While traditional vector computers have both types of accesses, usually it is not possible to use both in an efficient manner. Data transfer between separate vector and scalar cores can be slow, making it ineffective to operate on the same data in both cores.

ViVA's organization places its buffer outside of the core. Before performing calculations on data elements, the application must first transfer them to regular scalar registers. If, on the other hand, no calculations need to be performed — if a block of memory is just being moved — the application can load data into the ViVA buffer using vector operations, and then store it back to a different location, without ever moving it to the core. Irregular shuffles require intermediate transfer of data to the core, but a block of memory can be loaded using one type of transfer and stored with another: for example, the Corner Turn benchmark, described in Section 4.4, loads data with unit-stride operations and stores with strided instructions to transpose a matrix.

Often, vector architectures have specific address registers that are used to specify base or offset registers for vector memory operations. ViVA uses general-purpose registers for this purpose, which can increase register pressure, especially if all of the registers in the

ViVA buffer are being used. One solution is to replace the straightforward approach, which uses a single base register and multiple offsets, with a version that reuses offsets with multiple bases registers. For example, Figure 4.2 shows code that uses a total of six scalar registers to perform nine vector loads: three of the scalars are used as base registers, and three are offsets that are used with each of the bases. A larger, more typical real example might use four or eight bases and four offsets, to allow a total of 16 or 32 vector loads.

```
vivalud  %v0, %r10,%r20   # base 0, offset 0
vivalud  %v1, %r10,%r21   # base 0, offset 1
vivalud  %v2, %r10,%r22   # base 0, offset 2
vivalud  %v3, %r11,%r20   # base 1, offset 0
vivalud  %v4, %r11,%r21   # base 1, offset 1
vivalud  %r5, %r11,%r22   # base 1, offset 2
vivalud  %v6, %r12,%r20   # base 2, offset 0
vivalud  %v7, %r12,%r21   # base 2, offset 1
vivalud  %r8, %r12,%r22   # base 2, offset 2
```

Figure 4.2: Code using combinations of multiple base and offset scalar registers to load larger number of vector registers.

4.2 Synthetic Benchmarks

The first programs run on ViVA are synthetic benchmarks; that is, simple kernels used to test basic performance of the primitive operations. These programs do not do any real work on their own, but they are useful to study: they can provide insight into ViVA's performance and behavior, and sometimes they can represent parts of the basic memory access patterns that are used in real applications.

4.2.1 Unit Stride

The unit stride benchmark is the most straightforward application: it measures the performance of reading and writing long blocks of consecutive elements. Real applications use unit stride memory accesses extremely frequently.

Often, processors have specific optimizations for unit stride memory accesses. It is relatively easy to include hardware optimizations that significantly improve performance of unit-stride streams, such as including a hardware prefetcher. Additionally, real applications see a large benefit from having faster unit-stride accesses, because those streams are so common.

In order to ensure that unit strides are actually performed, the version of the benchmark used in ViVA executes a triad, which consists of scaling one input stream by a constant, adding its elements to a second input stream, and storing the result in an output stream: `z[j] = x[j] × factor + y[j]`. Since the elements are double-precision floats, the total byte-to-flop ratio is 24 to 2 (or even 32 to 2, if the store requires a cache line fill read first). For most modern architectures, performance on unit-stride triads will be limited by memory bandwidth, not by computation bandwidth. All of the memory accesses are independent, minimizing the effects of latency.

4.2.2 Strided

Strided transfers involve accessing elements in memory with a constant displacement. The offset between elements is the stride, so that a memory stream with a stride of two accesses every other element.

Strided accesses are relatively common in real applications. One common case that uses strided memory operations is accessing an array in its non-major direction (e.g., accessing a standard C row-major array in a columnwise fashion). Another typical example is accessing particular fields in an array of structures; a related case is accessing particular subpixels, such as the red parts of all RGB pixels in a graphics application.

Strided accesses present a number of challenges to the memory system. Caches operate on full cache lines, consisting of multiple elements. For data streams with spatial locality, including common unit-stride streams, pulling in a full cache line works well; for strided accesses, some — or, perhaps, all but one — of the elements brought in (thereby consuming available bandwidth) will not be needed.

A strided access also presents difficulties to a memory system because a single strided stream can touch a large span of memory addresses. A larger range of memory addresses puts pressure on the TLB, and can cause additional cache and memory bank conflicts.

4.2.3 Indexed

Indexed transfers, sometimes called scatter-gather, involve accessing memory elements at arbitrary locations. The transfers use a vector to specify the address each element, usually in conjunction with a base address. Indexed accesses are typically used for cases where element access is not regular. For example, sparse matrix operations operate on a dense array of nonzero elements. Another array specifies the original column index for each of the elements, which is then used to retrieve a value from a separate source vector. Because the matrix stores nonzero values by rows, the index coordinates do not have a regular pattern. An indexed memory access, using the column index as offsets, will retrieve the proper elements of the separate vector.

In some cases, indexed accesses are useful for streams that exhibit some regularity. Stanza patterns involve accessing blocks of memory: each block consists of a number of consecutive elements, and there is a regular offset between successive blocks. If the stanzas are long, regular unit-stride accesses work well. If stanzas are short, on the other hand, they will limit the lengths of unit-stride memory accesses: unit-stride accesses can contain no gaps, so they are only able to reference one stanza at a time. Since indexed accesses do not have any constraint over element locations in memory, a single indexed access can refer to multiple stanzas. The downside is that an indexed access might incur a greater cost than a unit-stride access: a processor can usually perform simple optimizations for unit strides, such as minimizing memory address translations and performing accesses that operate on a full cache line at once. In the general case, indexed accesses cannot perform those optimizations without comparing every element address to every other address, but a processor can perform optimizations to handle the specific case of multiple stanzas as a single access: the processor can compare each element's address to those of its neighbors, and merge consecutive elements together using a temporary buffer.

Indexed accesses present challenges to the memory system that are similar to those of strided accesses. If anything, the difficulties are greater: indexed accesses can potentially request values from anywhere in memory, so the bandwidth and memory range issues still apply.

4.2.4 Striad

The last synthetic benchmark that I use is stanza triad, also called striad. The calculation used in striad is the same used for the previous synthetic benchmarks: `z[j] = x[j] × factor + y[j]`. The difference is that striad applies the calculation to stanzas of memory: the benchmark operates on a consecutive block of elements, skips ahead in memory, and then repeats.

The stanza access pattern does occur in real applications. If an application performs an operation to a subblock of data that is part of a larger two-dimensional array (or a two-dimensional slice of an array with three or more dimensions), the resulting pattern looks like stanza accesses. Some computational techniques, such as adaptive mesh refinement [BC89] and cache blocking [AG88], naturally break a larger problem into smaller subblocks; accessing the unit-stride subblocks in a regular fashion, as in stencil methods [FW60] and other computational patterns, leads to a stanza pattern.

It is useful to include stanza memory access patterns as a basic benchmark, because they are simple patterns that represent streams found in real applications, but they are difficult for hardware prefetchers to recognize. Hardware prefetchers are able to optimize unit-stride and even strided patterns, but often drop to much lower performance once the memory accesses include discontinuities. Real applications make use of unit-stride and strided streams, but they often include additional irregular accesses that can prevent hardware prefetchers from working optimally; stanza benchmarks can give a much better estimate of delivered bandwidth in those situations.

4.3 Dwarfs

One of the results from the Berkeley View report [ABC$^+$06] mentioned in Section 1.2 was producing a group of Dwarfs, which we defined at the time to be patterns of computation and communication. The Dwarfs helped to motivate the benchmarks that I used to evaluate ViVA. This section describes the idea behind the Dwarfs, how they have changed since we first listed them, and how they relate to my work.

4.3.1 Theory

The Dwarfs from the Berkeley View are the product of an earlier talk, which defined seven important algorithms for simulation in the physical sciences [Col04]. Because there were seven of them, they were playfully called the "7 Dwarfs of Computational Science". The Berkeley View group was trying to identify a set of important algorithms that would be important in the coming years, and the 7 Dwarfs provided a starting point.

Because the Berkeley View group was interested in more than just scientific computing, we looked at a variety of areas: embedded computing, desktop applications, and scientific computing. Within each area, we looked at representative applications and benchmarks, and tried to see the prevalent patterns of computation and communication. Later, as we received feedback from other people, we included the search to include algorithms from databases, games and graphics, and machine learning [Pat07]. Table 4.2 shows the full list.

Eventually, we realized that our concept of dwarfs covered a wide range of diverse programming concepts [Keu08]. For example, "graph traversal" covers a number of algorithms that have a variety of patterns of computation and communication. The definition is still being debated, but the basic idea of dwarfs defining categories based on patterns —

Dense Linear Algebra
Sparse Linear Algebra
Spectral Methods
N-Body Methods
Structured Grids
Unstructured Grids
MapReduce
Combinatorial Logic
Graph Traversal
Dynamic Programming
Backtrack and Branch-and-Bound
Graphical Models
Finite State Machines

Table 4.2: Berkeley View Dwarfs.

whether the pattern is one of computation, communication, or a higher-level design — is still a useful way to organize application behavior.

4.3.2 Dwarfs and ViVA

In this dissertation, I focus on benchmarks that cover the areas that are the most relevant to ViVA:

- striad, which represents structured grid applications;
- the corner turn transformation, used in processing multidimensional data in spectral methods and structured grids; and
- sparse matrix-vector multiplications (SpMV), which represent sparse linear algebra.

Hence, I examine aspects of 4 of the 13 dwarfs in the evaluation of ViVA.

4.4 Corner Turn

Corner turn is simply a matrix transpose operation. It can be in-place, or store the result to a different output buffer. The algorithm itself is conceptually extremely simple, but it is very data intensive, and challenging for modern microprocessors to execute at high performance. The example code and matrix sizes used to evaluate ViVA are taken from the HPEC (High-Performance Embedded Computing) Challenge benchmark suite [LRW05].

A number of applications use corner turn, usually to re-align data so that later processing can take place with greater efficiency; in effect, performing a corner turn is paying the cost of rearranging data so that later accesses can occur with unit strides.

ViVA has a few different options for how to perform corner turn. The first method operates much like standard ViVA algorithms: the processor loads data into the ViVA buffer, moves elements from there to scalar registers, processes them, moves them back to the buffer, and writes them back to memory. For corner turn, the processor does not need to perform any actual arithmetic processing; the moves to and from scalar registers serve only to reorder elements. The second method simply moves data into the ViVA buffer using unit strides, then writes it out using strided access, never moving any element to the scalar registers.

Rearranging elements in the core allows the main benefit of requiring only unit-stride loads. The application brings a square block of data into the ViVA buffer: for however many vector registers are used, the same number of elements per register are used. Once the algorithm rearranges data, it uses unit stride stores to write the elements back out. This method follows typical ViVA programming style relatively closely, and uses unit stride stores, but it has two main drawbacks. The first is that it requires elements to be transferred to and from the scalar core, even though the core will not perform any calculations on the data. Those moves individually complete quickly, but there are a large number of them (one per element), which adds up to a large number of extra instructions executed, and ultimately consumes more power.

The second drawback to rearranging elements using the scalar core is that that method limits vector lengths. Typically, a vector can store many more elements than the total number of vector registers. If the application loads full vectors of data, there will not be a vector register to move the later elements to. Since vectors get better performance as they increase in length, using the scalar cores to perform corner turn can result in slow operation.

Performing corner turn without using the scalar core requires many fewer instructions, and can use full vectors lengths. The primary drawback is that the application can no longer perform both of the memory transfers with unit-stride accesses: either the load or store must be strided. Depending on the total size of the input data and the lengths of the vector registers, the scalar access can have an extremely large span in memory, leading to many TLB misses.

4.5 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication, SpMV, multiplies a matrix, typically denoted as A, and a vector x, adds the result to a vector y, and stores the sum back into y. x and y are stored as typical dense vectors, one-dimensional arrays in memory. In SpMV, A is stored as a sparse matrix: typically, most of the elements in the matrix are zero, and are not explicitly stored. The data structure that represents A stores the value and original coordinates of its nonzero elements. Sparse matrices are commonly used for simulations in the physical sciences.

There are a large variety of sparse storage formats [DER89, MSG91], but the one I examine is based on a common format called Compressed Sparse Row, or CSR.

CSR represents the sparse matrix by three dense one-dimensional arrays: one that contains all of the nonzeros, a second that contains the column indices of the nonzeros, and a third that contains pointers to the elements in the first array that start each row. Figure 4.3(a) shows pseudocode to perform SpMV with the sparse matrix in CSR.

SpMV, and sparse matrix arithmetic in general, can be difficult to perform efficiently on any machine. In vector machines, the difficulty largely arises due to the irregular nature of the nonzeros. There are actually two aspects that cause problems: the first is that each row can have a different number of nonzeros, requiring the application to repeatedly modify vector lengths; the second is that the number of elements per row might be much smaller than the physical lengths of vector registers, leading to short vectors and poor performance.

The complexities of dealing with sparse matrices are a large part of the reason that there are so many formats for storing sparse structures. Some formats pad shorter rows with zeros, to keep row lengths long; other formats permute row orders to keep rows of the same length near each other. Another common pattern is modifying the structure to store substructures of the matrix in a dense format, with diagonals or square blocks being typical.

ViVA is able to use a straightforward approach that gives the benefits of full-length vectors, without requiring modification of the matrix structure. The approach is similar to a simplified version of the algorithm used in Segmented Scan [BHZ93]. The basic idea is to always load a full vector's worth of nonzeros, and only perform row summing when necessary. Figure 4.3(b) shows the pseudocode.

```
for (i = 0; i < num_nonzeros; i++) {
   double sum = y[i];
   for (k = row_ptrs[i]; k < row_ptrs[i+1]; k++) {
      sum += nonzeros[k] * x[col_indices[k]];
   }
   y[i] = sum;
}
```

(a)

```
vec_reg0 = viva_load_unitstride(nonzeros_ptr);
vec_reg1 = viva_load_unitstride(col_indices);
veg_reg2 = viva_load_indexed(vec_reg1);
for (i = 0; i < mvl; i++) {
   double sum += vec_reg0[i] * veg_reg2[i];
   if(i == row_end) {
      y[row_ctr] += sum;
      row_end = row_ptrs[row_ctr++];
      sum = 0;
   }
}
```

(b)

Figure 4.3: Pseudocode to perform SpMV with the matrix stored in CSR format, for (a) standard scalar implementation and (b) ViVA.

The benefit of using ViVA is that it allows programs to use a combination of vector and scalar instructions. The example code uses unit-stride vector loads for the nonzero values and the column indices, and an indexed vector load for the source vector values. Those memory accesses are all regular, in that they can all be loaded together as full-length vectors. The later processing performs the multiply and sum using scalar instructions, as is required by ViVA, and then looks to see if the current row has ended. If so, the code writes the sum to the destination vector with a scalar store, and loads the next row end pointer. Because the ViVA and scalar loads and stores do not touch the same locations in memory, the code is able to mix them both without requiring synchronization barriers.

4.6 Summary

ViVA is able to make use of regular vector programming techniques, with a minor modification: compilers must replace vector arithmetic and logical operations with a loop that transfers elements to the scalar core, operates on them there, and then transfers them back to the ViVA buffer. The modification is simple, and should easily allow traditional vector compilers to target ViVA as well.

ViVA also facilitates new programming techniques that combine scalar and vector processing. Its unique architecture means that there is not a large penalty for arbitrary access to the ViVA buffer, and that programs can use both vector and scalar memory accesses without requiring high-overhead synchronizations.

To measure basic performance of ViVA, I use a series of synthetic benchmarks, aimed at testing memory access speed.

The first algorithm I use to evaluate ViVA, besides the synthetic benchmarks, is corner turn, which performs a transpose on a matrix. It is a very data-intensive kernel, used by applications such as multidimensional FFTs. The second code is sparse matrix-vector multiply, which applications such as large physical simulations use as part of many matrix operations. SpMV, representing the sparse linear algebra dwarf from the Berkeley View report, has a mix of unit-stride and indexed accesses, and combines processing in both regular and irregular patterns. The next chapter describes how I developed and calibrated the simulation environment.

Chapter 5

Development and Calibration of Simulation Environment

Evaluating ViVA's performance through experimental evaluation is an important part of determining its potential value. This chapter describes the simulation environment that I developed to execute and study ViVA applications. The first sections describe the simulator and the system it models, and the chapter concludes by presenting details of ViVA extensions to the simulator and calibration runs that compare performance to real hardware.

5.1 Simulation Environment

The simulator I use to evaluate ViVA is a version of Mambo, IBM's simulator of the PowerPC architecture, that I modified to include ViVA operation. In addition to modification of Mambo, I created a simulator test bench that allowed me to calibrate the simulator to a real system, organize and perform a large number of simulation experiments, and collect and analyze results. The modification to the simulator, not counting writing applications to run on it, took a total of approximately two years.

5.1.1 Mambo Overview

Mambo is a full-system simulator developed by IBM. It models Power and PowerPC systems varying from single-processor in-order embedded configurations through multiple-MCM SMT-capable out-of-order computers. Mambo operates in both a faster functional mode as well as a cycle-accurate mode. The simulator models all parts of the computer, including processor, memory, disk, and network.

5.1.2 History of Mambo

IBM originally based Mambo on the SimOS machine simulator [RBDH97]. Eventually, they rewrote the entire program and renamed the simulator Mambo. The code currently consists of approximately half a million lines of C, as well as some associated configuration files and scripts.

IBM uses the Mambo simulator to model and evaluate a number of production systems, including Blue Gene/L [BPE$^+$04] and Cell [PBC$^+$06]. Architects use Mambo in order to compare various microprocessor feature tradeoffs, enabling different features or varying parameter options and examining the resulting performance for a variety of applications. Manufacturers that use synthesized embedded microprocessors use the simulator to verify the correct behavior of their processors. Operating system and application developers can use Mambo to program for systems in development before hardware becomes available.

IBM continues to develop Mambo by adding new features. Recently, IBM has adding power simulation [SBP$^+$03] and parallel execution [WZWS08] capabilities. IBM has publicly released a version of Mambo, renamed Systemsim, as part of the Cell SDK [IBM08].

5.1.3 Mambo Operation

Mambo's core builds on tsim, an internal event library created by IBM. The actual Mambo simulator uses a single operating system thread, but tsim internally uses multiple threads

— called jobs to distinguish them from full OS threads — to model parallel execution. Jobs in tsim do not precisely correspond to parallel threads of execution at the program level, or parallel features operating at a hardware level: instead, jobs model parallel operation at a variety of levels.

Mambo uses tsim jobs to represent a single logical flow of processor instructions. An in-order scalar processor uses a single tsim job to process instructions; an out-of-order processor uses multiple jobs to allow parallel instruction streams to execute simultaneously in the simulation.

Mambo also uses tsim jobs to represent parallel operation in hardware: caches use jobs to process requests, with multi-port caches using multiple jobs to satisfy parallel requests simultaneously.

The Mambo code base produces a single binary executable that can operate either as a functional or cycle-accurate simulator. To accomplish this, individual jobs make requests to a central arbiter for any resource that they access. The arbiter can turn resource limitations on or off, or delay the requesting job for the appropriate length of time. For example, a job might request access to a floating-point functional unit. The arbiter will see if any are available; if so, it will allow the job to proceed, after the appropriate FPU execution latency has passed.

Simulations occur in a fast functional mode when the arbiter places no limits on resources. In the case of a job requesting access to an FPU, an arbiter in functional mode will always return the functional unit status as available, and report that the execution has no latency whatsoever. Depending on the simulator activity, simulations in functional mode typically process 2–200 times faster than in cycle-accurate mode.

The ability to turn limitations on and off, as opposed to producing a binary that operates in a single mode, produces a few benefits. First, the functional and cycle-accurate code bases are never unsynchronized, since there is only a single set of source files. Simulators that use separate sources to produce different versions of the executable require some additional mechanism to ensure that any changes or fixes that occur to one version are propagated to the other versions, as well. Another benefit of being able to change resource limitations dynamically is that the executable can run in a fast mode until it gets to a point of interest — say, after the operating system has booted, but before a benchmark program has begun execution — and then switch to an accurate mode to capture the full behavior of the program.

Finally, changing resource limitations individually allows programmers to model one part of the system in greater detail, while running the other parts of the simulation as quickly as possible. Often, a programmer will want to study the memory hierarchy behavior, including cache performance, with high precision, while not caring about the accurate timing modeling of individual arithmetic instructions.

There is a potential downside to using a single executable to perform both functional and cycle-accurate simulations: extra overhead. An optimized functional-only simulator would not need to perform any resource checks and could avoid all of the calls to the arbiter that are in Mambo. Mambo's flexibility adds some overhead to cycle-accurate operation as well, but that overhead is a smaller fraction of overall execution time than in the functional mode, since any cycle-accurate simulator has to model resource constraints.

Table 5.2 shows typical simulation speed for different types of activities. By comparing the heavy integer and floating-point execution rates of the simulator in functional mode, and treating the integer case as consisting only of processing overhead, it is possible to get an upper-limit bound on overhead. The result is that the simulator overhead for heavy floating-point activity accounts for less than 3% of total execution time. In reality, the overhead is less than that upper-limit estimate, and some of that overhead would be present for a less-flexible simulator, suggesting that the added cost of designing Mambo to execute either in functional or cycle-accurate mode is small.

Configuration options in Mambo fall into three categories: compile time, run time, and dynamic. Compile-time options include the things that affect the basic structure of the system: for example, adding new instructions requires recompiling the simulator.

Run-time configuration options mostly pertain to setting up the organization of the simulated machine. The number of processors in a system, the type of DRAM used, and similar features are run-time configuration options. Before Mambo begins execution of the modeled system, it fixes run-time options so that they cannot later be changed.

Finally, dynamic configuration options can be changed while a simulation is running. Dynamic options include most of the resource limitations in the system: latency of functional units, whether the system can execute instruction out-of-order, and so on.

A Mambo simulation begins with the execution of a Tcl script. The script sets run-time configuration options, fixes those options, configures virtual disks, sets the starting execution binary, and calls the simulator. The virtual disk allows the simulator to reference a disk image file and present it as a physical disk to the simulated system. For simulation runs that boot a full operating system, the starting execution binary is the OS kernel. Once the initial configuration is complete, the simulator sets up internal simulated processor structures, such as caches, and starts jobs for things like request queues. The simulator begins execution as a real system begins: it executes the system firmware and boots the operating system.

It is possible to execute Mambo simulations without booting a full operating system. In the standalone mode, the simulator executes a statically-compiled program without booting the operating system. Since there is no OS, the simulator emulates system calls. The standalone mode can be useful for some simulations, but since it does not include interactions with the OS, it does not reflect the full environment of actual application execution. In order to ensure maximum accuracy, I ran all of the ViVA experiments in a full simulation that booted the operating system.

5.1.4 Modeled System

To evaluate ViVA, I use Mambo to simulate a modern microprocessor. The system I simulate models an Apple Power Mac G5 system, chosen because it represents a relatively recent system and I am able to validate the simulator against real hardware. Table 5.1 lists the relevant features of the modeled system.

Processor	
Model	IBM PowerPC 970FX
Data width	64 bits
Clock rate	2.7 GHz
Execution	Out-of-order
Issue	Up to 5 instructions
Load/Store Units	2
Integer ALUs	2
Floating-point ALUs	2
SIMD ALUs	1
Memory	
Protocol	DDR (1)
Module standard	PC-3200
Memory bus width	128 bits
Memory bus peak transfer rate	6400 MB/s
Latency (approx.)	160 ns
North Bridge peak transfer rate	5400 MB/s (per direction)
L1 I Cache	
Line width	128 B
Storage	64 KB
Associativity	1
L1 D Cache	
Line width	128 B
Storage	32 KB
Associativity	2
L2 Cache	
Line width	128 B
Storage	512 KB
Associativity	8

Table 5.1: Details of the ViVA evaluation system model.

5.1.5 Modifications to Mambo

IBM designed Mambo to be configurable, but most of the configuration is either modifying parameters, or selecting one option from a short list of predefined choices. In general, it is not possible to modify the low-level behavior of Mambo. For example, it is possible to change cache size and to select between several cache replacement policies, but there is no way to use a completely new cache replacement policy.

In order to simulate ViVA using Mambo, I had to modify some of its behavior at a low level. Fortunately, IBM was willing to cooperate with my research by allowing me to modify the simulator. They could not deliver the full source code of the simulator to me, but they allowed me to spend a summer at IBM modifying the simulator source code. In order to maintain flexibility for later development, and to enable future modifications, we agreed that my approach should not be just to add ViVA functionality, but to modify the simulator so that it would be possible to make substantial changes later, without access to the source code.

In order to add ViVA and the ability to include other extensions, I modified Mambo for run-time extensibility by adding the ability to perform three significant tasks:

- add new instructions;
- create extra state associated with simulator objects; and
- control internal simulation function flow.

New instructions

Mambo normally decodes and executes instructions by simply using a series of nested arrays. Eventually, the simulator finds an entry that contains details of the instruction type, as well as functions pointers that it uses to execute the instruction. The function tables are generated with a script, and compiled into the simulator. The script creates the tables based on a complete list of ISA instructions.

The simulator modification that allows for run-time ISA modification has two parts:

replacing existing instructions and adding new instructions. Replacing existing instructions is straightforward: when the modified simulator receives a request to add an instruction, it first tries to decode the opcode in the existing table; on a match, the code modifies the existing entry with the new information.

Adding new instructions is not as simple. The modified simulator keeps the original table of compiled instructions to preserve high decoding speed. It also maintains a new set of tables. If the simulator cannot decode an instruction in the original table successfully, the simulator attempts to decode it in the new set of tables before raising an illegal instruction exception.

Requests to add new instructions include an opcode and a bitmask that identifies which of the bits in the opcode are significant in determining the instruction type. For example, an add instruction contains bit fields that encode input and output registers, but those fields do not change the type of instruction. In some cases, a group of related instructions can be encoded either as separate instructions or as a single type of instruction with mode fields. Operations that are encoded as separate instructions will have multiple entries in the decode table. For example, the PowerPC ISA includes "andi" and "andi." instructions; the period at the end of the opcode signifies that the instruction should affect the condition code register. If the two operations were encoded as a single instruction, the function that executes that operation would have to check the opcode to see if it should change the condition code register. If the two operations were encoded as separate instructions, the functions would not have to check.

In any case, the simulator looks at groups of significant bits while adding new instructions. If the bits represent an entry that exists in the runtime decoder tables, the simulator simply updates the entry with the new information. If the bits do not correspond to existing entries, the simulator creates a table that contains an entry for every combination of those bits — four entries for a group of two consecutive bits, eight for a group of three consecutive bits, and so on — and updates the newly-created entry that corresponds to the added instruction. Once the simulator creates a new table, it must ensure that appropriate entries in other existing runtime tables point to the new location.

For example, assume that a program adds two instructions: one in which the only significant bit of the opcode is the last, which must be equal to 0, and one in which the only significant bits of the opcode are the two last, which must be equal to 01. If the program adds the instructions in that order, the simulator will create two runtime decoder tables, each of which has two entries. The first table will index only the last bit of an opcode: it will contain the instruction information for the 0 entry, and a pointer to a second table for the 1 entry. The second table will index only the second-to-last bit on an opcode: it will contain instruction information for the 0 entry (which corresponds to the 01 instruction, since the last bit is guaranteed to be 1 for all entries in this table), and an invalid instruction for the 1 entry (which corresponds to an instruction that ends in 11).

If, instead, the program adds the instructions in the reverse order, the simulator will only create a single runtime decode table that has four entries. Three of the entries (00, 01, and 10) will contain instruction information and the last entry will contain an invalid instruction identifier.

The mechanism for creating runtime decode tables can exhibit poor performance for some odd cases of adding instructions: if the simulator adds an instruction that contains an extremely large number of significant bits in a single group, it will create an extremely large table. Fortunately, many of the cases that would lead to poor performance are not possible because of the significant use of opcode space by the original ISA: there are just not enough bits left to create problem cases. In practice, the mechanism tends to perform quite well, and allows instructions to be added at any time.

New state

New architectural features may require additional state. For example, ViVA requires an additional set of registers that make up the ViVA buffer. The code that extends Mambo at runtime can create the new state, but the difficulty comes in associating it with the correct existing simulator objects. Any new features that need to access state associated with a particular object must have some way to reference the new state based on a reference to the object. In ViVA, new instructions need to be able to reference the ViVA buffer based on the core that they are executing on — a multicore chip would have multiple ViVA buffers, so instructions would need to reference the correct instance.

The solution is relatively straightforward. Existing objects all have a globally unique identifier of some sort: the simulator enumerates some objects sequentially, such as cores, and other objects have a fixed memory address. The modified simulator creates state arrays for the enumerated objects, and uses the memory address to access a hash table for the other objects. The array or table links to new state. At runtime, programs can create state and add links to the appropriate association array or table.

Simulator function flow

The final modification to add runtime extensibility to the simulator is to allow new features to modify its internal function flow. In particular, new features need to be able to:

- replace existing functions with new functions;
- call a new function before or after an existing function is called;
- modify the inputs or outputs of an existing function; and
- detect when an existing function is called.

All of the requirements can be satisfied by adding a level of indirection to function calls that a runtime extension can modify.

The modified simulator includes a function pointer table. It references any function that a runtime extension might need access to. The original functions exist in the new simulator only as stub functions that look up and call the appropriate entry in the function pointer table; the table begins with pointers to new functions that contain the body of the original functions they replace. If no runtime code modifies any function pointers, the simulator incurs a penalty of a table lookup and additional function call every time one of the original functions is called. Since the original functions tend to perform a significant amount of work, the overhead is small; additionally, the entire modification can be removed at compile time with the use of "ifdef" statements. When the modification is defined out, the stub functions are removed and the simulator does not pay any overhead costs.

Additional modifications

The final "modification" to the simulator consists of simply exposing several internal functions and objects. The modified simulator includes newly-created header files that include declarations for internal functions available to runtime extensions. The simulator also adds new functions that allow extensions to create internal data objects that the simulator uses, such as an object representing cache requests.

5.1.6 Integrating Runtime Extensions

The process of adding runtime extensions to the modified Mambo is relatively simple. After compiling the extension as a shared library, the developer adds a reference to the code in the simulator startup files. In the initialization Tcl script, a runtime extension initialization command loads the extension library using a dynamic library open (`dlopen()`) command. After opening the library, the command next attempts to run a library initialization function (`Library_Init()`). That function, after executing any load-time extension initialization, returns a pointer to a function that allows the Tcl script to execute later commands.

A general runtime extension might offer various configuration or execution options, and using a function to process Tcl requests enables that to happen. For ViVA, the only command necessary is to load the ViVA features into the simulator.

5.1.7 Support Tools

In addition the simulator itself, I used a number of small tools to analyze ViVA's performance. ViVA includes the capability to use small plug-ins, called emitters, that gather statistics on various simulator activities. I instrumented the ViVA runtime extension with emitter events, and wrote an emitter to collect statistics on the vector extensions.

The emitter produces a file that shows the type of event, the instruction number and address that caused the event, and the cycle count of the event. Post-processing of the event log with a Perl script produces a more readable result that shows the progression of individual instructions through the memory hierarchy. The event log is useful for debugging operations and for determining the causes of poor performance in executed code.

5.2 Simulation Methodology

The process of running experiments on the simulator is, in some ways, complex: the basic act of getting a full operating system to boot up can be difficult, and making sure that the simulator can efficiently execute many runs in an accurate, recordable manner requires careful planning. This section describes the process of setting up and executing experiments.

5.2.1 Assembly Programming

As mentioned in Section 3.2.3, I used assembly language to write the code for performing ViVA experiments. The main reason for using assembly, as opposed to a higher-level language like C, is to avoid having to create a ViVA-capable compiler. Fortunately, the use of ViVA does not depend on researching completely new compiler technology; instead, it can build on well-established vector compilers.

The assembly coding style is, for the most part, relatively basic. I did not use use special optimization techniques for any of the ViVA experiments: generally, the programs are simply a straightforward coding of the application, using the techniques described in Section 4.1.2.

5.2.2 Simulator Interaction

A single ViVA simulator run begins by starting the simulator in the faster, less accurate mode. Booting the operating system in the fast mode takes less than half of a minute; booting in accurate mode takes approximately 30–35 minutes. A simulation will usually execute 2–200 times faster in functional mode than in cycle-accurate mode, depending on sort of code it is executing.

The actual processing speed of a particular simulator run depends on the configuration as well as the activity occurring in the simulated system. Of course, executing at higher accuracy requires more processing time. Certain types of events also demand extra simulator processing. For example, executing floating-point instructions is slow: the simulator performs the computations in software, rather than relying on the underlying processor, because different hardware floating-point implementations can exhibit variations that the simulator does not want to reflect. The rate of execution also depends on the cycles per instruction (CPI) of the simulated hardware. During periods where simulated instructions incur many simulated pipeline stalls, the number of simulated cycles per instruction may be very high: the simulator is accounting for many cycles, but since they are stall cycles little simulator work has to occur. A different simulation may proceed at the same rate in simulated instructions per second, but if those instructions do not experience many stalls the simulated cycles per second will be much lower. Table 5.2 shows typical execution rates for various configurations and actions.

Since booting in accurate mode is so slow, I started all simulations in functional mode. The simulator would boot up and read inputs from a secondary script, which would begin the actual benchmark applications. For microbenchmarks and small applications that did not depend on much setup, I switched the simulator to accurate mode in the secondary

Action	Kilo-instructions per second			Kilo-cycles per second		
	Cycle-accurate	Functional	Functional speedup	Cycle-accurate	Functional	Functional speedup
Idle	170	450	2.6	20,000	56,000	2.8
Booting	100–500	10,000–20,000	40–100	100–2,000	10,000-20,000	10–100
Heavy FP	300–400	600–700	1.8–2	200–250	400–500	2
Heavy Int	300–800	20,000–30,000	38–67	300–1,000	10,000–50,000	33–50

Table 5.2: Approximate ViVA simulator execution rates for different configurations and activities.

script. Even though the simulator ran in functional mode until just before the benchmark application, there was still enough simulator activity to populate caches and warm up other architectural features: the shell would interpret the command line, the operating system would begin a new process and read the application from disk into memory, and so on.

Some applications required a lot of setup that was not interesting from a simulation perspective. For example, the striad benchmarks initialize large blocks of memory. In these applications, the simulator begins execution of the benchmark in functional mode. After initialization is complete, the program executes a specific invalid PowerPC instruction that the simulator recognizes as a request to switch to accurate mode. All of the benchmarks that begin in functional mode execute some code after switching to accurate mode, but before executing the core of the benchmark, in order to warm up caches.

The simulator is able to track statistics for critical sections of an experiment. Similar to the invalid instructions that put the simulator into accurate mode, programs can execute an invalid instruction that tells the simulator to reset the statistics or to dump their current values. The statistics show details at all levels of the simulator, including processor pipeline information such as instruction counts and mixes, cache details including histograms of access times, and memory use.

5.2.3 Simulator Test Bench

Evaluating ViVA required a large number of experiments for each application. Some of the microbenchmarks used command-line arguments to vary benchmark behavior, such as the latency calibration benchmarks, described below in Section 5.3.3, which used arrays of different sizes to measure performance of each cache. Most of the benchmarks required many runs to test ViVA options, such as physical vector length. In order to keep track of the experiments and organize the results, I created a test bench.

The simulator test bench consists of a set of calibrated configuration files, Perl scripts to automate multiple runs and store results, and blank execution scripts. A new application setup includes the application itself, the test bench files, and a dedicated virtual disk. The Perl script varies parameters as necessary, creates a separate execution script for each run, and launches the simulation. After the experiment completes, the Perl script moves the results to a separate directory and processes log files using the helper applications described in Section 5.1.7. The test bench can execute a benchmark multiple times, for greater accuracy of results.

Finally, after all of the experiments that vary a single parameter or set of parameters completes, the test bench executes another helper application that produces a higher-level summary log file by analyzing the individual summary files from each run.

5.3 Simulator Calibration

Before running useful experiments, the simulator must be able to produce results that accurately reflect the performance of a real system. After setting all of the configuration options to match the real hardware, I performed an extensive process of calibration to verify that the performance of applications run on the simulator matches the performance of the applications when run on real hardware.

5.3.1 Methodology

The overall methodology for calibrating the simulator is to write targeted applications that focus on a single feature of the system. For most components, I used two versions of calibration microbenchmarks: one that focused on testing the latency of the feature, and one that focused on maximum bandwidth.

In general, I tried to run the benchmarks so that they would depend only on features that had previously been calibrated. The timing of the first microbenchmarks depends on very few features; later programs move "outwards" from the processor, relying on proper timing performance from more and more of the memory hierarchy.

5.3.2 Arithmetic Calibration

The first microbenchmark targets arithmetic calculations. The core of the application is a tight loop that includes no memory references, and so its performance does not depend on the memory hierarchy.

The benchmark comes in two flavors: one that focuses only on integer calculations, and one that performs floating-point operations. Each version includes a small inner loop. The latency measurement version includes multiple statements that it forces to execute serially by making the inputs of each statement depend on the output of the previous statement.

Figure 5.1 shows the results of the add calibration benchmark. The match between the simulator and hardware is excellent. Three of the experiments are within 0.5%, with the integer independent experiment being within approximately 10%. Small differences come from two main sources: the first is that the simulator and hardware are both running full operating systems, and so other processes can add noise to the results. The real hardware includes a network connection and activity that the simulator does not represent. Furthermore, it was not possible to reboot the real hardware for each run, so experiments executed on the physical system do not have a completely fresh environment. The limitations of simulation cause another source of noise: every experiment takes a significant amount of time, so the number of times a simulation run could be repeated is limited. I ran the calibration experiments and all other benchmarks multiple times on the simulator to reduce noise, but yet more repetitions would probably have increased accuracy further.

The reason for the larger difference in the integer independent benchmark is that the processor uses integer ALUs for many operations, and the independent operations can take advantage of any integer ALU as soon as it becomes available. Any small amount of noise that affects the integer ALUs — that is, most of the noise — is multiplied by the integer independent operation benchmark. Fortunately, real applications include memory accesses and additional operations that avoid the situation set up by that benchmark.

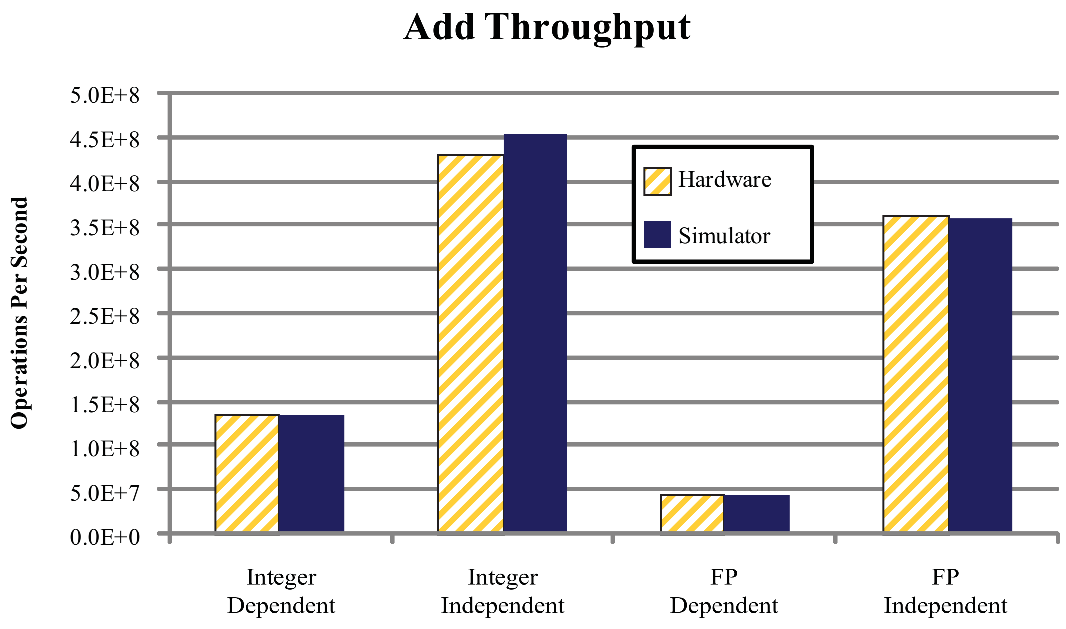


Figure 5.1: Throughput of add operations. Figure includes integer and floating-point operations. Dependent operations must execute serially, reflecting latency; independent operations can execute in parallel, reflecting maximum throughput.

Figure 5.2 shows the results of the multiplication calibration benchmark. Because integer multiplication operations are not pipelined in the PowerPC 970FX [Cor05], the performance of those operations is much lower than on the corresponding add calibration runs. The calibration results are quite similar to the add benchmark: most of the experiments are within 0.5%, and the integer independent benchmark is within approximately 5%. Once

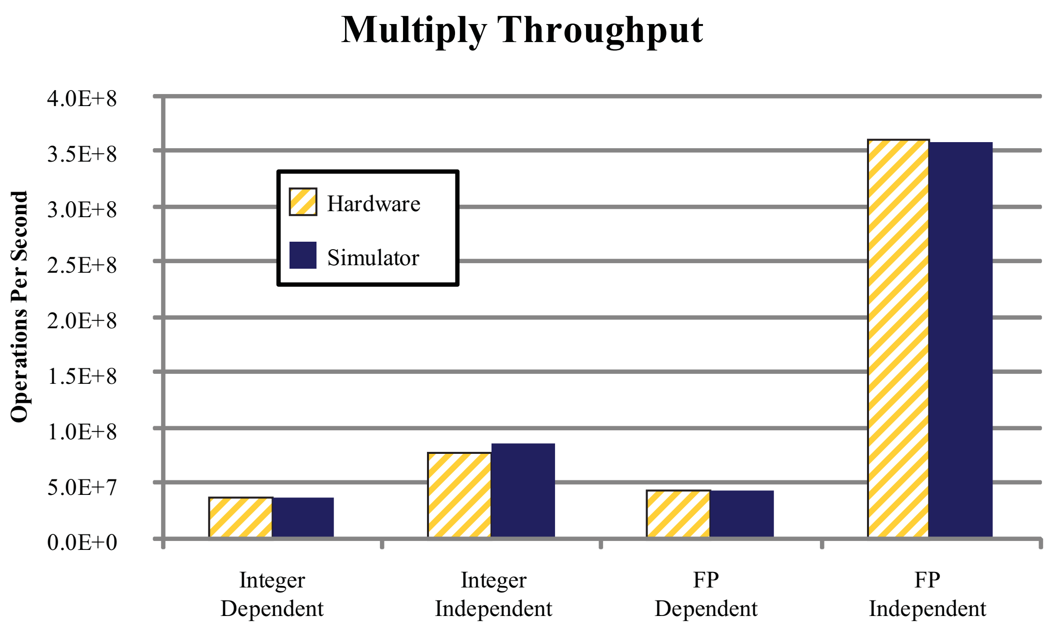


Figure 5.2: Throughput of multiply operations. Figure includes integer and floating-point operations. Dependent operations must execute serially, reflecting latency; independent operations can execute in parallel, reflecting maximum throughput.

again, the benchmark includes the artificially-constructed case of large numbers of independent integer operations and virtually no other operations, so small amounts of noise results in a slightly larger difference in speed.

In general, the arithmetic calibration results show excellent correlation between processing speed on the simulator and real hardware.

5.3.3 Memory Hierarchy Latency Calibration

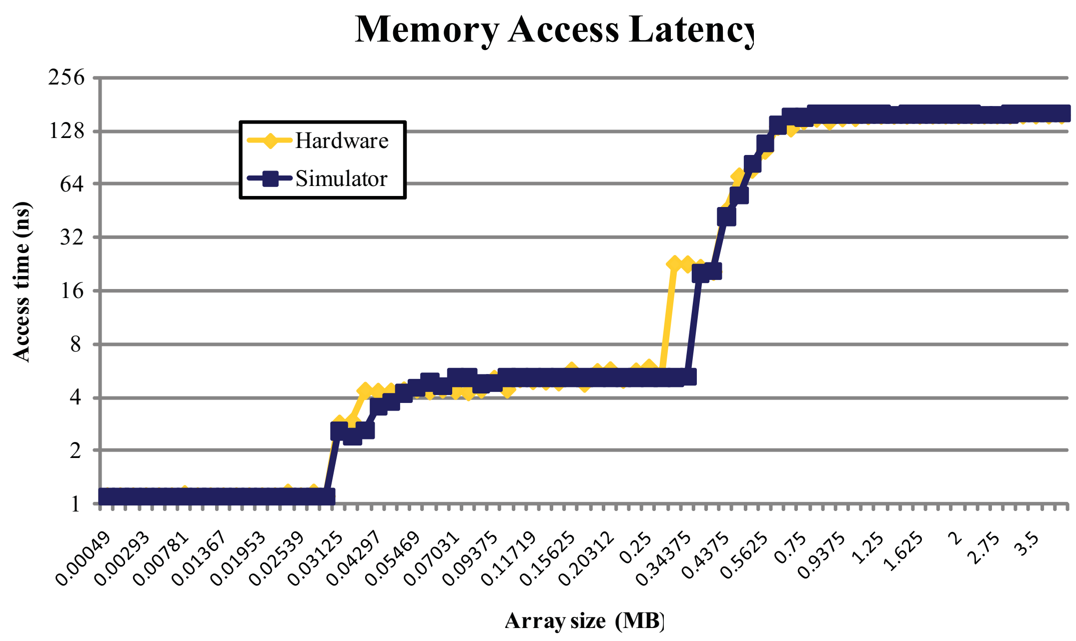
The next step in calibration, after verifying arithmetic throughput, is to calibrate the memory hierarchy. Since memory elements pass through multiple levels of the hierarchy, starting in DRAM and moving through L2 cache, L1 cache, and finally to the processor core, the benchmarks begin calibration at the level nearest the processor core — the L1 cache — and proceed outwards.

For each level of the memory hierarchy, the simulator has to give accurate performance for both latency and bandwidth. Bandwidth depends on the number of requests that the memory system can process simultaneously, as well as the how long they take to complete. Latency depends only on how long each request takes, and therefore it is a better choice to be calibrated first.

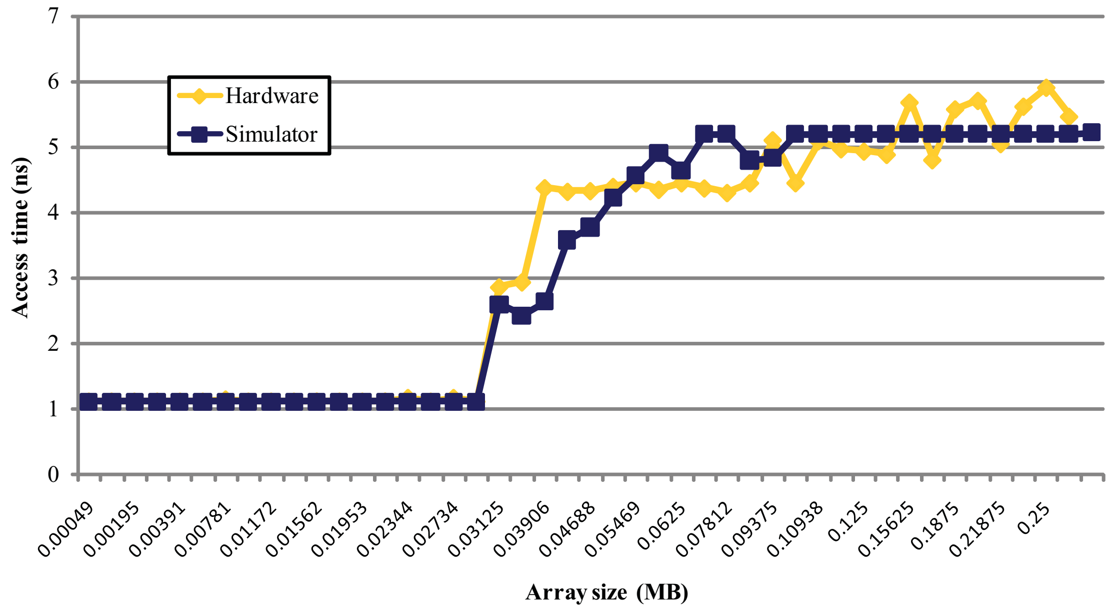
The latency measurements are based on the lat_mem_rd benchmark from the lmbench benchmark suite [MS96]. The full suite takes a long time to execute and would be prohibitive to run to completion on the simulator. Moreover, the suite includes microbenchmarks for system features that do not affect ViVA, such as network access. lat_mem_rd focuses solely on memory latency, and is a better match for performing memory system latency measurements.

lat_mem_rd works by creating an array consisting of linked memory references: each element is a pointer that leads to another element. It randomizes element order, to minimize prefetching effects. The benchmark follows the chain for many elements, determining the average access time for each element. After the program obtains a large sample, it increases the array size, and begins again. An array that is smaller than the L1 cache will eventually reside entirely within the cache, and each read will reflect the L1 cache access time. If the benchmark uses a larger array that does not fit within the L1 cache, it will see some accesses that reflect the L2 cache access time. As the array size increases, most of the accesses will miss in the L1 cache, and the average access will approximate the L2 cache access time. Eventually, the benchmark will move to an array that does not fire entirely within the L2 cache, and some accesses will go all the way to DRAM before being satisfied. The resulting graph of average memory access time for various array sizes shows plateaus for each level of the memory hierarchy; the height of the plateau represents the access time for that level, and the transition from one plateau to the next shows the size of each cache.

Figure 5.3(a) shows the results of the latency benchmark. The agreement between the simulator and real hardware is excellent at all levels of the memory hierarchy: L1 cache, L2 cache, and DRAM. Figure 5.3(b) shows details of the results for experiments near the



(a)



(b)

Figure 5.3: Latency of memory accesses at various levels of the memory hierarchy. The lowest plateau represents L1 cache accesses, the middle plateau represents accesses to the L2 cache, and the slowest accesses are to DRAM. (a) shows the full results; (b) shows detail in the transition from L1 to L2 accesses. Note that the y axis in (a) is logarithmic to show more detail, and does not begin at 0.

transition between the L1 and L2 cache. The hardware does show slightly more noise, likely because of other activity in the system.

There is another small difference between the two curves that is visible in the full results: the transitions between the L2 cache and DRAM plateau do not exactly match up. The hardware line appears to have a slightly different slope. The precise behavior when an array is near the size of a cache depends on a number of features of the system, including cache behavior and system activity. If the test array is the same size as a cache, a small amount of extraneous activity will evict elements from the cache, forcing the processor to refetch them on their next access. Since the difference in access time between the L2 cache and the DRAM is large — over 100 ns — a small number of misses can create a significant increase in average access time. The cache replacement policy also has a large effect on accesses when the test array size is near the cache size. An illustrative example is an array that is slightly larger the cache size, and a program that repeatedly reads array elements in a linear fashion, one after the next. In a cache that uses a true least recently used, or LRU, replacement policy, every access will eventually be a miss: once the program fills the cache and accesses a new line, the cache has to evict a resident line to make new space for it. Imagine that the accesses start at the beginning of the array and have proceeded to almost the end. The cache will evict a line from the beginning of the array, since those early lines were the least recently used; unfortunately, because the accesses will wrap around the array back to the beginning, those are also lines that the program will soon need. In a true LRU cache, every access will eventually be a miss. If the cache uses an approximate LRU, or pseudo-LRU, policy, the cache may not evict a line that was truly least recently used, and that line will be available when the accesses wrap around — resulting in a cache hit.

Fortunately, the results of the calibration benchmark show that any differences between the simulator and the hardware are small. In practice, the effect of those differences is small, and most likely unnoticeable for real applications.

5.3.4 Memory Hierarchy Bandwidth Calibration

The final step in calibration is to verify that the memory hierarchy accurately reflects delivered bandwidth. Once again, the tests begin by measuring the bandwidth at the nearest level, the L1 cache, and proceed outwards.

The latency measurement benchmark is a simple application that creates an array designed to fit within a specific level of the memory hierarchy. The benchmark then repeatedly accesses the entire array, one element after another, and measure how much bandwidth the system can deliver.

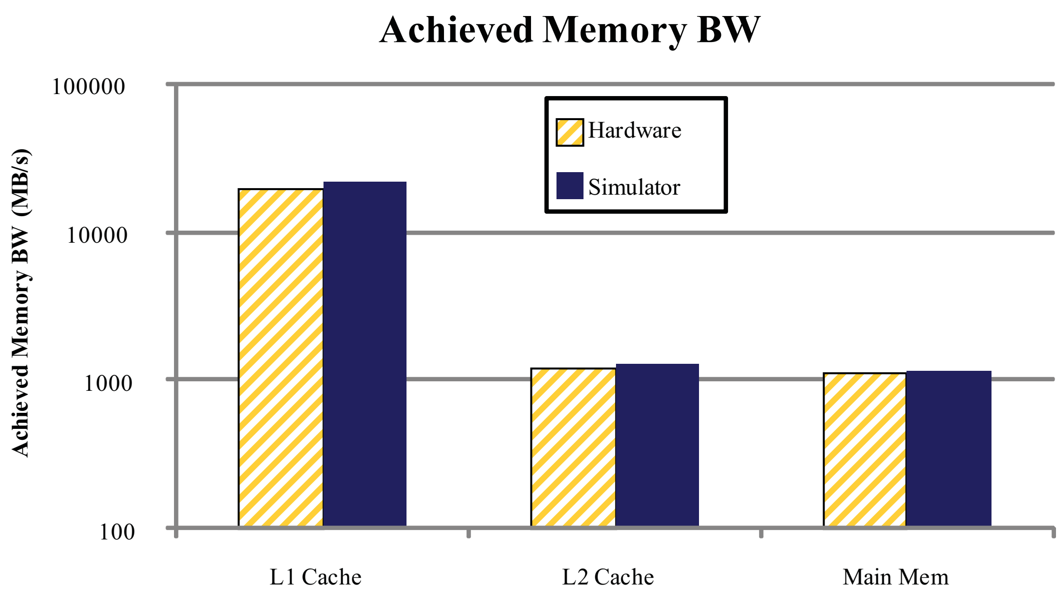


Figure 5.4: Delivered bandwidth for various levels of the memory hierarchy. Note that the y axis is logarithmic to show more detail, and does not begin at 0.

Figure 5.4 shows the results of the bandwidth benchmark. Once again, the results are quite close, with the simulator being within approximately 10% for all levels of the memory hierarchy. The largest difference is in the L1 cache, which is the most sensitive to noise caused by other activity in the system.

5.4 Summary

This chapter described the design of the ViVA simulation environment. The simulator is based on Mambo, a cycle-accurate full-system PowerPC simulator from IBM. I modified the simulator so that its capabilities can be extended at runtime, and then wrote an extension that models ViVA operation and performance.

I set the simulator to match the configuration of a modern microprocessor, the IBM PowerPC 970FX. Calibration experiments show that the simulator closely matches the performance of real hardware for arithmetic and memory operations.

The next chapter presents the performance results of ViVA simulations.

Chapter 6

Performance Results and Discussion of Implications

ViVA offers the potential for microprocessors to achieve better performance than scalar processors, while avoiding power-hungry hardware prefetchers. By adding a buffer between the processor core and second-level cache, ViVA is able to use new vector-style memory transfer instructions — and get many of the benefits of a full vector implementation — without paying the cost of adding a large vector datapath. In order to determine the value of adding ViVA to a microprocessor, we have to quantify the performance benefit that ViVA can deliver.

This chapter evaluates the performance of ViVA for a variety of applications. It begins by examining a short series of microbenchmarks, which measure the performance on different memory access patterns one at a time. This chapter also presents the results of corner turn and sparse matrix-vector multiplication, which many large scientific applications commonly use.

6.1 ViVA Results: Microbenchmarks

The first ViVA simulations performed are the most basic microbenchmarks possible: small applications that test one particular feature of the system.

6.1.1 Unit Stride

The first microbenchmark simply measures maximum unit-stride bandwidth. The benchmark compares a straightforward scalar implementation, a tuned scalar implementation, and a ViVA implementation simulated with various physical register lengths. Figure 6.1 shows the results.

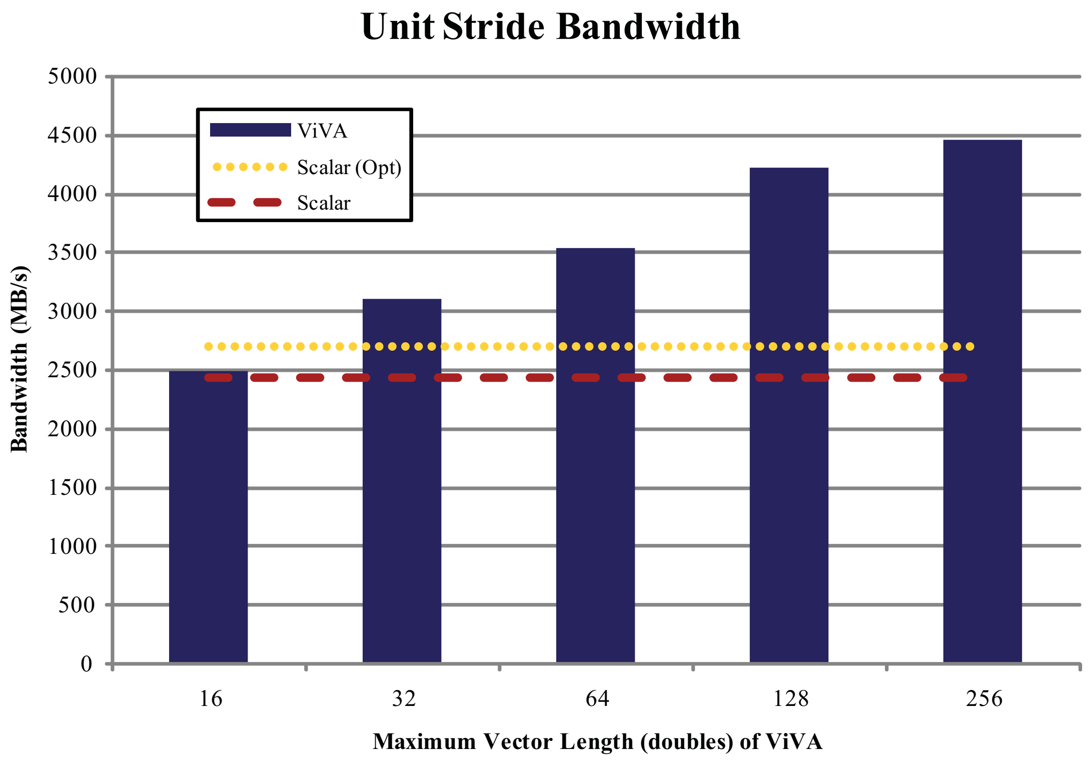


Figure 6.1: Unit stride bandwidth for straightforward scalar implementation, tuned scalar implementation, and ViVA implementation with various physical register lengths. Lengths shown in number of doubles.

The straightforward scalar implementation delivers approximately the same amount of bandwidth as ViVA with a hardware vector register length (MVL) of 16 doubles, or around

2.5 GB/s. After performing simple autotuning (automated searching over tuning parameters) to determine optimal prefetching organization and compiling options, the tuned scalar implementation is able to deliver slightly more than 10% better performance than ViVA with an MVL of 16. At longer physical register lengths, ViVA's delivered bandwidth improves significantly, providing greater than 80% more bandwidth than the straightforward implementation at an MVL of 256 doubles.

For an MVL of 16, ViVA's vectors do not contain enough concurrency to overcome the scalar implementation's advantage of being able to make use of the hardware prefetcher. For longer MVLs, ViVA is able to deliver greater bandwidth than a tuned scalar implementation for unit-stride streams, even though the scalar hardware is optimized particularly for that memory access pattern. There are a number of inefficiencies and tradeoffs in the scalar hardware prefetcher that allow ViVA to get better performance.

One difficulty that the hardware prefetcher faces is that it only deals with physical address streams, and therefore cannot prefetch past a virtual page boundary [Cor05]. The single-page hardware prefetch limit can be a significant impediment to achieving good performance [DSMR07]. Once a memory stream crosses a 4 KB page boundary, it cannot prefetch new entries until it re-identifies the stream. ViVA, in comparison, does not have that limitation. Since a ViVA load identifies a series of elements that the program will use, the hardware does not have to discover the stream on its own.

The hardware prefetcher has another challenge to achieving good performance: it performs a slow prefetch ramp-up [TDJSF$^+$02]. Since the hardware tries to predict future accesses based on past requests, it may incorrectly identify memory streams. False streams consume memory bandwidth (and power) needlessly. Worse than that, false streams request useless data, which evicts data from the cache that may have to be refetched: not only are false streams nonproductive, they can actually reduce performance below the level it would be if there were no hardware prefetching. False streams can be especially damaging because prefetch streams tend to request the most data when they are new, before the prefetcher can detect that it has identified a stream incorrectly.

The ideal state in prefetching is to request elements that are a certain distance ahead of the current demand request, so that the prefetched data is available before it is needed. Since no data is available when a prefetch stream begins, the hardware has to transition to the steady state by requesting all of the elements between the most recent demand request and the element that is the ideal distance ahead. Hardware prefetchers are the result of an engineering tradeoff between two competing goals: requesting transition items quickly to speed up moving to the ideal steady state, and avoiding loading many useless elements from false streams.

Slow prefetch ramp-up allows the prefetcher to achieve a balance between slow but safe behavior and fast but risky strategies: when the prefetcher identifies a stream, it does not request all of the transition elements. Instead, it requests a small subset of elements. If the program continues to request elements from the stream, the hardware prefetcher fetches more and more of the transition elements, until it has reached the ideal steady state. If the identified stream is a false stream, and the program does not continue to request elements that match that pattern, the prefetcher will only have requested a few useless elements; if the stream is true, the prefetcher will eventually reach the ideal prefetch distance.

Slow ramp-up is a good balance for the case where the system has to identify memory streams on its own. Fortunately, ViVA does not have to deal with that challenge: vector memory operations naturally explicitly identify a large number of elements that the program will use. In the case of the unit-stride microbenchmark, the hardware prefetcher will probably not find any false streams — all of the references belong to unit-stride streams that are easy for the hardware to detect. Even so, ViVA will be able to achieve better performance because the hardware prefetcher can only follow a stream within a single hardware page. Once the stream crosses a page, the prefetcher will have to re-identify it, and go through the slow ramp-up process again.

Another challenge that the prefetching hardware faces is determining the appropriate prefetch distance: in the steady state for a unit-stride stream, should the hardware prefetcher request memory that is eight lines ahead of the current demand request? Is that number too high, or too low? If the prefetcher uses a value that is too low, the prefetch will not cover the full latency to memory, and performance will suffer. If the prefetcher uses a value greater than the ideal, it will request data too early — possibly leading to a case where data is prefetched and evicted by a different request before it is used, and definitely increasing the time and complexity of the ramp-up period. Microprocessor designers fix the prefetch distance based on analysis of memory request latency and predicted program behavior, but the final result cannot account for the variability in different systems and programs. Every version of the IBM PowerPC 970FX, the microprocessor I model, uses a single prefetch strategy, regardless of the type or speed of system DRAM or the processor clock rate — which varies between 1.0 GHz and 2.7 GHz. No single prefetch distance can be ideal for all of those configurations, resulting in performance inefficiencies in some cases.

ViVA is able to get better performance on the unit-stride microbenchmark for another important reason: cache fills. As described in Section 3.2.2, a cache fill occurs in write-allocate caches when the program writes to an address that is not currently in the cache. Since the processor does not allow only part of a cache line to contain valid data, it must first execute a load for the cache line to retrieve all of the elements besides the one being written. If the processor stores an entire cache line's worth of data at once, it does not have to perform a cache fill: the entire cache line will contain valid data. A scalar processor can avoid cache fills by merging individual scalar writes and noticing the case where a store covers a full cache line, but performing that analysis can be costly. Even if the processor does attempt to avoid cache fills, it will only be successful if the program performs a full cache line's worth of stores and the processor manages to merge them all before they reach the cache. ViVA stores long vector registers, so it is both easy and common for the processor to detect the case where it can avoid a fill.

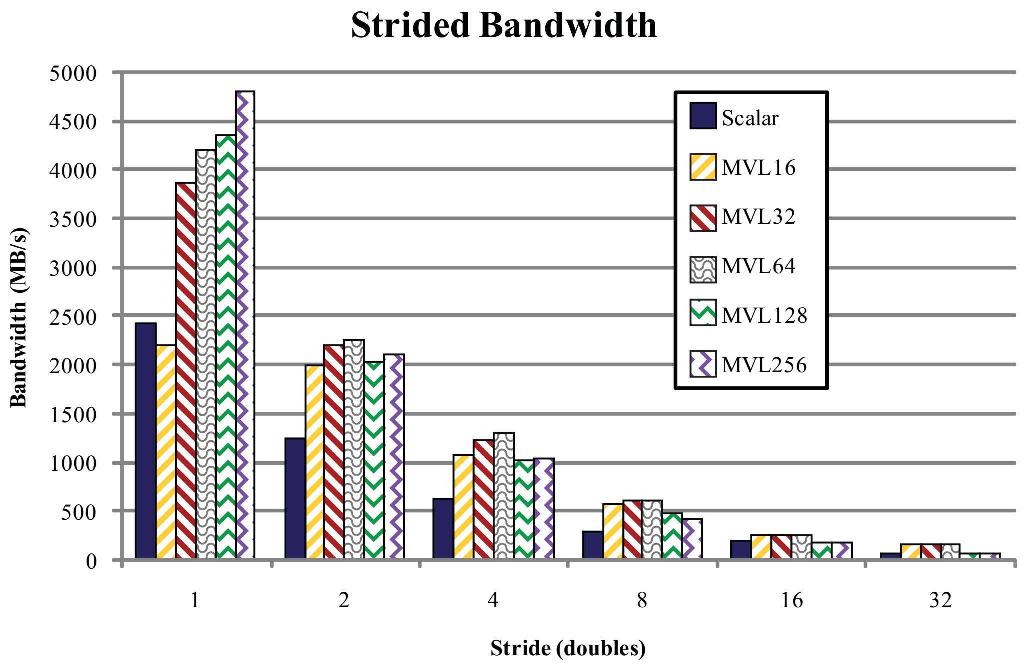
6.1.2 Strided

The strided benchmark is similar to the unit-stride benchmark, except that it accesses elements that are evenly spaced apart in memory. Figure 6.2 shows the results with both linear and logarithmic scales.

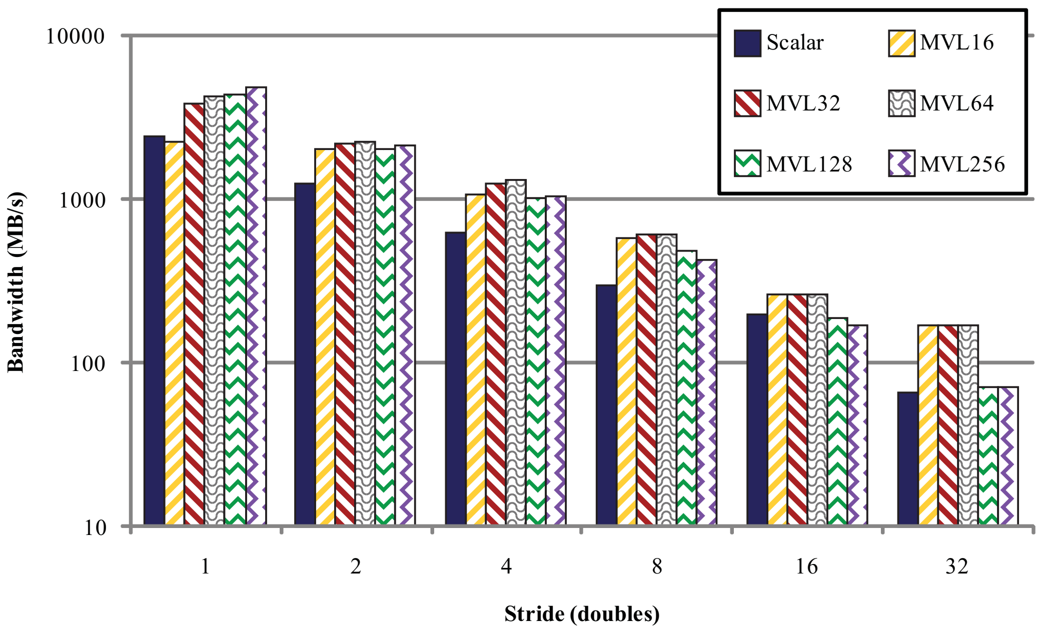
The strided results are slightly more complicated than the unit stride results, because the experiment tests performance for various strides. The benchmark tests power-of-two strides from 1 to 32. The stride 1 version of the code has the same memory access pattern as the unit stride microbenchmark. The performance on the stride 1 version of the strided benchmark is not identical to the unit stride version because the code itself is slightly different: it uses a strided instruction, as opposed to unit-stride, because it has to deal with the ability to handle arbitrary strides. Additionally, the ViVA version of the code handles arbitrary strides, but only performs full MVL transfers. Software prefetching gave virtually no improvement to the scalar code.

For a stride of 1, the scalar implementation slightly outperforms the ViVA implementation with an MVL of 16. For longer MVLs, ViVA delivers greater bandwidth.

The feature that first stands out in the performance graph is the steep drop-off in performance for longer strides. The problem arises from the way the cache retrieves data from memory: the smallest unit that the cache transfers to or from DRAM is a full cache line of 16 doubles. With unit-stride streams, full cache line transfers cause no problems. The program ends up using every element, the cache can take advantage of spatial locality, and multiple-element transfers amortize cache, bus transfer, and DRAM access overhead over more values. As soon as the program starts striding through memory, transfers of full cache lines end up wasting bandwidth. In the case of the stride 4 code, for example, the program only uses one out of every four elements transferred to the cache. As stride increases, full cache line transfers waste more and more bandwidth, until stride 16, when only a single element per cache line is used. At stride 32, the program also uses a single element per cache line, but the cache only accesses every other line in memory.



(a)



(b)

Figure 6.2: Strided bandwidth for scalar implementation and ViVA implementation with various physical register lengths (MVLs). Physical register lengths shown in number of doubles. Note that the y axis in (a) is linear, while the y axis in (b) is logarithmic to show more detail, and does not begin at 0.

Bandwidth continues to drop from stride 16 to stride 32, even through both of those experiments make use of the same fraction of transferred elements. One challenge that both the scalar and ViVA versions face is that the stride 32 code spans a greater address range. As a program accesses a greater address range, it accesses more memory pages, and requires more costly memory address translations. The statistics show that the stride 32 scalar version of the program has a 42% greater number of data TLB lookups per byte (1.0 versus 0.7) and a data TLB miss rate twice as high (1.143% versus 0.577%) as the stride 16 version.

The scalar version of the benchmark faces another problem as it moves to strides greater than 16. As long as the program's memory access stream touches consecutive cache lines, even if the program only uses one element in each of those lines, the hardware prefetcher will be able to detect the stream and prefetch future elements. As soon as the memory request stream does not access consecutive cache lines, the hardware prefetcher stops providing a benefit to delivered bandwidth. Once again, the simulator statistics show details of the behavior: the stride 16 version of the benchmark issues 499,644 hardware prefetch requests, while the stride 32 version issues only 19. It is possible for a hardware prefetcher to detect streams with greater strides, but hardware prefetchers of more complexity take longer to design and verify, tend to consume greater power, and may detect more false streams.

Figure 6.3 shows the performance of the ViVA version of the benchmarks, compared to scalar code. The results are illustrative, particularly for longer MVLs. For the stride 1 experiment, longer MVLs give better performance. As stride increases, medium-length MVLs give the best performance. At strides of 16 and 32, the experiments run with the longest MVLs give performance that is approximately equal to the scalar code. The behavior is a result of the greater memory address span mentioned above, but amplified because of ViVA's long register lengths. If a ViVA instruction is interrupted, the instruction must be restarted from the beginning. Longer ViVA instructions can amortize costs over more

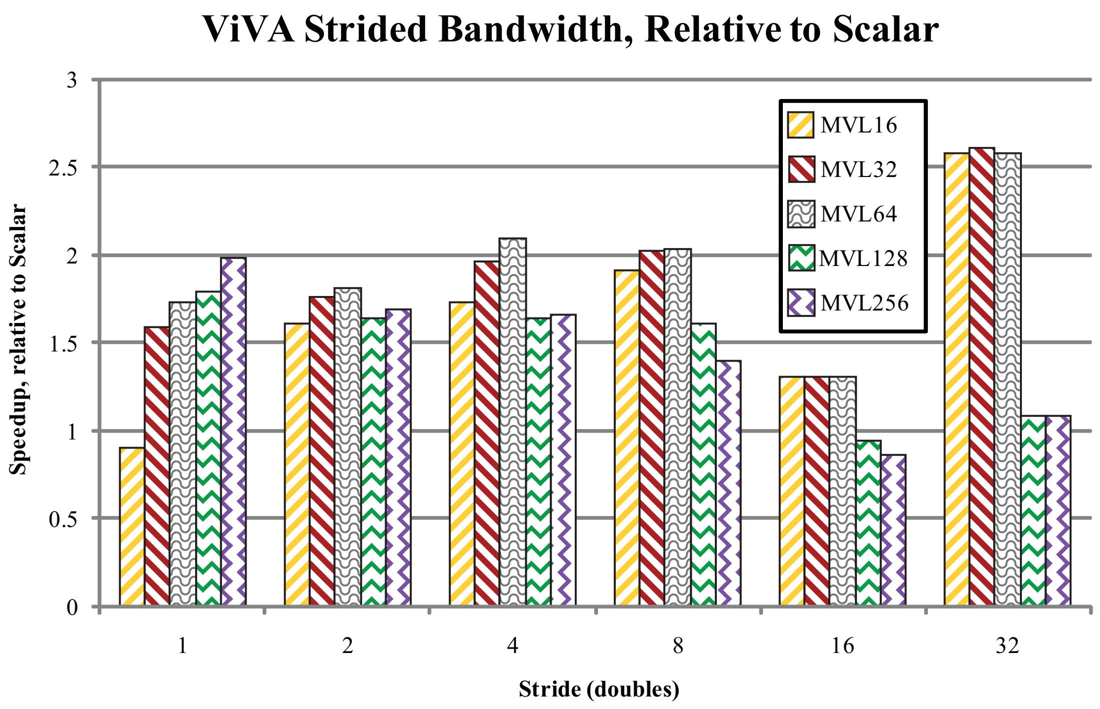


Figure 6.3: Strided bandwidth of ViVA implementations with various physical register lengths (MVLs), relative to scalar bandwidth. Physical register lengths shown in number of doubles.

elements, but those returns diminish as the physical register lengths get long. As ViVA instructions grow to long lengths, they have a greater chance of being interrupted, and have to pay a higher cost for restart once that happens.

As the strided microbenchmark shows, very long physical register lengths can cause reduced performance. Later experiments in the following sections use a length of 64 doubles: long enough to achieve good performance, but not so long that performance begins to drop off.

6.1.3 Indexed

The indexed benchmark does not have streams in the way that the unit-stride or strided benchmarks do. Instead, the benchmark first generates a large array of random indices, and then accesses memory at those locations. The benchmark only begins to measure performance after it creates the random indices. Thus, the benchmark does have a unit-stride stream, to retrieve index values, but the rest of the accesses are random. The benchmark operates by iterating over different array sizes, recording the delivered bandwidth for each size.

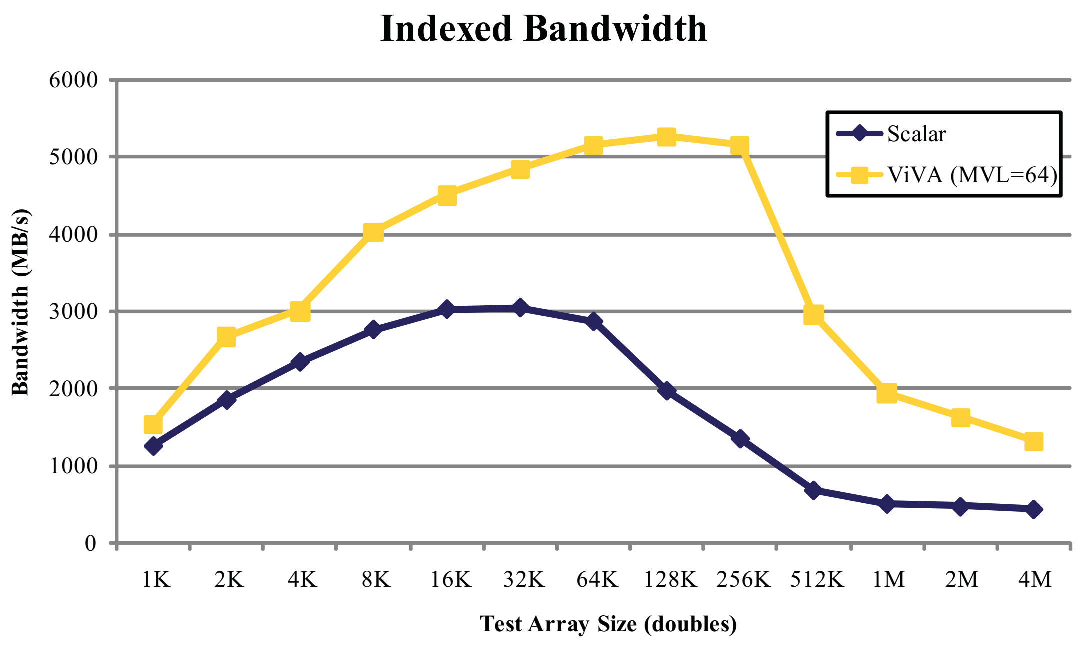


Figure 6.4: Indexed bandwidth of scalar implementation and ViVA implementation for various source array sizes.

Figure 6.4 shows the results. Both the scalar and ViVA implementations show the same general shape: performance increases with the array sizes, until it drops off for the largest sizes. As the array size gets larger, the processor has longer streams and can achieve better bandwidth. Eventually, as total array sizes increase beyond the range that the TLB covers, overall performance drops significantly. As with the strided benchmark, software prefetching gave virtually no improvement over the straightforward scalar implementation.

At the smallest array size, ViVA gives slightly better performance than the scalar implementation. As the array size increases, ViVA's benefit over the scalar performance increases. Additionally, ViVA's drop-off in performance occurs at a larger array size than for the scalar implementation.

```
for(full_array) {
   vec_reg0 = viva_load_unitstide(indices_ptr);
   vec_reg1 = viva_load_indexed(vec_reg0);
   viva_store_unitstride(vec_reg1, dest_ptr);
   indices_ptr += MVL; dest_ptr += MVL;
}
```

Figure 6.5: Pseudocode to perform indexed benchmark.

ViVA is able to outperform the scalar version of the benchmark for a few important reasons. For one, the memory access pattern allows ViVA to skip cache fills: the program performs all stores in a unit-stride fashion with long vectors, virtually guaranteeing that the L2 cache always sees full cache line writes. More importantly, though, the ViVA version of the benchmark never needs to pull data all the way into the core. Figure 6.5 shows the pseudocode for the ViVA implementation. The application simply reads in a block of indices using a unit-stride load, reads the source values using an indexed access, and stores the values to the destination location with another unit-stride store.

The scalar memory hierarchy usually depends on the hardware prefetcher to achieve good bandwidth, but the prefetcher has difficulties with the indexed benchmark program that can reduce overall performance. Hardware prefetchers can only track a limited number of memory streams. If an application uses more streams than the hardware can track, the later streams will not derive any benefit from the prefetcher — in fact, later streams may possibly evict established streams from the prefetcher, resulting in a steady-state stream being replaced by one just beginning its ramp-up phase [PK94]. Microprocessors can attempt to reduce this stream thrashing by adding filters to the prefetcher: the hardware will effectively track more streams than the prefetcher can handle, to ensure that the extra streams do not evict working streams [SSC00]. Nevertheless, once the number of detected streams exceeds the number that the hardware can track, hardware prefetchers will begin to lose performance.

The simulator statistics show stream thrashing occurring for the indexed benchmark. Ideally, the hardware prefetcher would detect the unit-stride streams available in the pro-

Array size (doubles)	Percentage of prefetches that are useless
1k	21.9
2k	32.1
4k	24.3
8k	17.9
16k	35.2
32k	42.9
64k	37.4
128k	33.4
256k	78.1
512k	78.8
1M	76.0
2M	70.7
4M	74.3

Table 6.1: Percentage of prefetches that are useless in the indexed benchmark.

gram, and ignore the memory accesses to random memory locations. Of course, the application has no way to tell the processor which type of access it is performing at any time, and so the processor simply looks for accesses to consecutive cache lines. In the case of the indexed benchmark, two accesses to consecutive cache lines in a short period can cause the prefetcher to detect a false stream. Table 6.1 shows the percentage of total prefetches that ended up delivering no useful data. Smaller array sizes have a significant fraction of useless prefetches, but tend to take advantage of at least two thirds of the prefetches they issue. Larger arrays are able to use useful prefetches a much smaller fraction of the time. By the time the application moves to the experiment with 256K doubles, after the point where performance has begun to drop off, almost 80% of the prefetch requests end up fetching data that is never used.

6.2 ViVA Results: Striad

The stanza triad, or striad, benchmark models a memory access pattern that is common in any application that accesses a subblock in a large array of data. Adaptive mesh refinement, or AMR, applications and stencil codes both exhibit stanza memory access patterns.

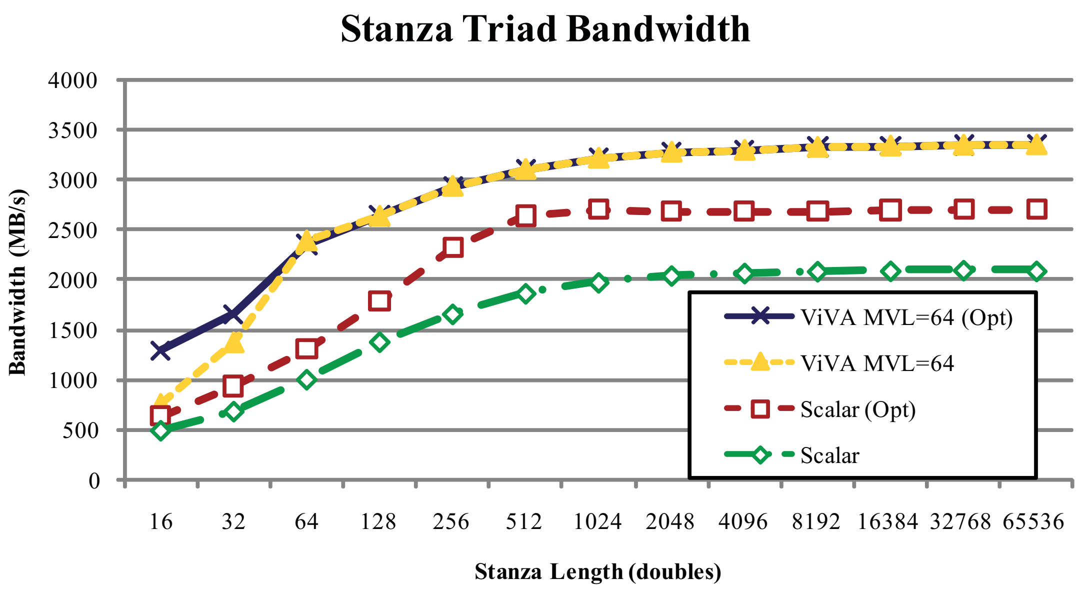


Figure 6.6: Stanza triad (striad) bandwidth for scalar and ViVA implementations.

Figure 6.6 shows the bandwidth achieved for a scalar and ViVA implementation of stanza triad. The benchmark operates by accessing a stanza – a block of consecutive memory locations – then skipping ahead in memory, and repeating the process. The graph shows bandwidth for various stanza sizes. For all implementations, the bandwidth begins low, and goes up with stanza length. One difficulty that short stanzas pose to all code is that memory address translation overhead is amortized over only a few elements. As stanza length increases, the cost of translating the address of a physical memory page is spread over more elements.

Prefetching was not able to improve the scalar bandwidth significantly, most likely because the hardware prefetcher cannot give good performance for the stanza memory access pattern. The shape of the scalar ramp-up mirrors the ramp-up of the hardware prefetcher. For the shortest stanzas, the hardware prefetcher might not find any streams. For slightly longer stanzas, in the range of 64 to 128 doubles, the hardware prefetcher begins to deliver some benefit, but the stream does not continue long enough for the prefetcher to reach its full bandwidth potential. After stanzas of 512 to 1024 doubles, the hardware prefetcher is able to recognize streams and transition to its steady-state behavior of giving the maximum possible benefit.

ViVA does not use the hardware prefetcher, but shares some of the short stanza issues with the scalar implementation. As with the scalar implementation, short stanzas pay a higher per-element cost for memory address translation. The straightforward ViVA implementation starts off at approximately the same bandwidth as the optimized scalar implementation for a stanza of 16 doubles, but ViVA's performance grows at a faster rate. The scalar implementation depends on a hardware prefetcher to get better performance with longer stanzas; ViVA, on the other hand, is able to present more elements to the memory hierarchy at the same time, for greater possible concurrency.

The shortest stanzas present a unique problem to ViVA. ViVA depends on vectors with multiple elements to achieve concurrency in memory accesses. The shortest stanzas end up limiting vector lengths, leading to reduced performance. The normal ViVA implementation uses unit-stride transfers, which are limited to accessing consecutive elements in memory. If a stanza is shorter than the physical register length, a unit-stride memory instruction can only transfer a single stanza, and cannot take advantage of full-vector operations.

It is possible for ViVA to access multiple short stanzas with a single operation, but not by using unit-stride transfers. Instead, ViVA can use indexed instructions to refer to the elements in two or more stanzas. The optimized line in Figure 6.6 shows the results of using indexed transfers for the two smallest stanza sizes, 16 and 32 doubles. The bandwidth of the optimized ViVA code is just over double that of the optimized scalar code at a stanza length of 16 doubles. For stanza lengths of 64 doubles and greater, the code is the same for the two ViVA versions: only very short stanzas see a benefit from using indexed instructions.

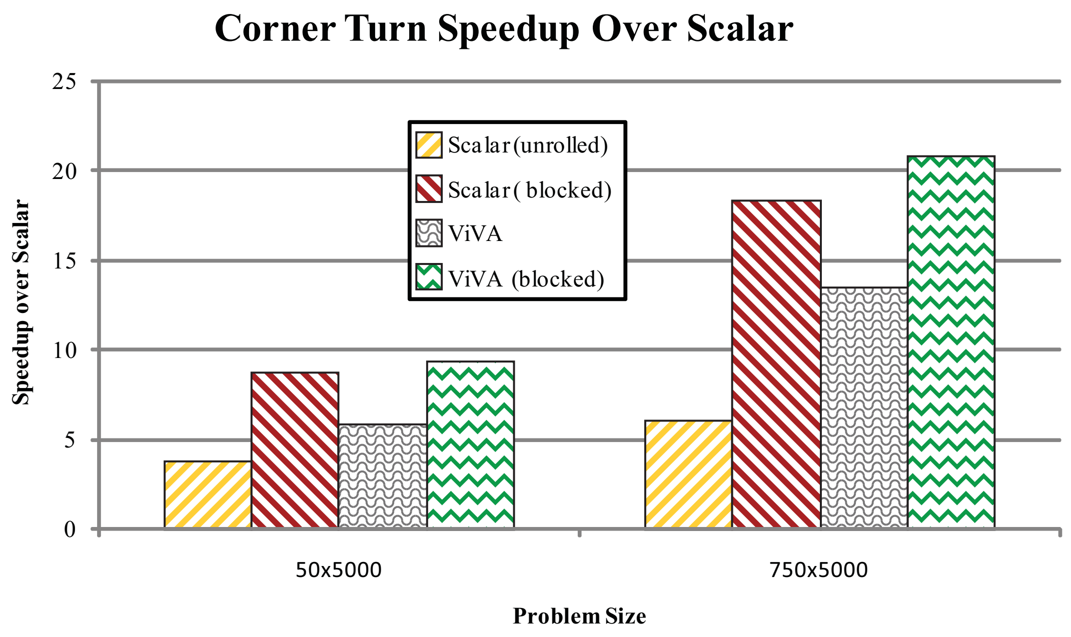


Figure 6.7: Corner turn bandwidth for scalar and ViVA implementations.

6.3 ViVA Results: Corner Turn

Corner turn is a compact kernel that is frequently used in processing multi-dimensional data. A corner turn operation is simply equivalent to transposing a matrix, and is used to reorient data elements so that later processing is more efficient.

The ViVA benchmark is based on the HPEC Challenge benchmark suite [LRW05]. The application runs for two different pre-defined array sizes. Both arrays have 5000 columns; the small array has 50 rows, and the large has 750 rows.

Figure 6.7 shows the results of the benchmark. All of the ViVA implementations and the optimized scalar implementations perform significantly better than the straightforward scalar version. The first scalar optimization, unrolling, can often be performed by compilers. This implementation was tuned, similar to the way an autotuner would operate, to find the optimal level of unrolling. The second scalar optimization, cache blocking, improves performance quite a bit. Cache blocking effectively restructures an application to maximize the benefit of caches. It rearranges the order of computation so that more work is done on a block of data before another block is accessed, even if the resulting order is not as conceptually straightforward. Cache blocking is often implemented by hand or tuned through search, and usually cannot be performed directly by a compiler.

The straightforward ViVA implementation gets better performance than the unrolled scalar code, but does not achieve the level attained by the cache blocked scalar code. Cache blocking allows many more memory accessed to be satisfied in the cache, leading to reduced DRAM accesses and greater performance. Figure 6.8 shows that the cache blocked code performs fewer than half as many DRAM reads as the optimized scalar version. ViVA's implementation does not change the computation order from the straightforward implementation; the resulting code is simpler and could be generated easily by a compiler, but it produces more lengthy DRAM accesses. The optimized ViVA code, that adds cache blocking to the straightforward code, gets the best performance of all. This suggests that ViVA, like many architectural features, is amenable to autotuning. The straightforward ViVA implementation gives good performance, and an autotuned scalar implementation gives good performance, but the best performance results from combining both strategies.

Problem Size	Scalar	Scalar (unrolled)	Scalar (blocked)
50x5000	.998	1.004	1.000
750x5000	1.000	1.000	1.000

Table 6.2: Relative performance of corner turn with larger L1 cache. Lower is better.

ViVA consumes additional chip area over a scalar design. An alternative to using that area for ViVA is to increase cache area. Section 3.2.4 shows that ViVA does not consume a large amount of chip area: an equivalent increase in L2 cache size would be tiny. An increase in L1 cache size, on the other hand, would be more significant. In order to evaluate ViVA's benefit in comparison to more L1 cache, I simulated corner turn using a scalar implementation with a larger L1 cache. A ViVA implementation that includes 32 vector registers, each holding sixty-four 64-bit words, would add 16 KB of storage to the core. For comparison purposes, I simulated with an L1 cache of 64 KB that doubles the original

32 KB cache, and I did not increase access time. I simulated a number of different blocking factors to attempt to find a better size. Table 6.2 shows the results. Notice that the differences in performance are tiny. All are well within 1% of the original performance. Some of the results, counter-intuitively, are worse than the original version. The original L1 cache size is large enough that it is not a limiting factor in corner turn's performance; a larger cache size then changes the precise order of processor activity and slightly changes overall execution time, but does not result in significantly better performance. Because the larger L1 cache does offer the potential to exploit data reuse in a larger block, it is likely that a more complex implementation, perhaps a tuned double-cache blocked version, would be able to achieve some more performance. Creating an autotuner to examine the numerous combinations of doubly-blocked scalar options is complex, and outside the scope of my work.

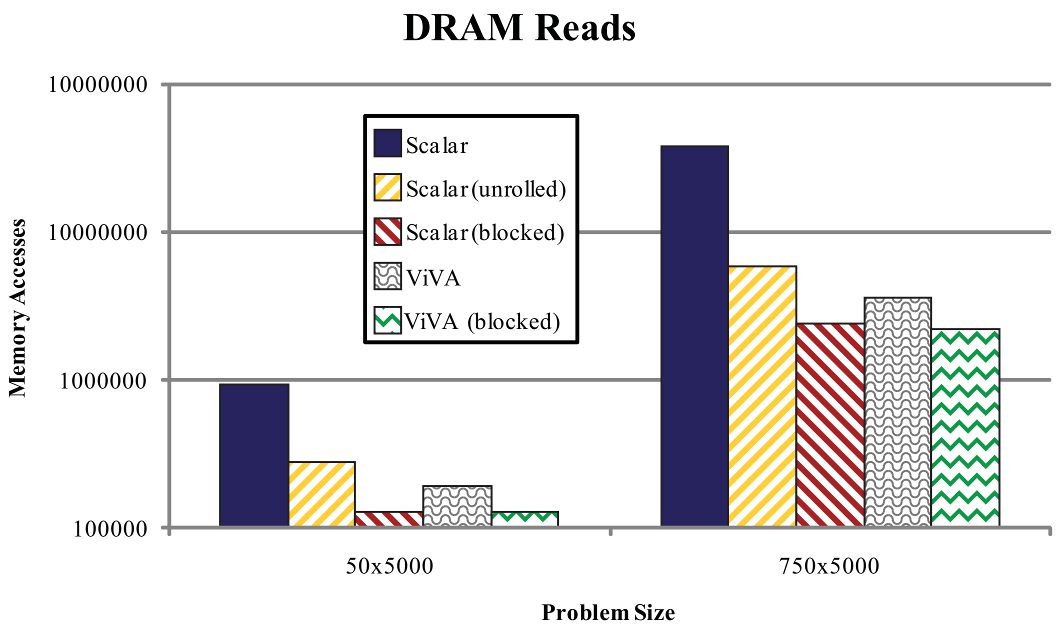


Figure 6.8: Total corner turn DRAM accesses for scalar and ViVA implementations. Note that the y axis is logarithmic to show more detail, and does not begin at 0.

Figure 6.7 shows an interesting result of the corner turn experiments that highlights another potential advantage of ViVA. The figure plots the total number of DRAM accesses performed while executing the corner turn benchmark. DRAM accesses are a significant source of DRAM power consumption, which can be the dominant component of overall system power [DKV+01]. As the graphs show, DRAM accesses with even the straightforward ViVA implementation are much lower than with the straightforward or unrolled scalar implementations. The blocked ViVA implementation has the lowest number of DRAM accesses, less than all of the scalar implementations.

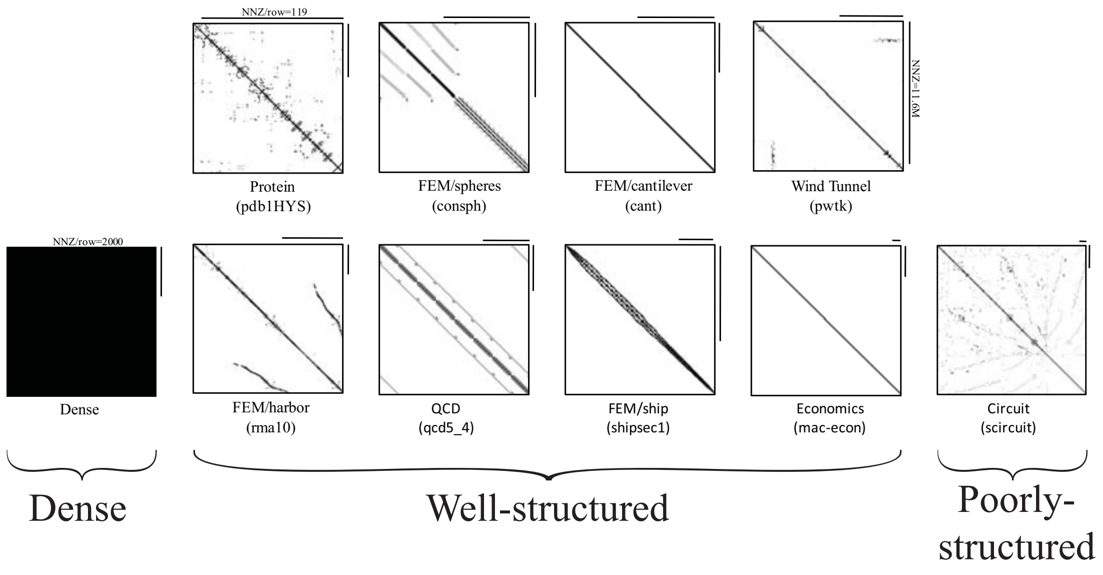
6.4 ViVA Results: Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector multiplication, or SpMV, is commonly used for scientific applications. Many larger matrix operations are based on SpMV. Simulations and other large applications use sparse representations to improve performance, or to allow the matrix to use memory more efficiently.

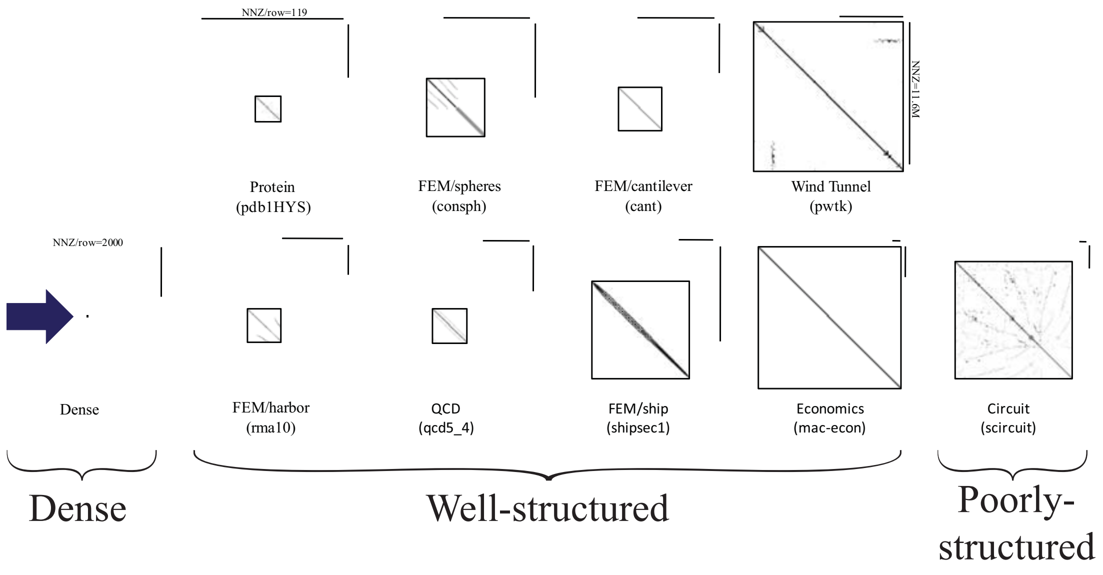
Matrix	Filename	Rows	Cols	NNZ	NNZ/row	Notes
Dense	dense2	2K	2K	4M	2000	Dense matrix in sparse format
Protein	pdb1HYS	36K	36K	4.3M	119	Protein data bank 1HYS
FEM/Spheres	consph	83K	83K	6M	72.2	FEM concentric spheres
FEM/Cantilever	cant	62K	62K	4M	64.5	FEM cantilever
Wind Tunnel	pwtk	218K	218K	11.6M	53.2	Pressurized wind tunnel
FEM/Harbor	rma10	47K	47K	2.4M	50.4	3D CFD of Charleston harbor
QCD	qcd5_4	49K	49K	1.9M	38.9	Quark propagators (QCD/LGT)
FEM/Ship	shipsec1	141K	141K	4M	28.2	Ship section / detail
Economics	mac_econ_fwd500	207K	207K	1.3M	6.1	Macroeconomic model
Circuit	scircuit	171K	171K	1M	5.6	Motorola circuit simulation

Table 6.3: Details of matrices used for SpMV.

The ViVA benchmark tests SpMV with a variety of matrices collected by the BeBOP group [BeB08]. Figure 6.9 shows the matrices, which cover a wide range of density, size, structure, and application. Table 6.3 shows additional details of the structure of the matrices. The matrices can be split into three broad categories, based on their structure. The first is simply a dense matrix, stored in a sparse format. The dense matrix gives an upper bound on the performance of SpMV: generally, as a matrix contains fewer nonzeros per row, the performance decreases. The second category is well-structured matrices. These matrices



(a)



(b)

Figure 6.9: Matrices used for SpMV. Matrices are split into three categories: dense, used to get a baseline comparison value; well structured, where the nonzeros are relatively dense or mostly placed along the diagonal; and poorly structured, where the nonzeros are sparse and irregular. The line above each matrix shows the relative number of nonzeros per row, normalized to 119 for a full bar. The line to the right of each matrix shows the total number of nonzeros, normalized to 6 million for a full bar. (a) shows the matrices depicted at the same size. (b) shows the matrices with proper relative proportions. Table 6.3 gives quantitative details of the matrices.

exhibit two characteristics: they are relatively dense, containing many nonzeros per row, and the nonzeros are located mostly along the main diagonal. The final category includes an extremely poorly-structured matrix, which contains a large number of nonzeros that do not lie along the diagonal.

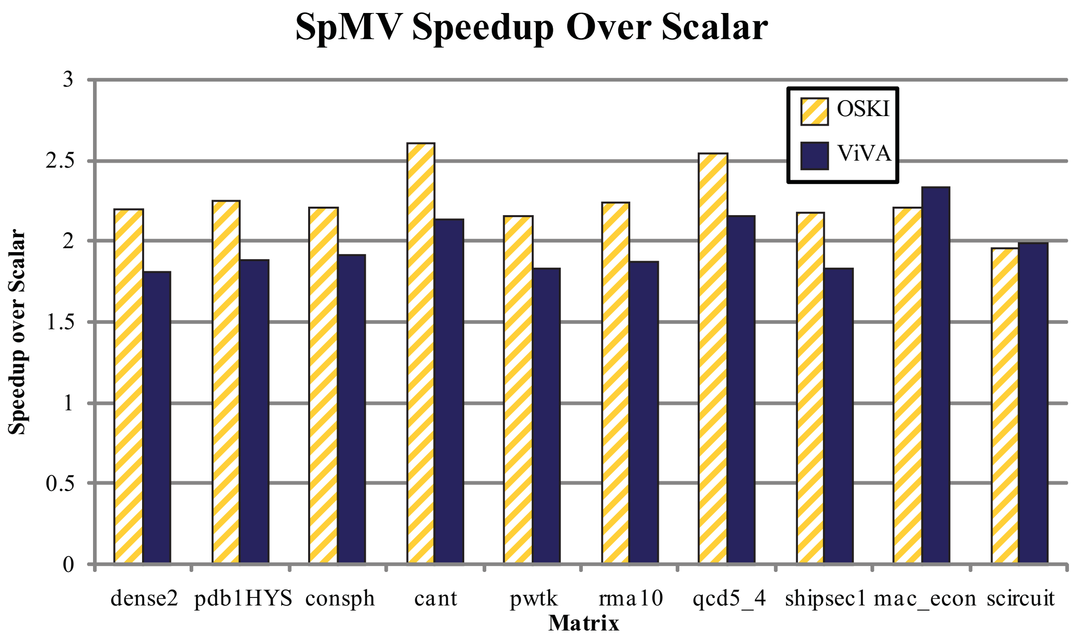


Figure 6.10: Sparse matrix-vector multiplication bandwidth, relative to scalar, for OSKI and ViVA implementations.

Figure 6.10 shows the SpMV results. The figure includes results, relative to a straightforward scalar version, for a ViVA implementation and an implementation that uses a high-performance autotuned sparse kernel library, OSKI [VDY05]. The figure does not include the significant install time for OSKI, nor does it include the runtime matrix tuning required to achieve good performance. The runtime tuning, paid at least once during the execution of an application that uses OSKI, is equivalent to approximately 20–100 sparse matrix multiplies for the test matrices.

The results show that both OSKI and ViVA are able to deliver much better performance than the straightforward scalar implementation. In most cases, OSKI is able to deliver better average bandwidth than ViVA, which reflects OSKI's optimized library procedures and runtime tuning call. Note that ViVA's performance is close to what OSKI provides in all cases, and is even slightly better on the least-structured matrices, even though its code is straightforward and it does not require runtime tuning.

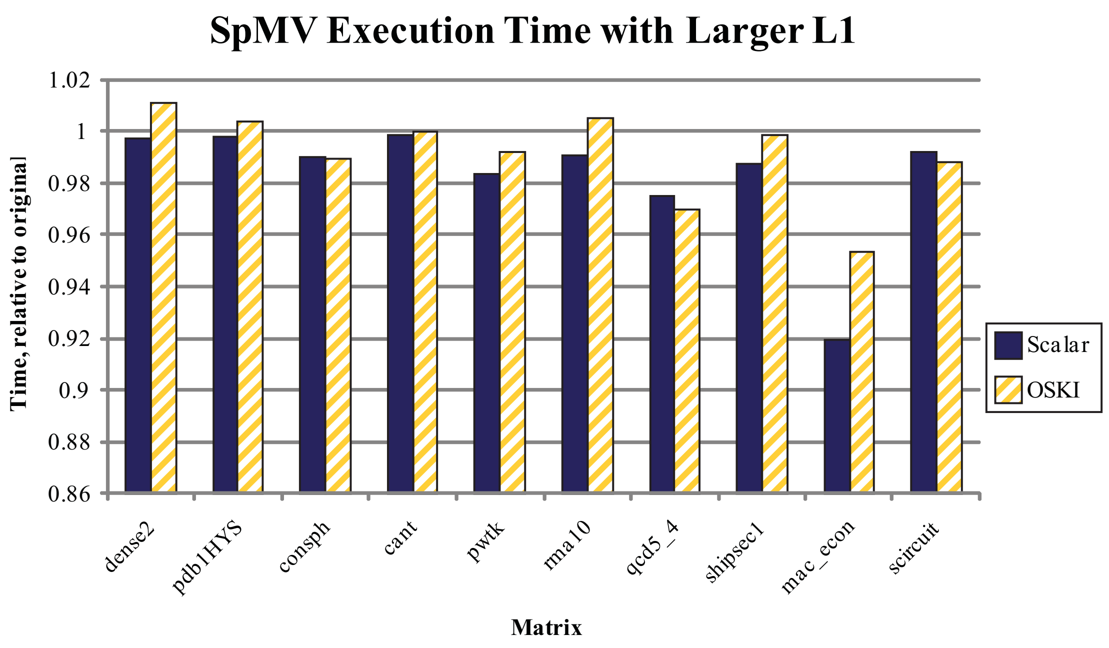


Figure 6.11: Sparse matrix-vector multiplication execution time with larger L1 cache, relative to original time. Lower is better. Note that the y axis does not begin at 0.

Once again, I ran experiments to compare the performance of the scalar code running on the original processor to the performance with a larger L1 cache. As with the corner turn case, the experiment uses a L1 cache size of 64 KB and the same access time as the smaller cache. Figure 6.11 shows the results. Performance for most matrices barely improves with the larger cache. For one of the matrices, mac_econ, the execution time for the scalar code using the larger cache is almost down to 0.9 times the execution time with the regular cache; OSKI's performance for that matrix gets about half of that benefit. The improvement is significant, but is still dwarfed by ViVA's result of providing over twice the average bandwidth that the original scalar code can achieve.

The reduction in DRAM accesses is much less dramatic for SpMV than it is for corner turn. Figure 6.12 shows the DRAM reads observed for the OSKI and ViVA implemen-

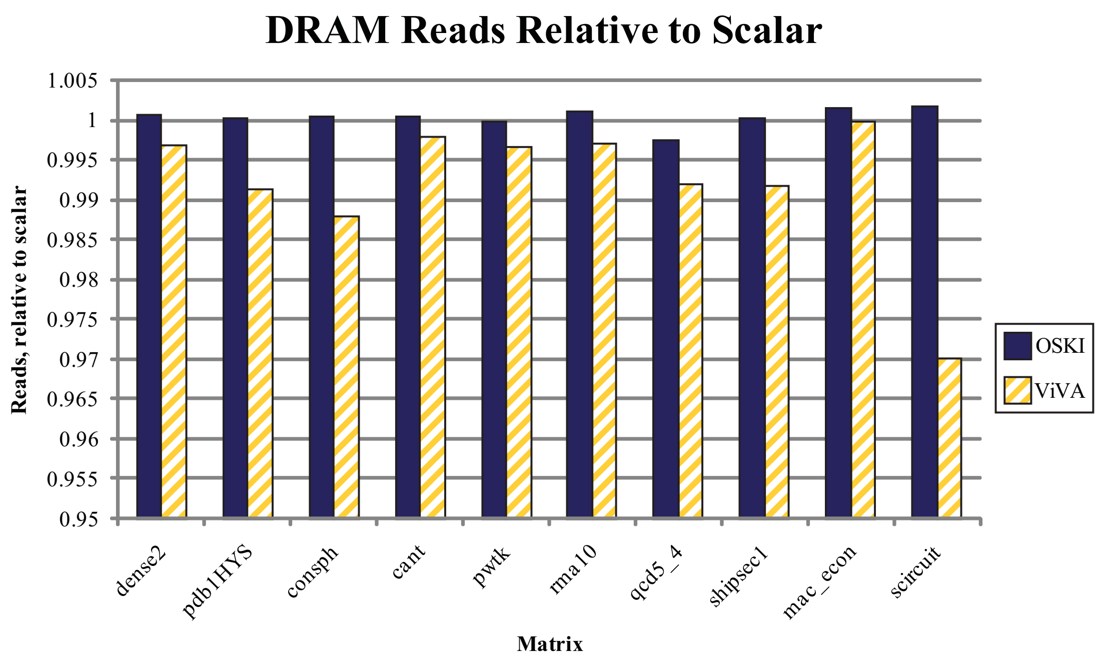


Figure 6.12: Sparse matrix-vector multiplication DRAM accesses. Note that y axis does not begin at 0.

tations, relative to the scalar code. In some cases, OSKI actually performs more DRAM accesses than the original code. OSKI restructures the matrix in order to obtain better cache performance, which can lead to different memory access behavior. Many of the techniques that improve cache performance rearrange the order of cache accesses, in order to take advantage of hardware prefetching behavior. The optimized behavior can have much better execution performance, even if the application transfers the same amount of data from DRAM to the L2 cache. In some ways, the ViVA code is similar, although it does not rely on hardware prefetching: performance comes through presenting many concurrent accesses to the memory system, not through reduction in memory data transfer.

In essence, autotuning and ViVA both attack the problem of scalar code that delivers poor performance. Autotuning focuses on wringing the most performance from the existing architectural features, avoiding the task of understanding the complexities of multi-level scalar cache and hardware prefetchers by taking advantage of a computer's ability to experiment with many combinations of code alternatives. ViVA, on the other hand, avoids the complexity from the beginning by providing a simpler path to memory. Fortunately, the idea of autotuning can be applied to any architectural feature, including ViVA: even though vectorized memory accesses in ViVA tend to perform better than straightforward scalar accesses, autotuning can still be used to experiment with code structure and organization to maximize performance.

6.5 Summary

This chapter presented the performance results for ViVA. First, I ran a series of microbenchmarks to test execution speed for some basic memory access patterns. The results show that ViVA is able to deliver significantly better performance than scalar code, without relying on a power-hungry hardware prefetcher.

I also look at performance on corner turn, used in multi-dimensional FFTs and other applications that need to rearrange large amounts of data, and sparse matrix-vector multiplication, used in many scientific simulations. ViVA gives much better results than regular scalar code, and gives performance close to highly-optimized autotuned libraries.

Code	Base version	ViVA speedup (median)
Unit stride	Optimized scalar	1.46
Strided	Scalar	1.92
Indexed	Scalar	1.80
Stanza triad	Optimized scalar	1.63
Corner turn	Scalar unrolled	1.88
		2.95 (ViVA blocked)
	Scalar blocked	0.70
		1.11 (ViVA blocked)
SpMV	Scalar	1.90
	Scalar autotuned	0.84

Table 6.4: Summary of ViVA speedup, relative to scalar code. Speedups of greater than 1.0 indicate performance improvement. MVL of 64 used for all ViVA runs. ViVA code unblocked unless otherwise noted.

Table 6.4 summarizes the results. ViVA gives a significant benefit over both straightforward and optimized scalar code. Autotuned scalar programs give better results than simple

ViVA examples, but ViVA and autotuning can be combined to deliver better performance than either is able to deliver on its own.

In Chapter 7, I will conclude my thesis by summarizing my contributions and the overall results of my work. Finally, I will take a look at some future directions towards which my research could lead, focusing on methods of extending simple vector microprocessor extensions to chip multiprocessors.

Chapter 7

Conclusions and Future Directions

For many years, microprocessor manufacturers were able to deliver greatly increasing single-thread performance by increasing clock rate and chip complexity. Recently, though, power limitations have caused that performance improvement to slow or halt. Chip architects have started to explore new design paths and focusing on energy-efficient processing.

Vectors are an excellent match for the goal of increasing throughput by taking advantage of parallelism — in this case, data-level parallelism — simply and efficiently. In this dissertation, I explored methods that allow microprocessors to add low-complexity vector extensions in an effective manner.

7.1 Summary of Contributions

My specific contributions in this thesis include:

- The design of ViVA, the Virtual Vector Architecture, which is a new extension to traditional microprocessors that allows them to take advantage of vector-style memory instructions. Vector transfers occur between a new buffer and DRAM; individual elements from the buffer can be transferred to the processor core, which can operate on them.

- The design and description of a technique that allows existing SIMD microprocessors to extend their ISA to a true vector ISA, in a simple manner. My approach details the steps needed to extend SIMD instructions in a way that allows initial vector implementations to be low cost — virtually the same cost as the original SIMD implementation — but easily extendable to higher performance and greater latency tolerance for future implementations.
- A comparison showing the similarities and differences between SIMD extensions and true vector processing, and a discussion of the implications of those differences.
- The modification of a cycle-accurate full-system simulator to allow it to be easily extended at runtime. The modifications allow new instructions to be added, new physical state to be created and associated with existing system objects, and simulator execution flow to be observed and modified.
- The evaluation of ViVA's performance in a number of different configurations, using the modified simulator to execute code that represents patterns from the Berkeley View dwarfs that are most relevant to my work.

7.2 Future Work

My work explored some of the details of extending microprocessors with vector extensions, but there is much more that can be done.

7.2.1 Further Evaluation

The first logical extension to this dissertation is the continuation of the evaluation of ViVA. SpMV and corner turn are useful beginnings, but many of the dwarfs described in Section 4.3 exhibit some degree of data-level parallelism that would work well with ViVA.

7.2.2 Vector CMPs

Microprocessor manufacturers have recently begun to embrace chip multiprocessors. Most likely, the trend of including greater numbers of cores on a single die will increase. A natural extension of my work is to examine ways that it interacts with multiple cores on a single die.

Vectors can help in striking a balance in regards to a controversial issue in the design of chip multiprocessors: should cores be large or small? Small, simple cores are more power efficient, and can lead to greater overall throughput; on the other hand, application performance will be limited by Amdahl's law if no individual core can deliver high single-thread performance. Vectors are low complexity, but can provide high performance for some applications.

CMPs can help deal with an important question in vectors, as well. Vectors are forced to deal with an inherent tradeoff between efficient use of chip area over a wide range of applications, and high performance on programs that contain a large amount of data-level parallelism. Adding lanes to vector processors increases peak performance, but reduces the number of applications that can use all of the hardware. Splitting the lanes over multiple processors has the potential to allow greater use of the hardware, while retaining a high overall throughput.

Designing a chip that specifically takes advantages of chip multiprocessors — multiple cores on a single die, rather than cores split across different physical silicon chips — will offer architects unique opportunities. It is possible just to include multiple copies of a processor on one larger die, but that approach wastes the possible benefits. On-chip communication is much faster than messages that have to pass through chip output drivers, a circuit board bus, and input drivers. Power consumption is lower as well.

Three main areas are critical for vector CMPs:

- synchronization;
- virtually combined datapaths; and

- cross-core data communication.

Synchronization

Synchronization plays a large role in multiprocessing. Currently, most CMPs synchronize through coherency mechanisms, which often use the L2 cache hardware as a communication path. Unfortunately, latency to the L2 cache can be quite large.

Blue Gene/L and other multiprocessors have examined approaches for improving synchronization performance, such as using a dedicated barrier network [GBC$^+$05]. Other researchers have looked at different techniques specific to chip multiprocessors [SGC$^+$05].

Vector CMP synchronization techniques will likely be similar to synchronization techniques for other processors, but a vector's organization of shared control of multiple lanes might result in slightly different tradeoffs or opportunities. Further research could help clarify the situation and provide insight as to which techniques work the best.

Virtually combined datapaths

Vector programs can work with different numbers of lanes. In fact, vector applications are typically unaware of the number of lanes available on a processor. Thus, vector CMPs have the opportunity to change the number of lanes available to different programs dynamically: multiple vector cores can work together on a single application, by sharing program control over the lanes of all cores.

Virtually combined datapaths offer a number of benefits. Individual cores can have a relatively small number of lanes, so that many applications can run at full efficiency. Applications can run on multiple cores, for greater performance. A processor can support many simultaneous hardware threads, or appear to have fewer threads for when an application cannot support a large amount of thread-level parallelism.

There are a number of issues and challenges that come with the idea of combining vector datapaths. It might simplify application programming if the operating system was able to decide when to combine datapaths, but applications might be able to improve hardware utilization if they had a way to communicate the number of lanes they could use. If applications are going to participate in the act of combining datapaths, how should the programming model change? The possibility of changing the number of cores dedicated to a single program during its execution would allow the processor to use resources more efficiently, but would also demand careful design. Greater numbers of lanes per core reduces control overhead, but also reduces the flexibility of the processor to offer a wider range of threads; the ideal ratio involves tradeoffs that should be investigated.

Cross-core data communication

Generally, vectors work most efficiently when data stays within its own lane. Nevertheless, programs sometimes need to perform some amount of cross-lane communication. Individual vector processors handle data movement in a variety of ways, from dedicated buses to forcing communication to occur through memory.

If an architecture can combine multiple vectors cores and make them appear as a single processor, it must decide how to deal with cross-lane communication — now cross-core communication. Processors could use the memory path to communicate data across cores, but there are a number of drawbacks to that approach: even though vectors tolerate latency, memory-bus communication might be slow enough to have a large effect on overall program performance; worse yet, using the memory bus would consume precious memory bandwidth.

An alternative is including a dedicated bus. Vector CMPs could use a single-purpose low-latency bus for implicit cross-core communication. Such a design raises many questions. How much benefit would a dedicated bus give to overall performance? Would power consumption make such a design infeasible? What are the tradeoffs of including buses that only connect a small number of cores, versus a more flexible design that would allow many cores to communicate? How would a hierarchical design work?

7.2.3 Other Low-complexity Vector Extensions

In this dissertation, I explored a few ideas of low-complexity vector microprocessor extensions. Those concepts, though, were only a few points in a wide spectrum of possibilities. Further research could explore other similar ideas. A few examples include:

- A small number of vector cores as part of a heterogeneous multicore processor.
- Low-power multi-lane vector extensions, run at lower voltage and clocked slower than the scalar core.
- A processor similar to ViVA that can reallocate storage between scalar cache and vector registers.

References

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[AAA$^+$04] Pratul K. Agarwal, Richard A. Alexander, Edoardo Apra, Satish Balay, Arthur S. Bland, James Colgan, Eduardo F. D'Azevedo, Jack J. Dongarra, Jr. Thomas H. Dunigan, Mark R. Fahey, Rebecca A. Fahey, Al Geist, Mark Gordon, Robert J. Harrison, Dinesh Kaushik, Manojkumar Krishnan, Piotr Luszczek, Anthony Mezzacappa, Jeff A. Nichols, Jarek Nieplocha, Leonid Oliker, Ted Packwood, Michael S. Pindzola, Thomas C. Schulthess, Jeffrey S. Vetter, James B. White, III, Theresa L. Windus, Patrick H. Worley, and Thomas Zacharia. Cray X1 evaluation status report. Technical Report ORNL/TM-2004/13, Oak Ridge National Laboratory, January 2004.

[ABC$^+$06] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[ABI$^+$96] Krste Asanović, James Beck, Bertrand Irissou, Brian E. D. Kingsbury, and

John Wawrzynek. T0: A single-chip vector microprocessor with reconfigurable pipelines. In *Proceedings of the 22nd European Solid-State Circuits Conference*, pages 344–347, September 1996.

[AFR67] Richard A. Aschenbrenner, Michael J. Flynn, and George A. Robinson. Intrinsic multiprocessing. In *AFIPS Conference Proceedings of the Spring Joint Computer Conference*, volume 30, pages 81–86, April 1967.

[AG88] Ramesh C. Agarwal and Fred G. Gustavson. A parallel implementation of matrix multiplications and LU factorization on the IBM 3090. In *Proceedings of the IFIP WG 2.5 Working Conference on Aspects of Computation on Asynchronous Parallel Processors*, pages 217 – 221, August 1988.

[AHP06] Krste Asanović, John L. Hennessy, and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, fourth edition, 2006. Appendix F.

[Amd67] Gene M. Amdahl. Validity of a single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485. AFIPS Press, April 1967.

[Asa87] Krste Asanović. *Vector Microprocessors*. PhD thesis, University of California, Berkeley, Berkeley, California, 1987.

[Bai97] David H. Bailey. Little's law and high performance computing. Technical Report RNR, RNR, September 1997.

[BB90] Dileep Bhandarkar and Richard Brunner. Vax vector architecture. In *ISCA '90: Proceedings of the 17th annual International Symposium on Computer Architecture*, pages 204–215, New York, NY, USA, 1990. ACM.

[BC89] Marsha J. Berger and Phil Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, 1989.

[BCBY04] Christian Bell, Wei-Yu Chen, Dan Bonachea, and Katherine Yelick. Evaluating support for global address space languages on the Cray X1. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 184–195, New York, NY, USA, 2004. ACM.

[BDA01] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 237–248, Washington, DC, USA, 2001. IEEE Computer Society.

[BDM$^+$72] W. J. Bouknight, Stewart A. Denenberg, David E. McIntyre, J. M. Randall, Amed H. Sameh, and Daniel L. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, April 1972.

[BeB08] Berkeley benchmarking and optimization group. Website, 2008. `http://bebop.cs.berkeley.edu/`.

[BHZ93] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, August 1993.

[BPE$^+$04] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *SIGARCH Computer Architecture News*, 28(2):83–94, 2000.

[Buc86] Werner Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[Chi91] Tzi-cker Chiueh. Multi-threaded vectorization. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 352–361, New York, NY, USA, 1991. ACM.

[CHS$^+$99] John B. Carter, Wilson C. Hsieh, Leigh B. Stoller, Mark Swanson, Lixin Zhang, and Sally A. McKee. Impulse: Memory system support for scientific applications. *Scientific Programming*, 7(3-4):195–209, 1999.

[Col04] Phillip Colella. Defining software requirements for scientific computing. Presentation, 2004.

[Cor05] International Business Machines Corporation, editor. *IBM PowerPC 970FX RISC Microprocessor User's Manual.* International Business Machines Corporation, December 2005.

[DER89] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1989.

[DFWW03] Thomas H. Dunigan, Jr., Mark R. Fahey, James B. White III, and Patrick H. Worley. Early evaluation of the Cray X1. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 18, Washington, DC, USA, 2003. IEEE Computer Society.

[DKV$^+$01] Victor Delaluz, Mahmut Kandemir, Narayanan Vijaykrishnan, Anand Sivasubramaniam, and Mary Jane Irwin. Hardware and software techniques for controlling DRAM power modes. *IEEE Transactions on Computers*, 50(11), 2001.

[DSC$^+$07] Jim Dorsey, Shawn Searles, M. Ciraula, S. Johnson, Norman Bujanos, D. Wu, Michael Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core Opteron processor. In *IEEE International Solid-State Circuits Conference, 2007 (ISSCC 2007) Digest of Technical Papers.*, pages 102–103, February 2007.

[DSMR07] Ronald G. Dreslinkski, Ali G. Saidi, Trevor Mudge, and Steven K. Reinhardt. Analysis of hardware prefetching across virtual page boundaries. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 13–22, New York, NY, USA, 2007. ACM.

[dV03] Hans de Vries. Understanding the detailed architecture of AMD's 64 bit core. Website, September 2003. `http://chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html`.

[EAE$^+$02] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and André Seznec. Tarantula: a vector extension to the Alpha architecture. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 281–292, Washington, DC, USA, 2002. IEEE Computer Society.

[EEL$^+$97] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[EEM07] Embedded Microprocessor Benchmark Consortium. Website, 2007. `http://eembc.org/`.

[EV96] Roger Espasa and Mateo Valero. Decoupled vector architectures. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture, 1996*, pages 281–290, February 1996.

[EV97] Roger Espasa and Mateo Valero. Multithreaded vector architectures. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture, 1997*, pages 237–248, February 1997.

[EVS97] Roger Espasa, Mateo Valero, and James E. Smith. Out-of-order vector architectures. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 160–170, Washington, DC, USA, 1997. IEEE Computer Society.

[EVS98] Roger Espasa, Mateo Valero, and James E. Smith. Vector architectures: past, present and future. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 425–432, New York, NY, USA, 1998. ACM.

[Fly66] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[FR90] Dror G. Feitelson and Larry Rudolph. Distributed hierarchical control for parallel processing. *Computer*, 23(5):65–77, May 1990.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306 – 318, 1992.

[FW60] George E. Forsythe and Wolfgang R. Wasow. *Finite-Difference Methods for Partial Differential Equations*. John Wiley & Sons, Inc., 1960.

[GBC+05] Alan Gara, Matthias A. Blumrich, Dong Chen, George L. Chiu, Paul Coteus, Mark E. Giampapa, Ruud A. Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V. Kopcsay, Thomas A. Liebsch, Martin Ohmacht, Burkhard D. Steinmacher-Burow, Todd Takken, and Pavlos Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, March/May 2005.

[GBJ98] Michael K. Gowan, Larry L. Biro, and Daniel B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 726–731, New York, NY, USA, 1998. ACM.

[GOS$^+$08] Joseph Gebis, Leonid Oliker, John Shalf, Sam Williams, and Kathy Yelick. Improving memory subsystem performance using ViVA: Virtual vector architecture. 2008.

[Gro06] Greg Grohoski. Niagara-2: A highly threaded server-on-a-chip. Presentation, August 2006. Presented at Hot Chips 18.

[GWKP04] Joseph Gebis, Sam Williams, Christos Kozyrakis, and David Patterson. VIRAM1: A media-oriented vector processor with embedded DRAM, June 2004. 41st Design Automation Conference (DAC) Student Design Contest.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, fourth edition, 2006.

[IBM08] IBM. Cell broadband engine resource center. Website, February 2008. `http://www.ibm.com/developerworks/power/cell/`.

[Jes01] Chris Jesshope. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. In *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 80–88, Washington, DC, USA, 2001. IEEE Computer Society.

[KBH$^+$04] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004*, pages 52–63, June 2004.

[Keu08] Kurt Keutzer. Architecting compelling applications using structured concurrent programming. Presentation, January 2008. Presented at the Par Lab Winter 2008 Retreat.

[Kon04] Poonacha Kongetira. A 32-way multithreaded SPARC processor. Presentation, August 2004. Presented at Hot Chips 16.

[Koz99] Christoforos Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical Report UCB/CSD-99-1059, University of California, Berkeley, July 1999.

[Koz02] Christoforos Kozyrakis. *Scalable Vector Media-processors for Embedded Systems*. PhD thesis, University of California, Berkeley, Berkeley, California, 2002.

[KP02] Christoforos Kozyrakis and David Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[KS02] Tejas Karkhanis and J. E. Smith. A day in the life of a data cache miss. Presentation, 2002. Presented at the Second Annual Workshop on Memory Performance.

[LD97] Corinna G. Lee and Derek J. DeVries. Initial results on the performance and cost of vector microprocessors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 171–182, Washington, DC, USA, 1997. IEEE Computer Society.

[Lev00] Marcus Levy. EEMBC 1.0 scores, part 1: Observations. *Microprocessor Report*, 14(33), August 2000.

[Lit61] John D. C. Little. A proof for the queuing formula: L=λW. *Operations Research*, 9(3):383–87, May–June 1961.

[LKL$^+$02] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002*, pages 59–70, 2002.

[LRW05] James M. Lebak, Albert Reuther, and Edmund Wong. Polymorphous computing architecture (PCA) kernel-level benchmarks. Technical Report PCA-KERNEL-1, MIT Lincoln Laboratory, June 2005.

[LSCJ06] Christophe Lemuet, Jack Sampson, Jean-Francois Collard, and Norm Jouppi. The potential energy efficiency of vector acceleration. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 77, New York, NY, USA, November 2006. ACM.

[LSF$^+$07] Hung Q. Le, William J. Starke, J. Stephen Fields, Francis P. O'Connell, Dung Q. Nguyen, Bruce J. Ronchetti, Wolfram M. Sauer, and Eric M. Schwarz. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, November 2007.

[McK04] Sally A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, page 162, New York, NY, USA, 2004. ACM.

[MKG98] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings of the 25th Annual Inter-*

national Symposium on Computer Architecture, pages 132–141, June – July 1998.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[MSG91] Ulrike Meier, Gregg Skinner, and John A. Gunnels. Technical Report 1134, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, July 1991.

[Nia07] Niagara 2 die microphotograph. Website, August 2007. `http://markrich.files.wordpress.com/2007/08/n2_die_photo5.jpg`.

[NIY+94] Hiroshi Nakamura, Hiromitsu Imori, Yoshiyuki Yamashita, Kisaburo Nakazawa, Taisuke Boku, Hang Li, and Ikuo Nakata. Evaluation of pseudo vector processor based on slide-windowed registers. *Vol. I: Architecture, Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences, 1994.*, 1:368–377, January 1994.

[Ous82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, October 1982.

[PAC+97] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, April 1997.

[Pat04] David A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, 2004.

[Pat07] David Patterson. The parallel computing landscape: A view from Berkeley 2.0. Presentation, June 2007. Presented at the 2007 Manycore Computing Workshop.

[Paw07] Stephen Pawlowski. Petascale computing research challenges — a manycore perspective. Presentation, February 2007. Keynote presented at the IEEE 13th International Symposium on High Performance Computing Architecture (HPCA), 2007.

[PBC$^+$06] James L. Peterson, Patrick J. Bohrer, Lei Chen, Elmootazbellah N. Elnozahy, Ahmed Gheith, Richard H. Jewell, Michael D. Kistler, Theodore R. Maeurer, Sean A. Malone, David B. Murrell, Neena needel, Karthick Rajamani, Mark A. Rinaldi, Richard O. Simpson, Kartik Sudeep, and Lixin Zhang. Application of full-system simulation in exploratory system design and development. *IBM Journal of Research and Development*, 50(2/3):321–332, March/May 2006.

[PC07] Donald W. Plass and Yuen H. Chan. IBM POWER6 SRAM arrays. *IBM Journal of Research and Development*, 51(6):747–756, November 2007.

[Phi07] Stephen Phillips. VictoriaFalls: Scaling highly-threaded processor cores. Presentation, August 2007. Presented at Hot Chips 19.

[PK94] Subbarao Palacharla and Richard E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[PMSB88] Andris Padegs, Brian B. Moore, Ronald M. Smith, and Werner Buchholz. The IBM System/370 vector architecture: Design considerations. *IEEE Transactions on Computers*, 37(5):509–520, 1988.

[PW96] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[QCEV99] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a vector unit to a superscalar processor. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 1–10, New York, NY, USA, 1999. ACM.

[RBDH97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[RPK00] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro*, 20(4):47–57, July/August 2000.

[RTDA97] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. On high-bandwidth data cache design for multi-issue processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–56, Washington, DC, USA, 1997. IEEE Computer Society.

[Rus78] Richard M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[SBP$^+$03] Hazim Shafi, Patrick J. Bohrer, James Phelan, Cosmin A. Rusu, and James L. Peterson. Design and validation of a performance and power simulator for PowerPC systems. *IBM Journal of Research and Development*, 47(5/6):641–651, September/November 2003.

[Sco96] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS-VII: International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, 1996. ACM.

[SFS00] J. E. Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 260–269, New York, NY, USA, 2000. ACM.

[SGC$^+$05] Jack Sampson, Rubén González, Jean-Francois Collard, Norman P. Jouppi, and Mike Schlansker. Fast synchronization for chip multiprocessors. *SIGARCH Computer Architecture News*, 33(4):64–69, 2005.

[SHK$^+$05] Thomas Skotnicki, James A. Hutchby, Tsu-Jae King, H.-S. Philip Wong, and Frederic Boeuf. The end of CMOS scaling. *IEEE Circuits and Devices Magazine*, 21(1), January – February 2005.

[SJV04] Asadollah Shahbahrami, Ben Juurlink, and Stamatis Vassiliasis. A comparison between processor architectures for multimedia applications. In *Proceedings of the 15th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC2004)*, 2004.

[SKH$^+$99] Kentaro Shimada, Tatsuya Kawashimo, Makoto Hanawa, Ryo Yamagata, and Eiki Kamada. A superscalar RISC processor with 160 FPRs for large scale scientific processing. In *Proceedings of the International Conference on Computer Design, 1999 (ICCD '99)*, pages 279–280, October 1999.

[Smi84] James E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, 1984.

[SMR$^+$03] Baruch Solomon, Avi Mendelson, Ronny Ronen, Doron Orenstien, and Yoav Almog. Micro-operation cache: a power aware frontend for variable instruction length ISA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(5):801–811, 2003.

[SSC00] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 42–53, New York, NY, USA, 2000. ACM.

[Tan02] David Tanqueray. The Cray X1 and supercomputer roadmap. Presentation, December 2002. Presented at the 13th Daresbury Machine Evaluation Workshop.

[TDJSF$^+$02] Joel M. Tendler, J. Steve Dodson, Jr. J. S. Fields, Hung Le, and Balaram Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1), January 2002.

[TM05] Emil Talpes and Diana Marculescu. Execution cache-based microarchitecture for power-efficient superscalar processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(1):14–26, January 2005.

[TMJ07] Shyamkumar Thoziyoor, Naveen Muralimanohar, and Norman P. Jouppi. CACTI 5.0. Technical Report HPL-2007-167, HP Laboratories, October 2007.

[TSI$^+$99] Yoshiko Tamaki, Naonobu Sukegawa, Masanao Ito, Yoshikazu Tanaka, Masakazu Fukagawa, Tsutomu Sumimoto, and Nobuhiro Ioki. Node architecture and performance evaluation of the Hitachi Super Technical Server SR8000. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems*, pages 487–493, August 1999.

[VDY05] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, volume 16 of *Journal of Physics: Conference Series*, pages 521–530, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[VEV98] Luis Villa, Roger Espasa, and Mateo Valero. A performance study of out-of-order vector architectures and short registers. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 37–44, New York, NY, USA, 1998. ACM.

[WM95] William A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[Wol95] Michael Wolfe. *High-Performance Compilers for Parallel Computing*. Addison Wesley, 1995.

[WSTT05a] Mingliang Wei, Marc Snir, Josep Torrellas, and R. Brett Tremaine. A brief description of the NMP ISA and benchmarks. Technical Report UIUCDCS-R-2005-2633, University of Illinois at Urbana-Champaign, February 2005.

[WSTT05b] Mingliang Wei, Marc Snir, Josep Torrellas, and R. Brett Tremaine. A near-memory processor for vector, streaming and bit manipulation workloads. In *Proceedings of the 2nd Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC2)*, September 2005.

[WZWS08] Kun Wang, Yu Zhang, Huayong Wang, and Xiaowei Shen. Parallelization of IBM Mambo system simulator in functional modes. *SIGOPS Operating Systems Review*, 42(1):71–76, 2008.

[ZFP$^+$01] Lixin Zhang, Zhen Fang, Mide Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. The Impulse memory controller. *IEEE Transactions on Computers*, 50(11):1117–1132, 2001.