Code Optimizers and Register Organizations for Vector Architectures

By

Corinna Grace Lee

B.S.(Hon) (Simon Fraser University) 1984

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA at BERKELEY

Approved:

Chair: D. A. Patterson .... May 1, 1992 ....

Date

M. Ad[illegible]r .... 4/27/92 ....

Paul N. Hilfinger .... 5/31/92 ....

* *

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **MAY 1992**	2. REPORT TYPE	3. DATES COVERED **00-00-1992 to 00-00-1992**

4. TITLE AND SUBTITLE **Code Optimizers and Register Organizations for Vector Architectures**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

A major challenge facing computer architects today is designing cost-effective hardware that executes multiple operations simultaneously. The goal of such designs is to improve performance by taking advantage of fine-grain parallelism. In this dissertation, I study vector architectures, the oldest of several processor designs that support fine-grain parallelism. Because implementing a cost-effective processor that performs well requires studying not only the design of processors but also the design of algorithms for compilers, this dissertation encompasses aspects of both hardware and software design. In the first half of this dissertation, I demonstrate that a vector architecture is a cost-effective processor that supports fine-grain parallelism. I show that implementing a vector architecture is no more costly than implementing a superscalar architecture, which is currently popular among designers of VLSI microprocessors. I then show that programs that are rich in parallelism tend also to be vectorizable and are also the ones that execute the longest in a workload, thus demonstrating further the effectiveness of vector architectures. Finally, I show that superpipelined hardware in combination with a vector architecture can take advantage of what little parallelism is available in non-vectorizable programs. In the second half of this dissertation, I investigate the cost and performance of different organizations for a vector register file in the Cray Y-MP vector processor, an investigation that emphasizes the interaction between processor design and compiler algorithms. After showing that instruction scheduling has a major impact on how effectively more vector registers can be used, I present data from simulation experiments indicating that 16 vector registers and a list scheduling algorithm can improve performance significantly over that of 8 vector registers and the scheduling algorithm used in the Cray vectorizing compiler. I also investigate the usage of an alternative register organization, called a partitioned vector register file, which is less costly to implement than a traditional one but places some restrictions on accessing vector registers. To circumvent this restructive access, I develop an algorithm for assigning vector registers and present data showing that, when using my algorithm, the performance of a partitioned vector register file is comparable to that of a traditional one.

15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT **Same as Report (SAR)**	18. NUMBER OF PAGES **170**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Code Optimizers and Register Organizations
for Vector Architectures

Code Optimizers and Register Organizations for Vector Architectures

Corinna Grace Lee

Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

A major challenge facing computer architects today is designing cost-effective hardware that executes multiple operations simultaneously. The goal of such designs is to improve performance by taking advantage of fine-grain parallelism. In this dissertation, I study vector architectures, the oldest of several processor designs that support fine-grain parallelism. Because implementing a cost-effective processor that performs well requires studying not only the design of processors but also the design of algorithms for compilers, this dissertation encompasses aspects of both hardware and software design.

In the first half of this dissertation, I demonstrate that a vector architecture is a cost-effective processor that supports fine-grain parallelism. I show that implementing a vector architecture is no more costly than implementing a superscalar architecture, which is currently popular among designers of VLSI microprocessors. I then show that programs that are rich in parallelism tend also to be vectorizable and are also the ones that execute the longest in a workload, thus demonstrating further the effectiveness of vector architectures. Finally, I show that superpipelined hardware in combination with a vector architecture can take advantage of what little parallelism is available in non-vectorizable programs.

In the second half of this dissertation, I investigate the cost and performance of different organizations for a vector register file in the Cray Y-MP vector processor, an investigation that emphasizes the interaction between processor design and compiler algorithms. After showing that instruction scheduling has a major impact on how effectively more vector registers can be used, I present data from simulation experiments indicating that 16 vector registers and a list scheduling algorithm can improve performance significantly over that of 8 vector registers and the scheduling algorithm used in the Cray vectorizing compiler. I also investigate the usage of an alternative register organization, called a *partitioned vector register file*, which is less costly to implement than a traditional one but places some restrictions on accessing vector registers. To circumvent this restrictive access, I develop an algorithm for assigning vector registers and present data showing that, when using my algorithm, the performance of a partitioned vector register file is comparable to that of a traditional one.

To My Parents,
Hui Yuen Ming and Lai Wing Gai,
for their courage and resourcefulness

Acknowledgements

I thank the architects at Cray Research, Incorporated who are fellow advocates of vector architectures; much of the work in this dissertation benefited greatly from discussions with them. Greg Faanes introduced me to the Cray Y-MP simulator as well as *grap*, the program I used to draw all the graphs in this dissertation. James Smith encouraged me to write down my thoughts that eventually grew to become Chapter 3, *A Case for Vector Architectures*. Wei-Chung Hsu was a source of information and assistance. In addition to helping me with the integration of my scheduling algorithm into the *cft77* compiler, he also provided implementation details about the *cft77* compiler and the Y-MP processor that made my analyses in Chapters 5 and 6 realistic.

I also thank John Wawrzynek and Bradd Hart for contributions to my dissertation. The multiported register cells in Chapter 3 were designed with John Wawrzynek. The proof of optimality in Chapter 6 was worked out with Bradd Hart.

Thanks to several readers whose numerous suggestions have improved the exposition of this dissertation tremendously. James Smith and Wei-Chung Hsu commented on an earlier draft. Tom Phillipi proofread my dissertation and in the process of doing so taught me the finer points of expository writing. Paul Hilfinger provided timely comments and showed me where my explanations were inadequate for a reader knowledgeable about compilers. David Patterson read through several drafts and provided constant advice and encouragement when there was always more for me to write.

I thank David Patterson also for being my advisor. In addition to strongly supporting my thesis, he has guided me through the trials and tribulations of graduate school. During our many meetings, he has given me his enthusiasm for learning, his curiosity of things unknown, and his ability to consider an idea from many perspectives.

Funding for my years in graduate school came from a variety of sources. My first years were supported by a 1967 Science and Engineering Scholarship from the Natural Sciences and Engineering Research Council of Canada and by the SPUR/DARPA contract number N00039-85-C-0269. Later years were funded by a grant from Sun Microsystems, and my final year by the California MICRO Grant 91-123 with matching funds from Cray Research, Incorporated.

Thanks to the many people who enriched my life during graduate school. Kathryn Crabtree, Terry Lessard-Smith, and Bob Miller helped me negogiate the rules and regulations that are an inevitable part of a large institute such as the University of California. Susan Eggers and George Taylor provided unfailing support and encouragement for my abilities even when I did not believe in them. Ken Lutz and Shelly Kodimer showed me how to enjoy living in Berkeley. Margo Seltzer introduced me to soccer and the Bruisers, a diverse group of women who taught me the importance of teamwork. Yvonne Gindt and Robin Packel led our running group Monday nights, even during darkness and hail storms! Sue Dentinger, Jane Doughty, Mark Hill, Jim Larus, Diana Stone, and David Wood had the good sense to live in Madison, Wisconsin; I enjoyed my frequent visits with them during my work term at Cray Research.

Finally, my thanks to Bradd Hart who has provided me with years of emotional support, much of it delivered long distance over the phone; he has helped me to keep academic life in perspective. I can now say the words he has waited a long time to hear: I'm done.

Contents

List of Figures

Chapter 1

Introduction

In the fall of 1990, I worked at Cray Research, Incorporated with the architecture group that is designing the follow-on to the Cray Y-MP C-90, which in turn is the successor to the Cray Y-MP, the classic embodiment of a vector architecture. My project was to answer the following question:

> How many vector registers are enough to effectively use
> the functional units of the Cray Y-MP vector processor?

What this question really means, how I went about answering this question, and the actual answer itself are described in two chapters of this dissertation.

In the course of answering this question, I developed two compiler algorithms: a vector instruction scheduler and a vector register assigner. I also examined the design of a vector processor to determine the cost of implementing different configurations of vector register files. Finally, I carried out simulation experiments to evaluate the effectiveness of the algorithms I developed and to measure the performance of different register files as well as to determine the answer to the above question.

In addition to answering the above question, my dissertation also addresses the more fundamental question: "Why do research in vector architectures?" The short — but not often heard — answer is "because a vector architecture is an inexpensive processor design that supports fine-grain parallelism well." The longer version of this answer is given in two chapters: one that contrasts how vector architectures support fine-grain parallelism in comparison to other architectural classes, and another chapter that explains how vector architectures are inexpensive to implement.

1.1 Definitions

Before I present an overview of my dissertation, I discuss my usage of particular words.

In the discipline of computer science, the term *architecture* is typically used as an abbreviation for *computer architecture*, which refers to the organization and implementation of a computer. A computer, in turn, consists of three major components: processor, memory, and input/output. For my thesis, I examine the design of only the processor component, which is also commonly known as the *central processing unit* or *CPU* for short.

Hence, in this dissertation, I use the term *architecture* as an abbreviation for *processor architecture* to refer to the organization and implementation of a processor rather than an entire computer.

I make a distinction between an *operation* and an *instruction*. An *operation* is a task executed by an instruction. An *instruction*, on the other hand, is associated with a particular processor design and represents the smallest unit of work that is examined during one clock period by the instruction-issue logic in hardware. Furthermore, an instruction can cause one or more operations to execute, depending on the processor architecture, and specifies not only what operations to execute but also where the operands and results of those operations are located. In describing algorithms for code optimization, I use the more abstract term *operation* because it is less specific about its execution and hence has fewer restrictions on how it should be executed. The purpose of these algorithms is to transform a set of operations into a sequence of instructions. For example, an algorithm for instruction scheduling determines in what order a set of operations should execute, and an algorithm for register assignment determines where the operands and result of an operation are located.

Because an instruction in a vector architecture can specify more than one operation to execute, the term *instruction-level parallelism* inadequately portrays the amount of parallelism supported by a vector architecture. In contrast, the term *fine-grain parallelism* refers to the number of *operations* that can be executed at the same time and more accurately depicts the amount of parallelism supported by a vector architecture. Furthermore, it is more insightful to compare the amount of fine-grain parallelism, rather than instruction-level parallelism, supported by various architectures because different architectures can specify different amounts of work for an instruction but an operation specifies the same amount of work across different architectures. Hence, I use the term *fine-grain parallelism* to refer to the parallelism that can be supported by a uniprocessor. (Although the term *operation-level parallelism* would be a more logical choice, I use *fine-grain parallelism* instead because it is more commonly used among designers of compilers and hardware.)

Finally, I use the term *functional unit* rather than the acronym *ALU* to refer to a part of hardware that executes an operation. All types of operations, including memory accesses, are executed by some type of functional unit. A functional unit can be general purpose like an ALU or special purpose like a shifter or a floating-point adder. A memory port, which executes a memory operation, is a special-purpose functional unit that serves as the interface between a processor and its memory system.

1.2 Overview of Dissertation

This dissertation has two common themes: the use of fine-grain parallelism to improve performance, and the cooperation between software and hardware to design a cost-effective processor, particularly one that supports fine-grain parallelism. An example of the second theme is provided by the question posed in the opening paragraph. Because a solution to this question must provide improved performance at a reasonable cost, answering this question requires understanding the interaction between a code optimizer and the design of a register organization. Each of the four major chapters in this dissertation develops these two themes, with the first two chapters emphasizing fine-grain parallelism and the last two concentrating on how aspects of a vector processor interact with aspects of a compiler.

In addition to providing background information about vector architectures and fine-grain parallelism, the first two major chapters address the question "Why do research in vector architectures?" I begin Chapter 2, *Fundamentals of Vector Architectures*, with a short discussion on data dependence, a concept that affects both hardware and software when using parallelism. Next I describe how the hardware features of a vector architecture, and in particular the vector instruction, support fine-grain parallelism and contrast these features with those of other architectures that support fine-grain parallelism. Because not all parts of a program can be executed with vector instructions, I then describe the properties of a vectorizable program fragment and explain how a compiler can identify these.

At the ASPLOS[1] conference in 1991, cries of "Vector architectures are history!" were heard throughout the sessions. The rapid advancement of VLSI technology and the trend in microprocessor design towards superscalar architectures have led to this prediction of the vector architecture's imminent demise. Because of this dire prediction, many people may mistakenly believe that research in vector architectures is a futile activity. To counter this prediction as well as these mistaken beliefs, in Chapter 3, *A Case for Vector Architectures*, I present arguments with accompanying data that emphasize the hardware and software strengths of vector architectures and the weaknesses of superscalar ones.

Chapter 4, *Common Experimental Framework* is a short one in which I describe the basic vector hardware, performance tools, and workload used in the empirical studies carried out in the next two chapters.

In the last two major chapters, I answer the question posed in the opening paragraph. In Chapter 5, *Register Usage and Instruction Scheduling*, I analyze the performance of using more vector registers. As part of this analysis, I also show that algorithms for instruction scheduling have a major impact on how effectively more registers are used to improve performance. I present empirical data that determines the minimum number of vector registers needed to significantly improve performance over the current design of the Cray Y-MP vector processor. In Chapter 6, *Bus Usage and Register Assignment*, I examine the cost of using more vector registers and investigate a special organization, I call a *partitioned vector register file*, that is less costly but more restrictive in its access to individual vector registers. For this investigation, I describe a register assignment algorithm I developed that uses such a restrictive organization with minimal loss in performance. I also present empirical data for choosing a register organization that is most cost-efffective for improving performance.

In the closing chapter, *Concluding Remarks*, I summarize my work by highlighting the contributions of this dissertation and finish by discussing extensions of this work for future study.

[1] *ASPLOS* is an acronym for *A*rchitectural *S*upport for *P*rogramming *L*anguages and *O*perating *S*ystems.

Chapter 2

Fundamentals of Vector Architectures

In this chapter, I describe the fundamentals of vector hardware and compilation to demonstrate how suitable a vector architecture is for supporting fine-grain parallelism. Because my dissertation examines the interaction between vector hardware and compiler algorithms, the information in this chapter also serves as the background material for the remaining chapters. This discussion outlines problems faced by any architecture that supports fine-grain parallelism and differentiates what is specific to a vector architecture. I begin with a short discussion on data dependence, a concept that is fundamental to any architecture that uses parallelism. I then describe the hardware capabilities needed to support fine-grain parallelism and contrast how a vector architecture and three other architectures provide this support. Because not all parts of a program can be executed with vector instructions, I next describe the properties of a vectorizable program fragment, using the hardware as the basis for justifying each property, and outline how a program is transformed into vectorized code.

2.1 Data Dependence

Correct parallel execution requires that multiple operations be executed simultaneously without changing a program's functionality, which is typically defined by the output produced by the scalar version of the program. Imprudently executing any operations in parallel will likely alter a program's functionality. One way to maintain correct functionality is to guarantee that accesses to the same storage location occur in the same order as they do in the scalar version. In other words, references to the same location must be serialized whereas references to different locations can execute in parallel. Two references that access the same storage location form a *data dependence*[1] Moreover to ensure that the correct value is always in the common location, this dependence relation specifies the order in which the two references can execute: the reference that accesses the common storage location first must execute first [120]. Hence, any architecture that supports fine-grain parallelism must

[1] Another type of dependence that occurs in a program is *control dependence*. I focus only on data dependence in this discussion. Ferrante, Ottenstein, and Warren show how a compiler can uniformly treat control and data dependences [39].

DEPENDENCE TYPES		ACCESS TYPES AND ORDER
HARDWARE NAME	COMPILER NAME	
RAW	flow dependence	read after write
WAR	anti-dependence	write after read
WAW	output dependence	write after write

Figure 2.1: Hardware and Compiler Names for Dependence Types

This table shows the names given by the hardware and compiler communities to the different types of data dependences. The hardware names refer to dependences that occur in registers and the compiler names refer to those that occur in memory. A fourth combination — RAR or input "dependence" — is not really a dependence because no state is changed. RAW or flow dependences are considered the only true data dependences in that they cannot be eliminated without jeopardizing correct functionality. WAR/anti and WAW/output dependences occur because storage is finite; if every newly-created value were assigned its own storage location, these dependences would never occur.

maintain the orderings specified by all data dependences.

A dependence can be classified by access type and the order in which two references occur. In addition, data dependences can occur in two different storage locations: registers and main memory. Although techniques in either hardware or software can be used to detect, work around, or even eliminate data dependences, hardware normally handles dependences occurring in registers and a compiler handles ones occurring in memory. Consequently, the hardware and compiler communities have given different names, which are listed in Figure 2.1, to the same dependence type.

Register dependences can either be avoided by the software or surmounted by the hardware. With information about hardware, a compiler can assign values to registers in order to avoid WAR and WAW dependences. These dependences can still occur, however, if hardware is upgraded to provide more parallelism than what was compiled for. Fortunately, with appropriate hardware, register-dependent operations can execute correctly in parallel. This is because register addressing is explicit, allowing hardware to accurately recognize when register dependences occur. An example of a hardware mechanism that eliminates WAR and WAW dependences is *register renaming* (to be discussed in Section 2.2.1) whereas *data forwarding* uses bypass paths to ensure that a RAW-dependent instruction reads the correct data value.

Memory dependences can also be detected and handled by either the hardware or the compiler. Unfortunately, there is no comparable hardware technique to register renaming or data forwarding that will allow dependent memory references to execute simultaneously. At best, hardware that allows out-of-order execution, which is also known as dynamic scheduling, can be used to minimize idle cycles due to dependent memory references. Alternatively, a compiler can detect memory dependences using analysis techniques that provide information not only about memory addresses but also about the access pattern of related memory references. Once dependences are detected, a compiler can order the operations to execute in parallel in such a way that data dependences are still preserved.

How hardware or compilers handle data dependences is one central theme of this dissertation. Section 2.2 provides a fuller description on handling register dependences in the context of vector hardware and Section 2.3 details how memory dependences are detected by the compiler. Compilation techniques for scheduling around memory dependences and avoiding register dependences when using a vector architecture are examined in Chapters 5 and 6, respectively.

2.2 Hardware Support for Fine-Grain Parallelism

A major challenge facing computer designers today is determining how to execute multiple operations simultaneously to improve performance. For single, instruction-stream, load/store architectures, research groups are currently investigating four approaches: superpipelined, superscalar, VLIW,[2] and vector architectures. To support fine-grain parallelism, irrespective of the architectural approach, hardware must be able to perform simultaneously more than one instance of the basic sequence of tasks for executing an operation. In other words, hardware must be able to simultaneously:

1. initiate multiple operations,
2. fetch multiple operands,
3. execute multiple operations, and
4. store multiple results.

In the rest of this section, I will describe and contrast how each architectural class performs these tasks. Because this dissertation focuses on vector architectures as embodied by the Cray Y-MP, I will give more details on this class of architecture than on the other three. Although all four tasks are equally important in determining the maximum amount of parallelism that can be realized, initiating multiple operations has received the most attention from computer designers, who have named the architectural classes on the basis of how each class accomplishes this task.

2.2.1 Multiple Operation Initiation

Initiating operations and issuing instructions are closely-related activities. Initiating an operation is the first task in the basic sequence for executing an operation, whereas the method for issuing instructions determines how this task is done for more than one operation at a time. The limitation of issuing one instruction per clock period has become known as the *Flynn bottleneck* (based on [42]). Both superscalar and VLIW architectures overcome this bottleneck in order to initiate more than one operation simultaneously. This

[2] VLIW is an acronym for "Very Long Instruction Word," suggesting many operations per instruction.

is not a requirement, however, because both superpipelined and vector architectures support fine-grain parallelism and still issue only one instruction each clock period.

In this subsection, I describe how each architectural class initiates multiple operations simultaneously. This is accomplished by extending the instruction-issue mechanism of a scalar, pipelined architecture in a manner that is reflected by the name of a class. Because there is a large body of programs already compiled for scalar architectures, a desirable property of such an extension is that the resultant architecture can execute these *scalar-compiled binaries* with the possibility of improving performance, a characteristic of both superscalar and superpipelined architectures but not VLIW and vector ones.

In addition to describing how multiple operations are initiated at once, I provide examples of implementations in each class. I also discuss hardware mechanisms for handling register dependences because such mechanisms affect how frequently instructions are issued, which in turn affect how much parallelism can occur.

Superpipelined Architectures

This architectural class supports fine-grain parallelism by using deeper pipelines and a higher clock rate than those used in a basic scalar machine [70]. For example, whereas a pipelined machine will have a 4- or 5-stage pipeline, a superpipelined machine will have an 8- or 10-stage pipeline and half the cycle time. The higher clock frequency is obtained through the deeper pipelines, hence the name *superpipelining*. Although one instruction still specifies only one operation, if the cycle time of a pipelined machine is considered the basic time unit, then a superpipelined machine gives the appearance of multiple-operation initiation by issuing instructions at a faster rate. However, the inability to control clock skew will ultimately limit how fast the clock rate can be made and hence how much performance can be gained by this approach [57].

Because, in the strictest sense, operations are not actually issued at the same time, the organization of a superpipelined machine is not significantly different from that of a pipelined machine; the main difference lies at the level of hardware implementation. As a result, scalar-compiled binaries can be executed with the possibility of improving performance.

Superpipelined processors have been implemented at the high end of the cost spectrum beginning with the CDC 6600 in 1964 and continuing to this day with the scalar units of the Cray processors. More recently, in the 1990's the microprocessor world, which is at the lower end of the cost spectrum, produced the MIPS R4000, an 8-stage pipelined implementation of the MIPS architecture [71], and the DEC EV4, a 64-bit implementation of the new Alpha architecture [75].

Superscalar Architectures

For this architectural class, fine-grain parallelism is achieved by issuing multiple, scalar instructions in the same clock period, hence the name *superscalar*. Typically, the operational types of the simultaneous operations are different. Because one instruction still specifies one operation and the instruction-issue unit dynamically determines how many instructions can be issued, scalar-compiled binaries can be executed with the possibility of improving performance.

Among the various architectural approaches, the superscalar one is the newest and is currently the focus of the commercial microprocessor community. In 1989, the first superscalar implementations, the IBM RS/6000 and the Intel i860, were announced. The 1990 and 1991 Hot Chips Symposium hosted several presentations that described superscalar designs, such as the Metaflow Lightning and the Sun SuperSPARC [94, 13].

To increase the number of instructions that can execute simultaneously, additional hardware allows multiple register-dependent instructions to issue in the same clock period. Bypass hardware to forward data as it becomes available allows RAW-dependent instructions to be issued together [13]. Because a limited number of registers cause WAR and WAW dependences to occur, register-renaming hardware eliminates these dependences by providing more physical registers than the instruction set can specify. When a logical register is used by two instructions in either a WAR or WAW dependence, the register is mapped to two different physical registers, thus removing the dependence and allowing the instructions to execute at once. Register renaming was first implemented in the IBM 360/91 [112] and is included in contemporary superscalar processors such as the IBM RS/6000 and Metaflow Lightning [89, 94].

VLIW Architectures

Like a superscalar architecture, a VLIW architecture also issues more than one operation per clock period, where the type of each operation is usually different. However, whereas each operation in a superscalar architecture requires a separate instruction, in a VLIW architecture many operations are encoded in a single instruction, resulting in a Very Long Instruction Word from which this architectural class takes its name. Moreover, a compiler is responsible for grouping operations into a VLIW instruction, which means that, unlike execution on a superpipelined or superscalar architecture, scalar-compiled binaries cannot take advantage of the parallelism when executing on VLIW hardware.

VLIW architectures have typically implemented as minisupercomputers. An early VLIW computer is the Floating Point Systems AP-120B, which was first delivered in 1976 [59]. In 1983, J.A. Fisher actually coined the term *VLIW* to describe the ELI-512, a computer that was built at Yale University [40]. The Multiflow Trace computers, the commercial version of the ELI-512, became available in 1987 [23]. In 1989, Cydrome announced its VLIW computer, the Cydra 5 [97]. The Intel iWarp, a commercial realization of a research project at Carnegie-Mellon University, was developed in the same time period [6, 14].

In addition to extracting parallelism, a VLIW compiler is also responsible for handling register dependences. The hardware contains no synchronization mechanisms and, in particular, does not check for register dependences [23, 97]. It is the compiler's responsibility to ensure that operations use the correct register values. Although this precludes binary compatibility, it simplifies the control logic in the hardware.

Vector Architectures

Like a VLIW architecture, a vector architecture issues more than one operation per clock period and relies on a compiler to extract any parallelism that can use its hardware. However, whereas a VLIW instruction causes multiple operations of different types to initiate simultaneously in different functional units, a vector instruction causes multiple operations of the same type to initiate sequentially in one functional unit. The name *vector* comes from the fact that a vector instruction operates uniformly on a set of related data, such as a column or row in a matrix, to produce another vector of data.

There have been many commercially successful implementations of vector architectures, beginning with the load/store architecture of the Cray-1 in 1976 [110, 101, 8]. Since then Cray Research Incorporated has produced a succession of vector machines: the Cray X-MP in 1982, the Cray-2 in 1985, the Cray Y-MP in 1988, and the Cray Y-MP C-90 in 1991. In 1983, three Japanese vendors entered the supercomputer market with the Fujitsu VP200, the Hitachi S810, and the NEC SX/2 [85, 88, 116]. The 1980's also saw less costly vector implementations. For example, in 1985, Convex produced the superminicomputer, the C1, and now offers a multiprocessor version, the C2 [68, 19]. In 1986, IBM introduced the System/370 vector architecture, a family of vector computers designed to cover a range of cost/performance implementations, the first being the 3090 Vector Facility [16]. In 1988, the Ardent Titan entered the market as a superworkstation with graphics capabilities [31]. In 1989, Digital Equipment Corporation formally introduced the VAX 6000 Model 400 series, which extended the VAX architecture to include vector processing [102]. The most recent development occurred in 1991 when Thinking Machines Corporation included a vector execution unit as part of the processor node in its Connection Machine-5, a massively parallel processor [109], and both NEC and Fujitsu have fabricated a vector processor on a single VLSI chip [90, 64].

Because only one instruction is issued during each clock period, how a vector architecture initiates more than one operation per clock period is less obvious than how this is accomplished by the other classes of architectures. Moreover, a vector instruction can take a long time to execute because, in a load/store architecture, such as the Cray Y-MP, the maximum number of operations executed by a vector instruction is equal to the number of elements in a vector register, or 64 in the case of the Cray Y-MP. Hence, a Y-MP vector instruction, which sequentially initiates its operations, can take over 64 clock periods to execute. Two vector instructions can execute concurrently, however, when the instructions use completely different resources, such as separate functional units and distinct registers for operands and results. In this way, two operations, one from each instruction, will initiate simultaneously even though the instructions still issue one at a time (see Figure 2.2). Multiple operations will also commence simultaneously when any scalar instructions are issued during the execution of a vector instruction.

Vector hardware, unlike VLIW hardware, provides interlock logic for its registers. This means that without additional hardware a register-dependent vector instruction must wait until the independent vector instruction has finished using the common register, as Figure 2.3(a) shows. Two hardware mechanisms, however, can be used to eliminate this large loss in potential parallelism. If two vector instructions form a RAW dependence, *chaining hardware* allows the dependent instruction to begin executing as soon as the first opera-

Independent Vector and Scalar Instructions

```
TOP:  V0 ← M[R1]
      V3 ← V1+V2
      R1 ← R1+64
      R0 ← R0−64
      BNEZ R0,TOP
```

Execution Trace of Vector and Scalar Operations

	FUNCTIONAL UNITS		
TIME	LOAD PORT	ADDER	SCALAR UNIT
0	V0 ←M_0[R1]		
	M_1	V3 ← V1$+_0$V2	
	M_2	$+_1$	R1 ← R1+64
	M_3	$+_2$	R0 ← R0−64
	M_4	$+_3$	BNEZ R0,TOP
	M_5	$+_4$	
↓	M_6	$+_5$	

Figure 2.2: Multiple Operation Initiation with Independent Vector Instructions

This execution trace illustrates how independent vector and scalar instructions cause parallel initiation of more than one operation. A scalar instruction causes only one operation to execute while a vector instruction causes multiple operations (64 in this example) of the same type to initiate sequentially in one functional unit. The number of operations executed by a vector instruction is typically stored in a *vector length* register, called `VL` in the Cray Y-MP.

A vector instruction is identified by its use of vector registers, such as `V0` or `V2`, each of which consists of multiple registers. (A more thorough description of vector registers is provided in the next subsection.) The vector instruction `V0<-M[R1]` transfers into vector register `V0` data from `VL` consecutive memory locations starting at the address stored in register `R1`. The vector instruction `V3<-V1+V2` stores into the i^{th} register of vector register `V3` the sum of the i^{th} registers of vector registers `V1` and `V2`.

At most, only one instruction is actually issued every clock period. But because a vector instruction causes multiple operations to initiate over time, the overlapped execution of vector instructions allows fine-grain parallelism to occur.

tion of the independent instruction has finished executing. Figure 2.3(b) gives an example of chaining. On the other hand, with *tailgating hardware*, a WAR-dependent instruction is issued immediately following the independent instruction as shown in Figure 2.3(c). A WAR dependence is avoided because register reads occur near the beginning of a pipeline and register writes occur at the end, thus guaranteeing that the independent operation reads the register before the dependent one writes into it. In summary, using chaining or tailgating allows multiple operations to initiate simultaneously in the presence of dependent vector instructions. Neither of these approaches, however, affects the instruction set design; their main impact is to improve performance by increasing the opportunities for fine-grain parallelism.

Chaining and tailgating hardware were first implemented in different Cray computers. Because the Cray-1 uses single-ported register cells, it provides a limited form of chaining; a RAW-dependent instruction can begin executing within a certain period of time, called the *chain slot time*, after the independent instruction has issued. If, for some other reason, the dependent instruction cannot issue within the chain-slot time, it must wait until after the independent instruction has finished executing completely. By using dual-ported register cells, the Cray X-MP and Y-MP implement fully flexible chaining; a RAW-dependent instruction can begin executing any time after the first operation of the independent instruction has finished executing. Rather than using chaining, the Cray-2 implements tailgating. To date, there is no vector machine that implements both chaining and tailgating.

2.2.2 Multiple Operands and Results

The second and fourth tasks in the basic sequence for executing an operation fetch and store data for an operation. Simultaneously handling multiple operands and results in a load/store architecture requires a register file with multiple read and multiple write ports. A register file with R read-ports and W write-ports provides the capability of reading R registers and writing W registers during the same clock period. Figure 2.4 shows three distinct configurations for implementing a multiported register file: *monolithic*, *partitioned*, and *distributed*. Combinations of these types of register files are also possible. In this subsection, in addition to explaining how these types of register files provide simultaneous, multiple access, I also:

- discuss how these configurations differ by examining how well individual registers are connected to all of the available functional units; and
- give examples for each type of register file. Although these configurations can be used with any of the architectural approaches, there is a natural tendency for an architectural class to use a specific configuration.

The most straightforward configuration for implementing a multiported register file is to use a register cell with multiple read- and multiple write-ports. Although the number of registers actually accessed is determined by the number of ports, all registers in such a *monolithic register file* are available simultaneously as an operand or a destination for any functional unit. This type of register file is also known as a *shared register file* [105, 15].

Execution Traces of Dependent Vector Instructions

Vector Instructions
with RAW and
WAR Dependence

V0 ← M[R1]
V2 ← V0+V1
V0 ← V3*V4

	(a) no special hardware			(b) chaining hardware			(c) tailgating hardware		
TIME	LD	+	*	LD	+	*	LD	+	*
0	M_0			M_0			M_0		
	M_1			M_1			M_1		
	M_2			M_2			M_2		
	M_3			M_3			M_3		
	M_4			M_4	$+_0$		M_4		
	M_5			M_5	$+_1$		M_5		
	M_6			M_6	$+_2$		M_6		
	M_7			M_7	$+_3$		M_7		
	:			:	$+_4$		:		
	:			:	$+_5$		:		
		$+_0$			$+_6$			$+_0$	
		$+_1$			$+_7$			$+_1$	$*_0$
		$+_2$			:			$+_2$	$*_1$
		$+_3$			:			$+_3$	$*_2$
		$+_4$				$*_0$		$+_4$	$*_3$
		$+_5$				$*_1$		$+_5$	$*_4$
		$+_6$				$*_2$		$+_6$	$*_5$
		$+_7$				$*_3$		$+_7$	$*_6$
		:				$*_4$		:	$*_7$
		:				$*_5$		:	:
			$*_0$			$*_6$			:
			$*_1$			$*_7$			
			$*_2$			:			
			$*_3$			:			
			$*_4$						
			$*_5$						
			$*_6$						
			$*_7$						
			:						
V			:						

Figure 2.3: Multiple Operation Initiation with Dependent Vector Instructions

These execution traces illustrate how chaining and tailgating hardware increase parallelism in the presence of RAW and WAR dependences between vector registers. The above chart assumes that the latency for one load operation is four clock periods. The notation for vector instructions is explained in Figure 2.2.

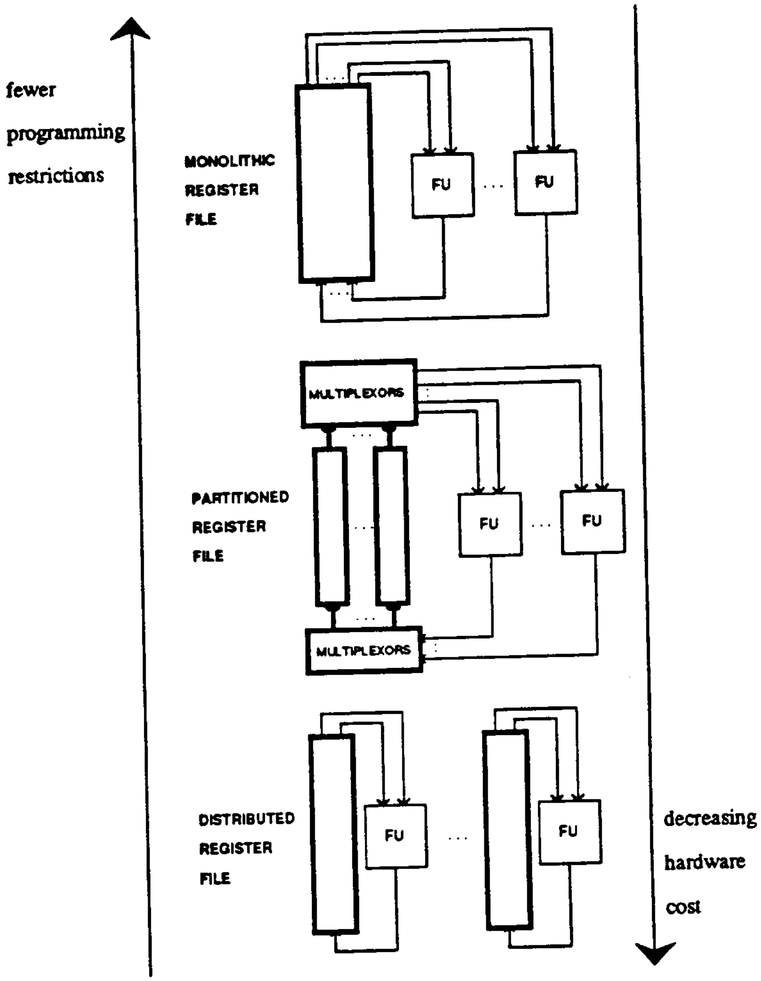


Figure 2.4: Types of Multiported Register Files

This figure shows three different configurations for providing a multiported register file to support multiple functional units. The components that are part of a register file have a bolder outline.

When implementing a register file with a given bandwidth, these configurations represent a tradeoff between decreasing area cost and increasing restrictions on accessibility per clock period. The *monolithic* configuration uses a multiported register cell that increases in size with increasing bandwidth, which means that any register can be accessed, even multiple times, in the same clock period. The *partitioned* configuration uses a dual-ported register cell with one read port and one write port. Although the register banks are fully connected to the functional units by a pair of multiplexors, at most two registers from each register bank can be accessed by any functional unit per clock period. The *distributed* configuration does not use multiplexors and lacks the full connectivity of the other two configurations. Although any register is available for its associated functional unit, an explicit transfer is needed if a different functional unit requires access.

A monolithic register file is used by most superpipelined, superscalar and VLIW designs. For example, the floating-point register file of the IBM RS/6000 has 4 read-ports and 2 write-ports to support a three-input multiply-add unit, a load port and a store port. The SUN SuperSPARC also uses registers with 4 read-ports and 2 write-ports [13]. A register file on the Multiflow Trace handles four reads and four writes at once to support a floating-point multiplier, a floating-point adder and two memory ports [23]. Whereas these register files are similar in organization, the Intel iWarp, in contrast, has a a 17-ported register cell for 128 32-bit words [67].

An alternative configuration for providing a multiported register file is to partition the registers into banks where each register has only one read-port and one write-port — a configuration which I call a *partitioned register file*. This configuration is used in vector architectures, where a register bank is more commonly known as a vector register and is comparable in organization to a simple, scalar register file. Each vector register consists of many dual-ported registers and has its own read bus and write bus. Multiple vector registers, which are considered collectively as a vector register file, give the appearance of a register file with multiple read and write ports. A register file that is partitioned into N banks can allow N accesses to occur in the same clock period. With chaining or tailgating hardware, N reads and N writes can occur simultaneously.

When compared to a monolithic register file, a partitioned one provides less connectivity between any individual register and any functional unit. Despite being partitioned, such a register file has two sets of multiplexors that provide complete flexibility in connecting any register bank with any functional unit. One set of multiplexors connects the register-read buses to the input buses of the functional units, while another set connects the output buses of the functional units to the register-write buses. However, not all registers are available simultaneously as an operand or as a result. Instead, only two registers per vector register are available, one for a read and one for a write, during each clock period. In other words, a vector register can be used concurrently by at most two vector instructions: as the destination for one and the source for the other.

As Figure 2.5 shows, the number of registers provided in vector register files varies greatly, mainly because the number of registers per vector register spans a wide range. Vector processors typically have eight vector registers; eight of the eleven implementations listed use a vector register file that can have eight vector registers. In contrast, the number of registers in one vector register ranges from 4 registers in a Thinking Machine processor node to 2048 registers in an Ardent Titan. The Thinking Machines, Ardent, and Fujitsu processors are different in that the vector register file is reconfigurable: software can partition the register file into any number of vector registers where the number of registers per vector register can be varied as long as the total number of registers remains constant. Reconfigurability is possible because a vector instruction can address individual elements of a vector register.

A VLIW machine, the Cydrome Cydra, also uses a variant of a partitioned register file, called a *multiconnect* in Cydrome terminology [28], which demonstrates that a type of register file is not necessarily confined to a specific architectural class. A multiconnect is slightly different from a vector register file in that the connectivity between registers and functional units in the former is more restrictive for writes and less restrictive for reads. In

VECTOR PROCESSOR	NUMBER OF VECTOR REGISTERS	VECTOR REGISTER LENGTH	TOTAL NUMBER OF REGISTERS	REFERENCE
processor node of Thinking Machines CM-5	4–16	16–4	64	[109]
IBM 3090	8	8–256	64–4096	[16]
NEC VPP ULSI	4(+4)	64(+96)	256(+384)	[90]
Cray-1, Cray-2, Cray X-MP, Cray Y-MP	8	64	512	[110]
Convex C1	8	128	1024	[92]
Fujitsu VPU	8–64	128–16	1024	[64]
DEC VAX 6000	16	64	1024	[102]
NEC SX-2	8(+32)	256	2048(+8192)	[116]
Ardent Titan	4–8192	2048–1	8192	[31]
Fujitsu VP200	8–256	1024–32	8192	[85]
Hitachi S810/20	32	256	8192	[88]

Figure 2.5: Configurations of Vector Register Files for Commercial Processors

This table shows configurations of register files for eleven commercial vector processors, listed in order of total register capacity and minimum number of vector registers. All configurations store 64-bit data.

Some of the implementations have special features. The vector-register length in the IBM 3090 can vary among implementations, providing a range of cost/performance models. The implementations from NEC include a set of vector registers, whose size is enclosed in parentheses, that are not connected directly to the arithmetic functional units and are used mainly for temporary storage. The implementations from Thinking Machines, Ardent, and Fujitsu provide a register file that is reconfigurable in software. The number of vector registers and their lengths can vary in the indicated range as long as the total number of registers remains constant: 64 for the CM-5, 512 for the VLSI implementation by Fujitsu, and 8192 for the Ardent Titan and Fujitsu VP200.

With the exception of the CM-5 processor node, which is part of a massively parallel processor, and the low-end IBM implementations, the total number of registers provided in a vector register file ranges from 256 to 8192, much larger than the typical 32 registers used in today's scalar and superscalar architectures.

a multiconnect, a register bank can only receive results produced by a particular functional unit; hence, the number of register banks, or partitions, is equal to the number of functional units connected to such a register file. In contrast, not only can any register bank deliver an operand to any functional unit, but each register bank can deliver, in the same clock period, operands to all functional units. To provide this functionality, each register bank must have as many read ports as there are inputs to functional units; to implement this in the Cydrome Cydra, a bank uses dual-ported registers that are replicated rather than registers with multiple read-ports. In the Cydra, there are 64 registers per register bank and six functional units, for a total of 384 registers that can store unique data. Because a register bank is replicated to provide increased accessibility to its registers, the physical implementation of this multiconnect requires six times as many registers.

Finally, a configuration different from the monolithic and partitioned ones is a *distributed register file* (although the definition of a register file becomes somewhat vague at this point). In this configuration, the registers are divided into sets where each set is connected to its own functional unit(s). A slight variation of a distributed register file is a *split register file* where each register set has a specific purpose such as an integer register set or a floating-point one [105, 15]. The VLIW architecture, the Multiflow Trace, has a distributed register file where each register set is actually a monolithic one, which was described earlier. A distributed register file is also used in the vector processor node for Thinking Machine's CM-5. The number of register sets can vary from four in the CM-5 processor to 7, 14, or 28 in a Multiflow Trace, depending upon the model.

Unlike the partitioned register file, a distributed one is more restrictive in its connectivity between itself and any functional units. If a value that is stored in one register set is needed for a different group of functional units, the value must first be transferred to the appropriate register set. This transfer is often made through the memory system, making this configuration similar to the separate register files in a multiprocessor. Without a special algorithm for assigning registers, up to 50% of performance could be lost due to excessive data transferring [41].

2.2.3 Multiple Operation Execution

The third task in the basic sequence for carrying out an operation is to actually execute it. Executing multiple operations at the same time can be accomplished with a pipelined functional unit or multiple functional units or both. When multiple functional units are used, they can be general purpose or special purpose (e.g., a floating-point adder) or somewhere in between (e.g., a floating-point unit or an integer unit). The main advantages of a general purpose functional unit are fewer unique components to design and simpler algorithms for scheduling instructions in a compiler. The main advantage of special purpose functional units is a faster implementation of hardware for specific types of operations.

Vector architectures typically implement multiple, fully-pipelined, special purpose functional units. Superpipelined designs also use multiple, special purpose functional units, although these are not always pipelined; the functional units of the CDC 6600 are not pipelined whereas those of the CDC 7600 and the Cray machines are. Superscalar designs, such as the IBM RS/6000, Metaflow Lightning, and Sun SuperSPARC, use a fully-pipelined, floating-point unit and integer unit as well as a special purpose unit for handling branch operations. VLIW designs have used various combinations: the Cydrome Cydra uses special purpose, fully-pipelined functional units; the Multiflow Trace uses several fully-pipelined, floating-point and integer units; and the Intel iWarp uses non-pipelined, special purpose units.

In addition to providing multiple functional units, hardware also provides a mapping between the functionality of a given operation to any functional unit that can provide that functionality. Changing the configuration of functional units merely changes this mapping; this is part of a hardware implementation and is independent from instruction set design. Thus, of the four tasks, this is the most decoupled from the design of an instruction set, and there is no inherent reason why special purpose or general purpose functional units should be used in one architecture and not another.

2.3 Vectorization

In this section, I describe how a vectorizing compiler transforms a source program that is written for a scalar processor into code containing vector instructions. An understanding of this software procedure, known as *vectorization*, is important for hardware designers because vectorization facilitates the effective use of a vector architecture. Consequently, this description is not intended to be a thorough examination of the research issues concerning vectorization but rather a tutorial that outlines the main aspects of vectorization for those who are more familiar with hardware design.

Because vector hardware imposes some restrictions on what can be executed with vector instructions, not all parts of a program can be translated into vectorized code. Using vector hardware as the motivating factor, I first present the properties of a vectorizable program fragment. I next outline how a vectorizing compiler identifies these properties and generates vectorized code. This last part also describes how using vector instructions is conceptually similar to using "loop unrolling", an optimization technique used in scalar compilation.

2.3.1 Properties of A Vectorizable Program Fragment

Not all parts of a program can be translated into vector instructions. In this subsection, I show that a program fragment can be translated into vectorized code if it has the following properties:

1. it is a loop;
2. it has at least one variable, called an *aggregate variable*, that can reference different memory locations (e.g., an array);
3. the memory accesses of each aggregate variable form an arithmetic progression; and
4. any statement that contains an aggregate variable cannot depend on itself.

Because it is a compiler's job to identify these properties, many presentations have been made using a simplified model of vector execution as seen by a compiler. I take a different approach by explaining how a hardware implementation of a vector instruction contributes to these properties.

Other properties, such as the presence of conditional statements and array references with nested indices, do not prevent vectorization as long as the above properties and appropriate vector hardware, such as conditional vector execution and gather/scatter memory, are present. For the interested reader, Hennessy and Patterson describe the necessary hardware for allowing such constructs to be vectorized [92, Section 7.6].

For now, I assume that a vector instruction can execute as many operations as needed. I will remove this simplifying assumption in Section 2.3.2 when I discuss the generation of vectorized code.

Property 1: Loop

The first property of a vectorizable program fragment is that it be a loop, which repeatedly executes the same sequence of statements. This is because the operations performed by a vector instruction are all of the same type. For example, the vector instruction `V0<-M[R1]` loads a vector register with values stored in consecutive memory locations beginning at the address held in register `R1`. A loop statement is analogous to a vector instruction in that successive iterations of a loop statement correspond to successive operations of a vector instruction. For example, in the following loop

```
      DO 10 I=1,N
10       A(I)=A(I)+B(I)
```

the addition in the i^{th} iteration corresponds to the i^{th} operation of a vector add instruction.

Although a loop is certainly a compact expression of such a regular computation, it is possible to express vectorizable computations in a full "unrolled" fashion. For example, the following two program fragments express the same computation:

```
                                    C(1) = A(1) + B(1)
                                    C(2) = A(2) + B(2)
      DO 10 I = 1,5          <=>    C(3) = A(3) + B(3)
10       C(I) = A(I) + B(I)         C(4) = A(4) + B(4)
                                    C(5) = A(5) + B(5)
```

Today's vectorizing compilers will translate the loop on the left into vector instructions but not the straight-line code on the right. Determining that the five individual statements are somehow related requires techniques substantially different from those used in current vectorizing compilers. The situation is complicated by the possibility that these statements could be interspersed among other, less related statements.

Property 2: Aggregate Variables

The second property of a vectorizable program fragment is that it must contain at least one variable that can reference a different location on each iteration. This is because each operation in a vector instruction accesses a different storage location. I call such a variable an *aggregate variable* because it may reference a number of different storage locations. For instance, in the following loop:

```
        DO 20 I=1,N
          A=A+21
20        B(I)=B(I)+21
```

the array reference B(I) is an aggregate variable whereas the scalar variable A is not. Other examples are pointers and procedure parameters. Just as a vector instruction is used to execute a statement in a loop, a vector register stores the values of an aggregate variable. For example, the i^{th} element of the array B(I) is stored in the i^{th} register of a vector register.

Property 3: Arithmetic Progression for Memory-access Pattern

The third property of a vectorizable statement is that the memory accesses of each aggregate variable form an arithmetic progression [101]. This is because the vector address-generator is only capable of addition. As a counterexample, the access pattern for B(J) in the following loop

```
        DO 10 I=1,N
           J=J*2
10         A(I)=A(I)+B(J)
```

is a geometric progression requiring multiplication for successive addresses. This property is not all that restrictive because array references in most programs proceed in an arithmetic sequence.

This property does not completely describe all aggregate variables that can be vectorized. For example, the access pattern of an array reference with a nested index, such as A(B(I)), can be completely random. However, because calculating the addresses uses only addition and the B addresses still form an arithmetic progression, this reference can be vectorized if gather/scatter hardware is provided.

Property 4: No Self-dependent Statements

Finally, the fourth property of a vectorizable loop is that any statement that contains an aggregate variable cannot depend on itself. Although it is not immediately obvious, the reason for this property is that vector instructions do not interleave the execution of their operations. For example, two vector instructions that use the same functional unit execute serially rather than alternating the execution of their operations. At the beginning of this chapter (in Section 2.1), I stated that any architecture that supports fine-grain parallelism must deal with data dependences. This fourth property shows the relationship between dependences and a vector architecture.

Figure 2.6 shows how a vectorizable statement can directly or indirectly depend on itself.[3] Direct self-dependence is often simple enough to vectorize with either special hardware or a major software transformation. In addition, there are other special cases of self-dependence that can be vectorized; I will give more details about these shortly. More

[3] Statements involving only scalar variables, such as A=A+5, are also self-dependent. However, because scalar self-dependences do not prevent vectorization, I exclude them from this discussion. For the sake of brevity, I use the term *self-dependent statement* to mean one that involves aggregate variables.

generally, an indirect self-dependence specifies an execution order that vector instructions cannot satisfy but that statements of a loop must execute to preserve data dependences for correct functionality.

To explain why self-dependent statements prevent vectorization I will first describe the execution order that is specified by a self-dependent statement. To better visualize this, I make a distinction between a *statement* and an *instance* of a statement; one iteration of a loop contains one instance of each statement. For example, for the following loop

```
      DO 10 I=1,N
 5       A(I)=B(I)*C(I)
10       B(I+2)=A(I)+T
```

A(2)=B(2)*C(2) is the second instance of the statement labeled 5 (S_5). The execution order specified by a self-dependent statement is dictated by the order in which accesses to the same memory location occur. For instance, for the above loop, the following table summarizes the order of accesses to the two memory locations A(1) and B(3):

MEMORY LOCATION	FIRST REFERENCE	SECOND REFERENCE
A(1)	first instance of S_5	first instance of S_{10}
B(3)	first instance of S_{10}	third instance of S_5

In order to preserve the order of these accesses, the first instance of statement 5 must execute before the first instance of statement 10, which in turn must execute before the third instance of statement 5. In general, to preserve the dependences involving two self-dependent statements, S_v and S_w, the instances of each statement must be executed in an interleaved fashion. In other words, some instance of S_v must execute before some instance of S_w, and vice versa.

However, using vector instructions to execute self-dependent statements in a loop does not produce the necessary interleaved pattern. This is easiest to see when vector instructions execute serially. Although, in reality, this is not often the case (otherwise little parallelism would occur), two vector instructions will execute serially if they use the same functional unit. Figure 2.7 shows that, because a vector instruction is analogous to a vectorizable statement and operations of a vector instruction correspond to instances of a statement, the serial execution of two vector instructions is conceptually equivalent to all instances of a statement being executed before all instances of another statement. However, a self-dependent statement requires that instances of different statements execute in an interleaved fashion to ensure correct functionality. As a result, vector instructions cannot be used to execute any statement that indirectly depends on itself.

As mentioned at the beginning of this discussion on self-dependence, a statement that directly depends upon itself is often a special case that can be vectorized. For instance, the following loop

```
      DO 10 I=1,N
10       A=A+B(I)
```

is an example of a *scalar reduction*, which is a function that reduces a vector of data to a scalar value; other examples of scalar reductions are minimum and maximum operations.

NO SELF-DEPENDENT STATEMENTS	DIRECTLY SELF-DEPENDENT STATEMENT	INDIRECTLY SELF-DEPENDENT STATEMENTS
DO 10 I=1,N 8 A(I+1)=B(I)+X(I) 9 B(I)=A(I)*S 10 C(I)=C(I)*Y(I)	DO 20 I=2,N 20 A(I)=A(I-1)+X(I)	DO 30 I=2,N 28 A(I)=B(I)+X(I) 29 C(I)=A(I)*S 30 B(I+1)=C(I)*Y(I)

Figure 2.6: Self-dependent Statements

These loops contain examples of self-dependent statements. Recall from Section 2.1 that a dependence specifies the order in which two references that access the same memory location must execute in order to guarantee correctness. To more easily recognize when self-dependence occurs, a dependence is associated with the statements that contain the two references. It is also necessary to distinguish between *instances* of a statement; one iteration of a loop contains one instance of each statement in the loop. A statement is *self-dependent* if instances of that statement must be executed in succession for the associated program to execute correctly. Moreover, a statement can be either directly or indirectly self-dependent.

A statement S_w *directly depends* on another statement S_v if S_w and S_v contain references that access a common storage location and S_w accesses the location after S_v does. For example, in the loop labeled 10, statement 9 directly depends on statement 8 because statement 9 accesses A(2) after statement 8 does. This order of access, in fact, is true for all elements of the arrays A and B. A statement S_w is *directly self-dependent* if it contains references that access the same location but in different iterations because in order to maintain the order specified by the dependences among these references, the instances of S_w must execute in succession. For example, the statement in the loop labeled 20 is self-dependent because the two references A(I) and A(I-1) access the same memory locations, albeit in different iterations. On the other hand, statement 10 in the loop labeled 10 is not self-dependent even though it contains two references to C(I).

Dependence and self-dependence are also transitive. A statement S_w *indirectly depends* on another statement S_v if S_w depends, directly or indirectly, on a statement which in turn directly or indirectly depends on S_v. For example, in the loop labeled 30, statement 30 indirectly depends on statement 28 because statement 30 directly depends on statement 29 because of the references to C(I), and because statement 29, in turn, depends on statement 28 because of the references to A(I). Note that indirectly dependent statements do not necessarily access the same memory location. A statement S_w is *indirectly self-dependent* if S_w depends, directly or indirectly, on another statement S_v which in turn directly or indirectly depends on S_w. Moreover, any statement, such as S_v, that is part of these indirect dependences is also indirectly self-dependent. For example, in the loop labeled 30, statement 28 indirectly depends on itself because it directly depends on statement 30 because of the references B(I+1) and B(I), and because statement 30 indirectly depends on statement 28. In fact, each statement in this loop is self-dependent.

The presence of a self-dependence is signaled by a dependence whose references occur in different iterations. Such a dependence is called a *loop-carried* one. In contrast, a dependence whose references occur in the same iteration is called *loop-independent*. Although a self-dependence contains at least one loop-carried dependence, not all loop-carried dependences are part of a self-dependence. For example, the loop labeled 10 has a loop-carried dependence but no self-dependent statements.

	EXECUTION ORDER	
SOURCE CODE	SCALAR VERSION	VECTOR VERSION
DO 10 I=1,N A(I)=B(I)+C(I) 10 B(I)=A(I)+D	A(1)=B(1)+C(1) B(1)=A(1)+D A(2)=B(2)+C(2) B(2)=A(2)+D A(3)=B(3)+C(3) B(3)=A(3)+D ⋮	A(1)=B(1)+C(1) A(2)=B(2)+C(2) A(3)=B(3)+C(3) ⋮ B(1)=A(1)+D B(2)=A(2)+D B(3)=A(3)+D ⋮

Figure 2.7: Execution Orders When Using Scalar and Vector Instructions

This figure illustrates how the order in which instances of statements in a loop are executed is different when using vector instructions instead of scalar instructions. Scalar code executes one iteration of a loop — i.e., one instance of each statement in the loop — before executing the next iteration (ignoring scalar optimization techniques such as loop unrolling or software pipelining). The difference in execution order when using vector instructions is easiest to see when two vector instructions use the same functional unit. All operations of a vector instruction will initiate before any operations of the other instruction are initiated. Because successive operations of a vector instruction correspond to successive iterations of a loop, vector code conceptually executes all instances of a statement before executing all instances of the next statement.

Either hardware or software techniques can be used to vectorize such functions. As an example of the former, the IBM System/370 vector architecture includes the reduction operations *accumulate, minimum,* and *maximum* in its instruction set [16]. The Cray-1 also provides some hardware support for computing scalar reductions as an unexpected benefit of using single-ported vector registers [65]. Alternatively, a scalar reduction can be vectorized using an algebraic transformation in software without any special hardware support [92, page 382]. For example, a summation can be vectorized as follows (The notation for vector instructions is explained in Figure 2.2 on page 10.):

```
                                  R1 <- address of A(1)
                                  R2 <- address of A(1) + N
      DO 10 I=1,N          ⟹   TOP: V1 <- M[R1]
 10      A=A+B(I)                 V0 <- V0 + V1
                                  R1 <- R1 + 8
                                  BRNE  R1,R2,TOP
```

The arithmetic laws of commutativity and associativity are used to separate the sum into n partial sums that can be vectorized, where $n \ll$ N. Because floating-point arithmetic is not associative, this transformation can compute a value that is different from what the original code computes. The transformed code on the right-hand side is the vectorized version where the vector length is 8, the vector register V0 contains the $n = 8$ partial sums, and N is assumed to be a multiple of 8. The final sum is obtained by using scalar code to add the partial sums that are stored in V0.

Another example of a direct self-dependence is a *first-order linear recurrence*:

```
      DO 20 I=1,N
 20      A(I)=A(I-1)*B(I) + C(I)
```

This function is given its name because the expression on the right-hand side is a linear function that uses a value from the previous iteration. By extension, an n^{th}-order linear recurrence uses a value from the n^{th} previous iteration. Again, either hardware or software techniques can be used to vectorize such functions. The Hitachi S-810 has a special macro-vector instruction called VITR that executes first-order linear recurrences [114]. Alternatively, an algebraic transformation similar to the one for scalar reductions can vectorize such functions using basic pair-wise vector instructions [103].

In addition to the above self-dependences, there are other special ones that can be vectorized. For example, the following anti-dependence

```
      DO 10 I=1,N
 10      A(I)=A(I+1)*B(I)
```

forms a self-dependence that can be vectorized because, on most modern vector implementations, the fetch of the A elements will occur before the store. Another self-dependence, consisting of both a flow dependence and an anti-dependence on the B elements, can be vectorized by separating the loop into two parts, one with only the flow dependence and the other with only the anti-dependence:

```
                                        DO 5 I=1,N/2
                                           A(I)=B(N-I+1)+C(I)
     DO 10 I=1,N                       5   B(I)=D(I)*T
        A(I)=B(N-I+1)+C(I)     ⟹         DO 10 I=N/2+1,N
  10    B(I)=D(I)*T                        A(I)=B(N-I+1)+C(I)
                                      10   B(I)=D(I)*T
```

As a last example, a major software transformation vectorizes array references with nested indices:

```
      DO 10 I=1,N
10       A( K(I) ) = A( K(I) ) + B(I)
```

Because the index values for the A references are not known at compilation time, this statement must be assumed to be self-dependent. Nonetheless, using gather/scatter memory instructions, the Cray Research compiler, *cft77 version 5.0*, is able to vectorize such a loop and still preserve any dependences that may exist. Michael Wolfe describes other techniques for vectorizing in the presence of certain self-dependences [120, pages 64–67 of Chapter 3].

In summary, although there are special instances of self-dependences that can be vectorized, a self-dependent statement normally prevents vectorization because the dependences force instances of statements from different iterations to execute in an interleaved fashion. The absence of self-dependent statements imposes no restrictions on the order in which statements from different iterations can execute; only those from the same iteration must execute according to a partial ordering based on intra-iteration dependences. This results in statements that can be ordered in such a way that all dependences will be preserved when vector instructions are used.

2.3.2 Generating Vectorized Code

Whereas vector hardware determines the properties of a vectorizable program fragment, a vectorizing compiler is responsible for identifying these properties and then generating the appropriate mixture of vector and scalar instructions that will execute a vectorizable program fragment. Using this mixture of vector and scalar instructions to execute a loop is conceptually similar to using *loop unrolling*, an optimization technique used in scalar compilation. To highlight this similarity, I outline in the following paragraphs how a vectorizing compiler identifies a vectorizable program fragment and generates vectorized code.

For the following discussion, I distinguish between vectorizable and non-vectorizable operations: the former are translated into vector instructions and the latter into scalar instructions. In a vectorizable loop, operations that either have an aggregate variable as input or generate an aggregate variable as output are vectorizable operations; all other operations are non-vectorizable ones. For example, the loads, multiplication, and store in the statement A(I)=B(I)*C(I) are vectorizable operations whereas the three additions for address calculations are non-vectorizable ones. Other examples of non-vectorizable operations include loop-index calculations, branch comparisons, and explicit, scalar self-dependences such as X=X+21.

Identifying a vectorizable program fragment is a simple matter of identifying a program fragment that has the four properties described in the previous subsection. The first three properties are easily identified. Loops can be identified by the semantics of a language. For example, in FORTRAN, the DO construct signifies a loop whereas in C, the for and while constructs do. Because other constructs, such as if ... goto, can also form a loop, a more semantic-independent methodology, that is based on *flow graphs*, can be used to identify loops. A flow graph, which is normally constructed for scalar optimization, is a directed graph that represents the control flow of a program; a loop is merely a sub-graph with a special structure within a flow graph. Aho, Sethi, and Ullman give algorithms for constructing flow graphs and identifying loops in them [2, Chapter 10]. The second property, the presence of aggregate variables, is easily determined by examining the type of a variable; that is, whether a variable is a scalar, an array, or a pointer. The third property, memory accesses that form an arithmetic sequence, can be determined by examining the *use-def chains*, which are built for scalar analysis [4, 2], to identify variables (typically indices of arrays) that are used to compute the addresses of an aggregate variable and check that their computations involve only additions or subtractions. For the purposes of code generation, this analysis can also extract more information, such as the value of memory offsets, about the access pattern of an aggregate variable.

Identifying the fourth property, the absence of self-dependent statements, proceeds in two steps. The first is to construct a *dependence graph*, which is a directed graph that represents the dependence relations of a program. In such a graph, a vertex is a statement or operation depending upon the level of detail desired, and there is an edge from one statement to another if the second statement accesses a common memory location after the first statement does. Because a dependence graph is used to identify vectorizable loops, only vectorizable statements or operations are represented in the dependence graph for a vectorizing compiler. For example, in the following loop

```
       DO 10 I=1,N
    5     B=B+21
   10     A(I)=A(I)*8
```

statement S_{10} would be included in the dependence graph but statement S_5 would not. Similarly, the operations for loop overhead and address calculation would not be part of the graph.

Constructing a dependence graph, or more specifically, identifying dependences that exist among vectorizable statements, is not trivial. Unlike the other functions I have described above for a vectorizing compiler, identifying dependences, a process also known as *dependence analysis*, is necessary for any compiler that is generating parallel code because knowing where the dependences are is the key to correct functionality. Consequently, much research, past and present, has been spent in this area not only for vectorizing compilers but also for more general, parallelizing compilers. There are two major methods for detecting dependences. The more established method operates only on arrays. Because an array has an explicit addressing mechanism, detecting a dependence between two statements is a matter of solving an algebraic equation constructed from the address functions of the associated array references [120, 5, 83]. A newer, graph-theoretic technique, which is based on data-flow analysis, is aimed at linked-list data structures that use pointers with implicit addresses [79, 56]. Because the early vector implementations were used for scientific programs that modeled physical phenomena in a discrete fashion that is a good match for the array data structure, vectorizing compilers typically use the algebraic method.

Two critical aspects of any dependence analyzer are efficiency and accuracy. Efficiency is important for practical reasons whereas accuracy is critical because any dependence that really exists must be identified. If there is doubt about the presence of a dependence, the dependence is assumed to exist; otherwise the program will not function correctly. Identifying too many "false" dependences, however, will reduce the opportunities for parallelism. Array-based dependence analysis tends to be more accurate than dependence analysis for linked lists due to the array's more explicit mechanism for naming memory locations. Furthermore, although efficiency is often traded for accuracy, Maydan, Hennessy, and Lam have shown that a suite of selected algorithms for array-based dependence analysis can accurately identify all dependences in the PERFECT Club benchmark set of 13 scientific programs and add only an average of about 3% to the compilation time [83].

Once a dependence graph is constructed, the second step to identifying self-dependent statements is to determine whether the graph contains a directed path from any statement to itself. Any statement on such a path is a self-dependent one. Such a path is called a *dependence cycle* because it forms a directed cycle in a dependence graph, and the statements on that path are said to form a dependence cycle. Because a vectorizable loop is one whose dependence graph has no directed cycles, its corresponding dependence graph is also known as a *dag*, an acronym for *d*irected *a*cyclic *g*raph. When compared with constructing a dependence graph, finding cycles in a dependence graph is relatively easy. Wolfe describes an algorithm for doing so that is attributable to Tarjan [120, page 57].

After identifying a vectorizable loop, a vectorizing compiler schedules vectorizable operations and assigns registers. Scheduling determines an execution order that preserves dependences among the operations whereas assigning determines which register stores the value produced by an operation. Both these tasks use a loop's dependence graph to represent the functionality of the vectorizable portion of a loop. Although these tasks are also performed for non-vectorizable operations, more emphasis is placed on the vector aspect because execution of vectorizable operations will dominate the execution time of a loop. These tasks are important not for correct functionality but more for performance reasons. Because the performance impact of instruction scheduling and register assignment is a central theme of my dissertation, details about each task are provided in upcoming chapters. For now, I will simply discuss some issues concerning the order in which these two tasks are executed.

Scheduling and assignment can be done in either sequence, both of which present potential compromises to performance. Scheduling first with disregard for register usage can lead to execution orders whose register requirements exceed the physical limitations of the processor. To accommodate such a schedule, extra instructions are introduced to spill values out to memory. In an architecture that supports fine-grain parallelism, such register spilling may not reduce performance because the execution of these extra instructions can be potentially overlapped with that of the original instructions. On the other hand, assigning can be done first using the execution order taken directly from the source code, thereby providing more control over register usage. However, this introduces extra register dependences that the scheduler must now preserve, and more dependences reduce the opportunities to schedule for parallelism. Reassignment of the registers will help alleviate this restriction. In the Cray Research compiler, scheduling is done before register assignment but the scheduler attempts to minimize register usage.

Although these two tasks traditionally have been performed separately, research in scalar algorithms are examining techniques that allow for interaction between these functions [50, 15]. Based on information from the register assigner, the scheduler alternates between two scheduling strategies. The initial goal of the scheduling algorithm is to minimize execution time. To avoid exceeding the register limitation of the processor, the register assigner is used to inform the scheduler about the register usage of its schedule while the schedule is being constructed. Once a threshold has been exceeded, the scheduler changes strategies to reduce register usage until registers are no longer a critical resource, at which point the scheduler switches back to the initial strategy. Such a technique is being adapted in future versions of the Cray Research compiler.

In addition to scheduling and assigning, a vectorizing compiler translates operations into vector and scalar instructions. However, whereas the translation from a vectorizable operation to a vector instruction is straightforward, the translation from a non-vectorizable operation to a scalar instruction requires some explanation. First, it should be noted that a vectorizable operation produces a different result every iteration and is translated into a vector instruction that executes n instances of that operation. As with a vectorizable operation, a non-vectorizable one that is self-dependent, such as `A=A+1` and `X=X+21`, produces a different result every iteration and is translated into one or more scalar instructions that emulate the execution of n iterations of that self-dependence. The execution of n iterations of a scalar self-dependence can be expressed as a function of n that requires fewer instructions to execute than n instances of the corresponding scalar instruction. For example, the scalar instruction for executing the operation `A=A+1` n times is just `R1<-R1+R0`, where register `R1` holds the value of `A` and register `R0` holds the value n. A slightly more complex operation, such as `X=X+21`, requires two scalar instructions to emulate executing it n times: `R2<-21*R0` and `R1<-R1+R2`.

A technique called *stripmining* generates code to execute a loop in which the number of iterations executed exceeds the number of registers in a vector register. To reduce the number of parameters in the following discussion, I assume the number of registers in a vector register to be 64 (based on the Cray Y-MP). Stripmining executes such a loop in strips where each strip executes 64 or fewer iterations. When the number of iterations N is not a multiple of 64, one of the strips executes N *mod* 64 iterations.

The easiest type of loop to stripmine is one in which the value of N is known without having to execute the loop, such as:

```
      DO 10 I=1,N
10       A(I)=0
```

There are two ways to implement the code that controls the execution of the strips for such a loop. In a software-oriented approach, which is used in the Cray vector implementations, the strip that executes N *mod* 64 iterations is executed first:

```
     < scalar code to calculate N mod 64 >
     VL <- 64
     V0 <- 0
     VL <- N mod 64  ; assign length of first strip
     R1 <- address of A(1)
     R2 <- N
TOP: M[R1] <- V0     ; store 0 into elements of array A
     R1 <- R1+VL     ; update address
     R2 <- R2-VL     ; update branch counter
     VL <- 64        ; update strip length
     BRNZ R2,TOP
```

Because 64 is a power of 2, the value *N mod* 64 can be computed by using simple shift and mask operations and no divide or remainder calculations. Alternatively, for a hardware-only method, which is used in the IBM System/370 vector architecture [16], the strip that executes *N mod* 64 iterations is executed last. A special instruction, VLVCU (Load Vector Count and Update), first subtracts the number of iterations executed for a strip from a register that holds the number of remaining iterations to be executed, and then it sets the condition code to indicate whether or not the difference is greater than or equal to zero. Because the number 64, which is the number of registers in a vector register, does not need to appear in the generated code, implementing stripmining in this fashion allows processors with different vector-register lengths to be binary compatible.

Stripmining can be used even when the value of *N* can be determined only by executing the loop. For example, because there are no self-dependent statements, the following loop can be vectorized despite the potential early exit:

```
   DO 10 I=1,N
      A(I) = B(I)*C(I) + 4/D(I)
      if ( A(I).eq.S(I) ) goto 20
10 CONTINUE
20 ...
```

However, the resultant stripmined code becomes more complex and involves the possible execution of unnecessary operations. Because of the complexity and variability in performance, few vectorizing compilers attempt to vectorize such loops. For the interested reader, Wolfe describes techniques for vectorizing such loops [120, Chapter 10].

Stripmining is conceptually similar to the scalar optimization technique called *loop unrolling* whereby the body of a loop is replicated *n* times and *n* is called the amount of unrolling. A strip that executes *n* iterations of a loop is analogous to a loop body that has been unrolled *n* times. One of the benefits of loop unrolling is that more independent operations are available to execute in parallel. However, in order to correctly execute these operations, accurate information about data dependence is needed as is the case for vectorization and stripmining. Moreover, optimizations applied to an unrolled loop body to eliminate redundant computations should result in code similar to that produced for scalar instructions in a stripmined loop. Based on this analogy, the hardware-only implementation of stripmining in the IBM System/370 vector architecture can be considered hardware support for loop unrolling where the amount of unrolling is set by the hardware.

Despite the similarities between stripmining and loop unrolling, the former has advantages over the latter. For example, other than having to calculate *N mod* 64, no extra code is needed to execute the *N mod* 64 iterations because the same code is used to execute strips of any length. Moreover, this code is reasonably optimized for most amounts of unrolling. In contrast, the extra code required to execute the *N mod* 64 iterations for loop unrolling can be either a non-optimal, rolled version of the loop or optimized, unrolled versions for each residual value that is possible. Another advantage to stripmining is that the number of instructions generated does not increase substantially over the number generated for the rolled version because vector instructions are used. Hence, stripmining and the use of vector instructions avoids the higher instruction-bandwidth incurred by loop unrolling while taking advantage of the same fine-grain parallelism that loop unrolling uses.

2.4 Summary

In this chapter, I described how vector architectures support fine-grain parallelism in both hardware and software, and I contrasted this architectural approach with three others: superpipelined, superscalar, and VLIW. What follows is a summary of those aspects that are common to any architecture that supports fine-grain parallelism and those that are specific to a vector architecture.

A program's functionality must not change even though the operations of a program are performed in a different order when using parallel instead of scalar execution. A program's functionality, which is reflected by its output, will not change if the order of accesses to each storage location does not change. Because data dependences stipulate the order of accesses, preserving data dependences is the key to guaranteeing correct functionality. As a result, any architecture that supports fine-grain parallelism must provide a mechanism for handling data dependences to ensure correct functionality.

Dependences can occur in two different storage locations: registers or main memory. Hardware mechanisms are typically used for resolving register dependences, examples of which are register-renaming and data-forwarding in superscalar and superpipelined architectures, and chaining and tailgating in vector architectures. In contrast, the compiler is usually responsible for handling memory dependences. Because it is responsible for ensuring correct functionality, a crucial component of such a compiler is its dependence analyzer that identifies dependent operations. There are two major methods for detecting dependences in software. The more established method operates only on arrays which have an explicit addressing mechanism. Dependences are detected by solving an algebraic equation constructed from the address functions of array references.

In addition to handling dependences, any architecture that executes more than one operation per clock period must be able to perform multiple instances of the basic execution sequence: initiate operation, fetch operand(s), execute operation, and store result. Whereas there are many techniques for performing multiple instances of each of these tasks, only the first is specifically associated with a particular architectural approach: for example, superpipelined architectures use longer pipelines to produce a faster clock; superscalar and VLIW architectures use the obvious approach of simultaneously issuing multiple operations from the instruction unit; and vector architectures use overlapped execution of multiple vector instructions. The second and fourth tasks, fetching and storing multiple values in the same clock period, can be accomplished by several register file organizations: monolithic, partitioned, distributed, and combinations thereof. These organizations vary in their degree of connectivity to the functional units, which creates a trade off between accessibility and hardware costs. Although there is no hard and fast rule for using a particular organization with a specific architecture, superscalar architectures typically use a monolithic configuration, vector architectures use a partitioned one, and VLIW architectures use combinations of these organizations. Finally, all of the architectures perform the third task, executing multiple operations, in a uniform fashion by using several functional units.

In fact, because only techniques for accomplishing the first task are associated with a particular architectural class and techniques for accomplishing the remaining tasks could be used by any architecture, the defining element of an architecture that can execute more than one operation per clock period is how it *initiates* more than one operation per clock period. For a vector architecture, the key characteristic is the *vector* instruction, one that causes multiple operations to execute sequentially in the same functional unit. This instruction provides a simple intuitive model for understanding how fine-grain parallelism can be used by hardware and by a compiler.

Whereas all four architectural approaches are attempting to use the parallelism present in a program, a vector architecture does so through the use of its vector instructions. Fine-grain parallelism exists in two places: among operations from different basic blocks and among operations within the same basic block. Parallelism across different basic blocks is used by the vector instruction itself, which is essentially a compact form of loop unrolling. This type of parallelism is also manifested when a vector instruction continues to initiate operations after the next basic block begins executing. Such a situation will occur as long as completely independent resources are used. Parallelism within the same basic block is made use of through the overlapped execution of several vector instructions.

Although vector instructions use the fine-grain parallelism present in a program, not all parts of a program can be executed with vector instructions because of how vector instructions are implemented. A program fragment with the properties listed in Figure 2.8 can be vectorized. The presence of either non-aggregate variables or scalar self-dependences does not prevent vectorization. A common example of a non-vectorizable loop is one that performs pointer-chasing through a linked list; such a loop cannot be vectorized because the statement that performs the pointer-chasing (`p=p->next`) contains an aggregate variable and directly depends upon itself. A technique, which is conceptually similar to loop unrolling and is called stripmining, is used to execute a vectorizable loop. Moreover, operations that either have an aggregate variable as input or generate an aggregate variable as output are translated into vector instructions; other operations, such as address calculation or branch computation, are translated into scalar instructions in a manner similar to how this translation is done when loop unrolling.

The properties of a vectorizable program fragment do not include any characteristic of a loop's branch computation, which determines the number of iterations, or *loop length*, which are executed for a particular invocation of a loop. Although it may seem that knowing the length of a loop without executing it is necessary for vectorization, this is not true. The easiest type of loop to vectorize is certainly one whose length can be determined in

CHARACTERISTIC OF VECTOR HARDWARE	PROPERTY OF A VECTORIZABLE PROGRAM FRAGMENT
a vector instruction specifies only one type of operation	a program fragment is a loop in which statements are executed multiple times
each operation in a vector instruction accesses a different storage location	a loop has at least one variable, called an aggregate variable, that accesses a different storage location on each iteration
a vector address-generator can only do addition	the memory accesses of each aggregate variable form an arithmetic progression, which involves only addition
vector instructions do not interleave the execution of their operations	any statement containing an aggregate variable cannot depend on itself

Figure 2.8: Properties of a Vectorizable Program Fragment

This table summarizes the relationship between a characteristic of vector hardware and the corresponding property of a vectorizable program fragment. Details explaining these relationships can be found in Section 2.3.1.

advance. An example of such a loop is a FORTRAN `DO` loop. But a loop whose length cannot be determined in advance will not prevent vectorization as long as the properties mentioned above are present, although the resultant stripmined code will be more complex. Examples are a loop with a conditional exit and a loop with the branch computation `while (X(I)<Y(I))`.

Deriving the properties of a vectorizable program fragment from hardware characteristics illustrates the restrictions imposed by vector hardware rather than those caused by inadequate compilation technology. An example of a restriction imposed by vector hardware is that vectorizable loops cannot contain any self-dependent statements because the execution of operations from different vector instructions cannot be interleaved. If this hardware restriction were removed, as suggested by Chieuh [20], the absence of self-dependent statements would no longer be a requirement for a vectorizable program fragment.

On the other hand, restrictions imposed by inadequate compilation technology can be removed without having to alter hardware. In particular, the accuracy of the dependence analyzer in a vectorizing compiler greatly affects what is vectorizable and what is not from the viewpoint of a compiler. Examples of vectorizable loops that could be executed using vector instructions are an unrolled version of a vectorizable loop and a vectorizable loop with an undeterminable loop length. A specific example of the latter type of loop is one that accounts for more than 90% of the time it takes to execute *eqntott*, a program in the SPEC benchmark suite [106]. Current vectorizing compilers do not vectorize this loop because the program is written in C and the loop has multiple exits. However, compilation improvements incorporated in the the next version of the C compiler from Cray Research will be able to vectorize this loop [62].

In addition to describing the fundamentals of a vector architecture, the contents of this chapter also serve as background material for later chapters. In Chapter 3, *A Case for Vector Architectures*, I examine in greater detail the hardware support for fine-grain parallelism and compare the techniques used by vector and superscalar architectures. Although a vector architecture can be used on only specific parts of a program, I also present data to show that this apparent restriction is minimal. In Chapters 5 and 6, I explore the interaction between vector hardware and the code-generation algorithms in a compiler. In Chapter 5, *Register Usage and Instruction Scheduling*, I improve upon the scheduling algorithm used by the Cray compiler to effectively use more than the eight vector registers currently provided in the Cray Y-MP. And finally, in Chapter 6, *Bus Usage and Register Assignment*, I develop an assignment algorithm for a special configuration of a vector register file that reduces the hardware cost of adding more registers but at the expense of more restrictive accessibility. In the latter two chapters, I also evaluate how effective my algorithms are at using the hardware.

Chapter 3

A Case for Vector Architectures

VLSI technology continues to improve at a phenomenal rate. The combination of decreasing area for one transistor and increasing area for an entire chip has resulted in a proliferation of transistors in a single chip. Since the introduction of the first microprocessor, the Intel 4004 with 2300 transistors, in 1971, the number of transistors per processor chip has consistently doubled every two years resulting in an average yearly growth-rate of about 1.4 [47]. In 1986, Myers, Yu and House predicted that a VLSI chip that has 10 million transistors could be manufactured in 1995 [86]. As evidence that this prediction is well within reason, the following table demonstrates the rapid growth in the transistor count of CMOS processors since Myers et al. made their prediction:

YEAR	PROCESSOR NAME	NUMBER OF TRANSISTORS	REFERENCE
1985	Intel i80386	275,000	[86]
1989	Intel i860	1,000,000	[73, 93]
1991	Intel i860 XP	2,500,000	[61]
	Sun SuperSPARC	3,100,000	

As shown by the transistor counts for 1989 and 1991, this growth rate is expected to increase to 1.5 as memory, which is denser than processor logic, is integrated with the processor. Based on these growth trends, producing a ten-million transistor chip should be feasible by 1995.

Such a large VLSI processor can be used in at least two types of computers. One type of computer is the workstation, a high-performance desktop or deskside computer with a graphical user interface. The workstation is targeted for scientific and engineering applications which is reflected by the fact that 6 of the 10 programs in the SPEC benchmark suite come from this application domain [106]. The purpose of the SPEC benchmark suite is to chronicle the performance of workstations. Programs for user-interface graphics and for scientific/engineering can be characterized as being compute-intensive with a high demand for memory bandwidth.

Another computer in which a large VLSI processor is advantageous is one in which 100 or even 1000 or more processors are connected. Such a computer, called a *massively parallel processor* or MPP, is designed to meet a hardware challenge of the 1990's: to build a computer system that is capable of executing 10^{12} floating-point operations per second or 1 TFLOPS. Because of physical limitations, computer architects generally agree that the only way to achieve 1 TFLOPS performance is through massive parallelism. One important aspect in the design of an MPP is its processor architecture. A more powerful processor reduces the number of processors needed to achieve teraflop performance, thus trading increased design complexity in the processor for decreased design complexity in the interprocessor communication network. In addition, fewer processors means shorter communication latencies, which reduces the burden on the software to schedule around communication delays.

As VLSI technology continues to improve, processor architects are considering designs that support fine-grain parallelism because such designs can easily use the increased hardware resources for continued gains in performance. Of the architectures that support fine-grain parallelism, VLSI designers appear, based on current design trends for microprocessors [58, 61], to prefer a superscalar architecture for the processor of both types of computers. This preference is based on the belief that such an architecture is more cost-effective than a vector architecture, which many designers mistakenly believe is inherently costly to implement and has limited application.

This misjudgement is due partly to the fact that the majority of commercially successful vector implementations have been at the high end of the cost range, the classical example being the Cray Y-MP. Some of this cost is attributable to using expensive technology for fabrication and exotic techniques for packaging and cooling. Nonetheless, a historical correlation between high cost and a vector architecture does not imply that a vector architecture could not be implemented using technology that is less costly. Moreover, although vector architectures have typically been used in multiprocessor computers with less than 20 processors, there is nothing inherent in a vector architecture that prevents it from being implemented in VLSI and consequently used in an MPP. In fact, three companies have recently implemented a vector architecture in VLSI:

- Thinking Machines Corporation has extended the SPARC architecture with a vector architecture for use in their MPP, the Connection Machine 5 [109];

- NEC has implemented a single-chip vector processor with 693,000 transistors in 0.8 μm BiCMOS technology using a clock frequency of 100 Mhz [90]; and

- Fujitsu has implemented a single-chip vector processor with 1.5 million transistors in 0.5 μm CMOS technology using a clock frequency of 70 Mhz [64].

Contrary to the beliefs of proponents of superscalar architectures, I believe that a vector architecture combined with superpipelined hardware would be a better cost-effective choice than a superscalar one for use in either a workstation or an MPP. Although the VLSI implementations of vector architectures are examples of practical feasibility, in this chapter, I explain *why* a vector architecture is a better choice. I first examine the hardware cost of a vector implementation, showing that the cost is comparable to or, in some aspects, even less costly than that of a superscalar implementation. I next address the criticism that a vector architecture is only effective for a limited number of programs showing that, in fact, a vector architecture can effectively use parallelism when it exists in abundancy and that, to mitigate the effects of Amdahl's Law, a combined vector and superpipelined architecture can take advantage of what little parallelism there is in non-vectorizable programs. Finally, I discuss some of the software advantages of a vector architecture.

3.1 Hardware Advantages of Vector Architectures

One common misconception about vector architectures is that the hardware resources they require are many times more expensive than those of superscalar architectures, in particular the vector register file and the high-bandwidth memory system. In fact, as I discuss in this section, the hardware resources required to implement a vector architecture are comparable or, in some aspects, even less costly when supporting the *same* amount of parallelism as a superscalar implementation. In particular,

- the number of functional units can be the same;
- the need for a high performance memory is the same;
- the area of a vector register file is comparable to that of a superscalar multiported register file; and
- the instruction-issue logic of a vector implementation is less complicated.

Moreover, as hardware designers increase the amount of parallelism in the processor, the cost advantage of a vector architecture over a superscalar one becomes more pronounced with respect to the register file and issue logic.

3.1.1 Number of Functional Units

To support the simultaneous initiation of N operations, N functional units must be provided in either vector or superscalar architectures. Furthermore, either one will need the same number of buses to deliver operands and results between the functional units and the register file because this number is solely dependent on the number of functional units provided. Hence, the implementation cost of functional units is identical for superscalar and vector architectures that support the same amount of fine-grain parallelism.

3.1.2 High-Performance Memory System

Based on past implementations, computer designers mistakenly believe that a vector processor must have a more expensive, high-bandwidth memory system than one required by a superscalar processor. Although this may be true historically, I do not believe an expensive memory system is a fundamental requirement of a vector processor. Moreover, I believe that a less costly memory system that is suitable for a superscalar architecture should also be suitable for a vector architecture. This is because either one, and in fact *any* processor architecture that supports fine-grain parallelism, will place a comparable demand on memory bandwidth as a natural consequence of executing multiple operations per clock period. Studies of instruction mixes show that memory operations make up 20–30% of the instructions executed for a typical program [92]. In a scalar implementation with a performance goal of one instruction per clock period, the memory demand for data is one memory access every three to five clock periods. In a superscalar or vector implementation, where multiple operations are executed per clock period, the demand for data can be as frequent as a memory access every two to three clock periods or even every clock period. In fact, with enough datapath parallelism in the hardware, the demand for data can be greater than one memory access per cycle.

Because memory demand for data can be so high, multiple memory-ports will be necessary with increasing datapath parallelism in *either* a vector or superscalar architecture. Continually increasing the parallelism of the datapath without increasing the number of memory ports will ultimately make memory the bottleneck to improved performance. Memory ports are more expensive to add than floating-point units, however, because increasing their numbers impacts the entire memory system. Although several vector computers have implemented multiple memory-ports, superscalar architectures have yet to do so.

Vector computers typically use a large, highly-interleaved main memory built from expensive SRAM chips. In contrast, superscalar computers follow their scalar ancestors by using cache-based memory systems that are less costly and reputedly provide sufficient performance. But, not only is the demand for memory bandwidth independent of the processor architecture, so is the implementation of the memory system. Consequently, although high-cost memory systems have routinely been used in vector computers, a cached-based memory system could be used as a more cost-effective solution. For example, the IBM 3090 has a 64-Kbyte, 4-way set-associative cache [113]. Furthermore, research into cache designs that provide high memory-bandwidth for superscalar architectures [104] could also be applied to vector architectures.

Cache-based vector computers are not common because popular wisdom suggests that scientific and engineering programs, which are most suitable for a vector architecture, have memory reference patterns with poor spatial and temporal locality. Although these characteristics result in poor cache performance, this performance has more to do with the program itself rather than the design of the processor. Accordingly, if a cache-based vector computer exhibits poor cache performance, so will a cache-based superscalar computer when executing the same program.

Evidence that the program is the major influence on cache behavior comes from Clark and Wilson who present performance data for the vector cache in the IBM 3090 [21]. To multiply two matrices of dimensions $300 \times N$ and $N \times 100$, where N is varied from 50–600 by increments of 50, they use three different algorithms[1] to improve the cache performance as measured by execution time. They find that for each algorithm, the cache performance curves of both scalar and vector processing have the same shape. Moreover, the straightforward algorithm for matrix multiply has declining cache performance for both scalar and vector processing as the problem size increases, whereas two *blocked* algorithms have cache behavior that is insensitive to the size of the problem. Rather than working on an entire row or column of a matrix, such *blocked* algorithms rearrange computations to work on submatrices or blocks that will fit in a cache [45, 77]. Such rearrangements are designed not to affect the vectorizability of a program [29]. Hence, if a blocked algorithm performs

[1] I use the words *program* and *algorithm* interchangeably.

well on a superscalar architecture, it should also perform well on a vector architecture.

Although memory implementation and memory demand are independent of the processor architecture, a vector architecture has several features that can simplify the design of a memory system: less memory traffic than in a superscalar architecture, a built-in mechanism for prefetching data, and fewer wires in the memory interface than what is needed for a superscalar architecture. The last feature is especially advantageous when multiple memory-ports are implemented. In the following paragraphs, I qualitatively describe the advantages of these features, leaving a quantitative analysis for a future study.

First, whenever parallelism is exhibited, memory traffic from a vector architecture is less than it is from a superscalar architecture. Parallelism demands substantial data bandwidth. In a vector architecture, however, memory traffic for instructions does not increase when memory traffic for data does because vector instructions are used. Conversely, in a superscalar architecture, instruction demand increases in conjunction with data demand because each operation corresponds to an instruction. Even if an instruction cache is used to reduce traffic to main memory, a cache for a superscalar architecture must deliver many more instructions than one in a vector architecture. Moreover, such an instruction cache may also have to be larger to provide a good hit ratio when techniques such as loop unrolling are used to increase the amount of available parallelism at the expense of increased code size.

Second, vector memory instructions prefetch data from the memory system. Because a stream of references through a memory port could exhibit a regular pattern, a high-bandwidth memory system can be designed to take advantage of this regularity. A vector memory instruction encodes this pattern in three pieces of information: the base address, the offset between successive addresses (known as the *stride*), and the number of words to access. Hence, the memory system is told about the pattern at the time a vector memory instruction is issued. In contrast, a superscalar architecture treats memory references individually. Consequently, additional hardware would be needed to first discover the pattern. Alternatively, prefetching could be performed by the software by including extra instructions [17, 72]. A disadvantage of this, however, is an increase in memory traffic for instructions.

Finally, the physical package that contains a vector VLSI processor requires significantly fewer pins to communicate with the memory system than one that contains a superscalar processor. This is an important consideration when implementing a single-chip VLSI processor with a limited number of pins. Because a superscalar architecture has no mechanism for encapsulating multiple memory references, the processor computes and sends the address of each memory operation to the memory system. Consequently, each memory port must have associated with it both a data and an address bus. In addition, because each address must be sent out from the processor and the demand for data will be one or more accesses each clock period, the address pins will be used just as frequently.

In contrast, a vector memory instruction provides only three pieces of information to specify multiple memory references. As long as the memory system can generate addresses, this information can be sent once at the time an instruction is issued rather than sending an address for each reference of a vector memory instruction. Although this complicates the memory controller somewhat, the number of address buses is kept to *only* *one*, even with multiple memory-ports.[2] In other words, each extra memory port requires 32 fewer pins (assuming 32-bit addresses) in a vector architecture than the number needed in a superscalar one. Moreover, the address pins are used only once for each vector memory instruction rather than for each memory reference, as in a superscalar architecture. Because placing an electrical load on many pins simultaneously causes electrical difficulties to arise, fewer pins and less frequent use of the address pins together simplify the electrical design of a VLSI chip with hundreds of pins.

3.1.3 Register File

Hardware designers also mistakenly believe that the vector register file is costly to implement because vector architectures have many more registers than do superscalar. Although the area cost is greater, the design of the vector register file results in an increase in area that is far less than the increase in the number of registers. As described earlier in Section 2.2, vector and superscalar architectures use different configurations for implementing a multiported register: vector architectures use a partitioned configuration whereas superscalar architectures use a monolithic one. These two configurations represent different tradeoffs between area and accessibility.

A partitioned register file can contain far more registers than a monolithic register file without a corresponding increase in area. For example, the vector register file in the Cray Y-MP vector processor consists of 8 dual-ported vector registers, each with 64 elements, for a total of 512 registers. The register file in the Texas Instrument Megacell chip, used in both the Hewlett Packard Snake workstation and Sun SuperSPARC chip set, consists of 32 registers with 5 read-ports and 3 write-ports. As shown in Figures 3.1 and 3.2, dual-ported registers have a smaller area than multiported ones. Assuming each register is 64-bits wide, the area of this vector register file ($67.2 \times 10^6\lambda^2$) is only 2.5 times that of the superscalar one ($26.9 \times 10^6\lambda^2$) while providing 16 times more registers.

Moreover, this difference in area could be smaller if a superscalar architecture implemented more than the typical 32 registers, a possibility indicated by Wall's parallelism study [115]. More registers are needed for additional datapath parallelism because supporting more parallelism will produce more intermediate results at once, thus requiring more registers to simultaneously store these results. Wall's data, reproduced in Figure 3.3, shows a direct correlation between more registers and more attainable parallelism. Because how a compiler uses registers affects how many will be needed, Wall's experiment uses register renaming to diminish a compiler's influence. Because Wall's study did not assume a particular architectural approach, this data suggests that any architecture that supports parallelism will need to provide more registers.

The need for more registers is also demonstrated in several commercial implementations. The IBM RS/6000, a superscalar design, actually implements 38 floating-point registers, six of which are invisible to the programmer and are used for renaming registers.

[2] The exception to this is a gather/scatter vector instruction where the processor computes and then sends the address of each memory operation. As long as only one gather/scatter memory-port is implemented, as is the case in the Cray Y-MP, only one address bus is needed to support multiple memory-ports in a vector architecture.

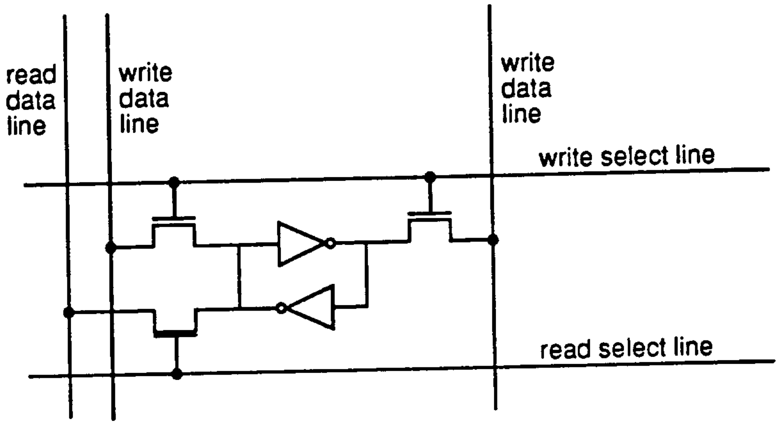


Figure 3.1: Design of a Multiported Register Cell

The multiported register cells listed in Figure 3.2 are a straightforward extension of the 1R,1W register cell design shown above. In designing the cells, the VLSI technology used is scalable CMOS where $0.4\mu \leq \lambda \leq 2\mu$ and the minimum line width is 2λ. I use only two metal layers with a minimum pitch of 8λ (i.e., the minimum width of a metal line is 4λ and the minimum distance between lines is 4λ). VLSI technology that is capable of more than two metal layers would not help in reducing the size of the register cell because the extra layers of metal are much coarser: a minimum width about three to five times wider than that of the first two layers.

The memory portion of each register cell is a pair of cross-coupled inverters consisting of four transistors that force a minimum height of 41λ. To access the register cell, each port requires one transistor, a select line, and a data line. In addition, a write port requires a second access transistor and data line. In the diagram above, the top two transistors are the access devices for the write port and the bottom transistor is the access device for the read port.

The area of the register cell grows approximately as the square of the number of ports added because each port forces the cell to grow in both height and width. Because the memory portion of the cell can accommodate three select lines running width-wise across the cell, the height of the cell does not grow until more than three ports are implemented, after which each port adds 8λ to the height. A read port adds 14λ to the width: 8λ for the data line and 6λ for the access transistor. A write port adds 28λ to the width because it requires two data lines and two access transistors.

NUMBER OF PORTS	DIMENSIONS (w × h)	AREA (relative area)	COMMERCIAL MACHINES
1R,1W	$50\lambda \times 41\lambda$	$2050\lambda^2$ (1.00)	
2R,1W	$64\lambda \times 41\lambda$	$2624\lambda^2$ (1.28)	MIPS R3000, MIPS R4000, most RISC scalar microprocessors
4R,2W	$120\lambda \times 65\lambda$	$7800\lambda^2$ (3.80)	IBM RS/6000, SUN SuperSPARC IU
5R,3W	$162\lambda \times 81\lambda$	$13122\lambda^2$ (6.40)	Hewlett-Packard Snake, SUN SuperSPARC FPU

Figure 3.2: Area Requirements of Multiported Register Cells

This table lists the area requirements of various multiported register cells based on the design in Figure 3.1. The description of the register cell design explains why the area increases approximately as the square of the number of register cell ports.

The column labeled COMMERCIAL MACHINES lists processors that use an analogous register cell, which is not necessarily the same size because it may use a different implementation technology. Both the Hewlett-Packard Snake and SUN SuperSPARC use the Texas Instruments Megacell chip that implements a register cell with 5 read ports and 3 write ports.

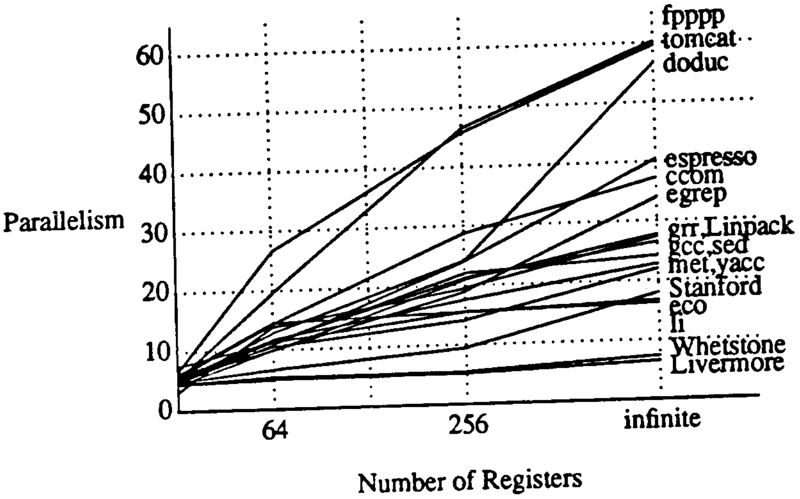


Figure 3.3: Number of Registers versus Parallelism

Based on Wall's parallelism data [115], this graph shows that the number of registers affects how much parallelism can be extracted. To show the extent to which the number of registers can impede parallelism, perfect branch/jump prediction and perfect dependence analysis are assumed. Similar trends also occur for more realistic models of computation.

VLIW implementations, which are designed with more datapath parallelism than is found in superscalar machines, have a large number of 64-bit registers:

IMPLEMENTATION	NUMBER OF 64-BIT REGISTERS
Intel iWarp	64
Multiflow Trace 7	7 × 32 = 224
Multiflow Trace 14	14 × 32 = 448
Multiflow Trace 28	28 × 32 = 896
Cydrome Cydra 5	6 × 64 = 384

Finally, vector architectures use a large number of registers ranging from 512 in a Cray processor to 8192 in Ardent and Fujitsu processors (see Figure 2.5 on page 15).

Given that more than 32 registers are required to support a reasonable amount of fine-grain parallelism, Figure 3.4 shows that the partitioned approach is more attractive than the monolithic approach from the perspective of hardware cost because for the same area many more registers can be implemented in a partitioned register file than in a monolithic one. The overall size of the register file is determined mainly by the size of the register cell, the most replicated part of the register file. Other components that are needed to access the register file, such as decoders and read/write drivers for the data lines, are typically less than 5% of the area required by the register cells themselves (assuming 64-bit registers). Consequently, the *relative* size of the two register files is equal to the relative sizes of the register cells. If $S_{R,W}$ is the size of a register cell with R read ports and W write ports, the partitioned register file can implement $\frac{S_{R,W}}{S_{1,1}}$ times more registers in the same area. Alternatively, the partitioned register file requires $\frac{S_{1,1}}{S_{R,W}}$ times less area to implement the same number of registers. Figure 3.2 lists some values for the $\frac{S_{R,W}}{S_{1,1}}$ ratio.

In fact, this difference in area may be *reversed* with increasing datapath parallelism because a monolithic register file needs to expand even if the number of registers remains unchanged, whereas a partitioned register file does not. Increasing datapath parallelism requires a corresponding increase in the number of ports in the register file because these should match the number of operands and results used and produced by the functional units. Of the two techniques for providing a multiported register file, the partitioned approach, which uses dual-ported registers, scales better with increasing datapath parallelism than does the monolithic approach, which uses multiported registers. To support more parallelism, the monolithic approach uses a register cell with more ports. As Figure 3.2 shows, adding more ports to a register cell expands the area in both dimensions. Hence, the area of a monolithic register file enlarges as the square of the increase in the number of ports even though the number of registers does not change. By contrast, adding more ports to a partitioned register file entails partitioning the file further but without having to change the size of the register cell. If the total number of registers remains unchanged, only a minimal increase in area will result when adding more "ports" to the register file.

Although a partitioned register file requires less area than a monolithic one, the former also has restrictions on which registers are available for use each clock period. However, as datapath parallelism is increased, the partitioned approach will be the better design for reasons of cost, despite the loss in functionality.

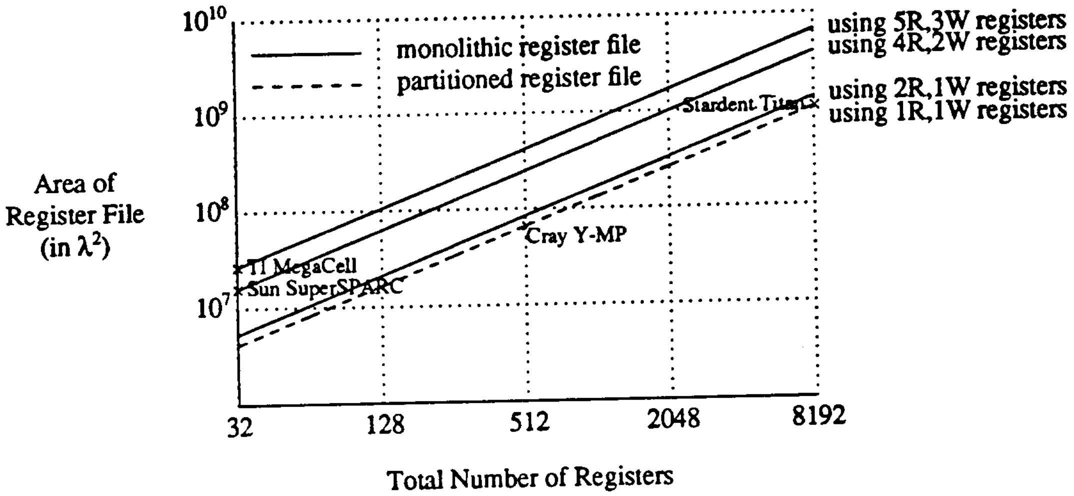


Figure 3.4: Area Requirements of Monolithic and Partitioned Register Files

The graph above compares the area requirements of the monolithic and partitioned approaches for implementing a multiported register file. Note the log scale on both axes. The areas are based on the design of the register cells described in Figure 3.2.

As points of reference, I have identified data points that correspond to machines with the same register file parameters although not necessarily the same area because the actual implementation may use a different technology. In particular, assuming the same technology, the register file of the Cray Y-MP would require 2.5 times as much area as the register file of the Texas Instrument Megacell, but it would provide 16 times as many registers.

A superscalar architecture could implement a partitioned register file, although this is not traditionally done. To prevent any loss in performance as a result of too many conflicts in register accesses, however, a new algorithm would be needed to assign values to registers in register banks. For example, the Multiflow Trace 14/300 system, which is a VLIW architecture, has a register file with read and write restrictions that effectively partition the file into register banks. Experience with this system led its designers to observe that "the more byzantine the constraints put on the code generator, the worse the code quality," which in turn results in performance loss [22]. Because of the similarities between a VLIW architecture and a superscalar one, this observation would probably hold true for a superscalar implementation with a comparable register file organization. Hence, the need for a good assignment algorithm is probably the main reason why superscalar architectures have not implemented a partitioned register file.

In contrast, a partitioned register file fits well with the vector architecture in that no special algorithm for register assignment is needed to overcome the restrictive accessibility of such a design. Because a vector instruction operates on a vector of data which can be assigned to a vector register, register assignment for a vector register file can occur at the vector-register level rather than at the level of individual registers. Hence, as a natural consequence of the vector computational model, traditional algorithms for assigning values to registers in a scalar register file can be used to assign vectors of data to vector registers in a vector register file. Furthermore, a vector architecture can easily support more datapath parallelism because a partitioned register file can support more register-ports with a minimal increase in area.

3.1.4 Instruction-Issue Logic

Another advantage of a vector architecture is its simpler instruction-issue mechanism for simultaneously initiating multiple operations. In general, before an instruction can be issued, interlock logic in the hardware must first determine that no data or structural hazards exist between an instruction and any previous one in the instruction stream. In addition to the number of instructions that must be examined simultaneously, another indicator of the amount of hardware needed to simultaneously initiate multiple operations is the number of hazard checks that must be performed in parallel.

In a vector architecture, because only one instruction per clock period is ever handled by the interlock logic, the amount of hardware required to check for hazards is about the same as in a scalar implementation. In addition, no hardware checks are performed among the individual operations of a vector instruction because a compiler has guaranteed that the appropriate operations have been grouped into one vector instruction; hence, hardware does not duplicate the work of the compiler.

In contrast, the issue mechanism of a superscalar architecture requires more hardware than that of a vector processor with equivalent parallelism support. First, the interlock logic must simultaneously examine a *minimum* of N instructions as a requirement for full usage of N functional units. Moreover, because there must be a hazard check between each instruction and any previous one in the instruction stream, not only are there hazard checks between each examined instruction and already-issued instructions, as shown in Figure 3.5, but there are also explicit pair-wise checks among the about-to-issue instructions. Each of

NUMBER OF FUNCTIONAL UNITS	NUMBER OF PAIR-WISE HAZARD CHECKS PER CLOCK
2	3
3	6
N	$\sum_{i=1}^{N} i = \frac{(N-1)N}{2}$

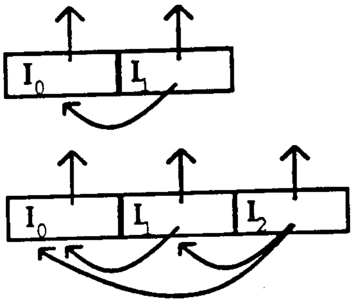


Figure 3.5: Instruction-Issue Logic in a Superscalar Architecture

This figure shows how the hardware cost for issuing instructions in a superscalar architecture grows as the square of the number of instructions that are simultaneously examined for issuing. The minimum number of instructions that must be examined per clock period is equal to the number of functional units. An instruction can be issued only if there are no data or structural hazards between itself and any previous instruction in the instruction stream. Moreover, these checks for hazards must execute in parallel. Consequently, for each instruction in the issue window, there is interlock logic for detecting hazards with instructions already issued (indicated by the vertical arrows). In addition, there is interlock logic for detecting hazards between each pair of instructions in the issue window (indicated by the right-to-left arrows).

these checks must be implemented in hardware so that they can execute in parallel. In addition, extra hardware is needed to handle any instructions with hazards, either to stall the instruction stream at the first instruction with a hazard or to design the pipeline to allow forwarding of data [13]. A consequence of this increase in hardware for issuing instructions is that the design and diagnostics required for functional testing are also more complex and hence, more likely to take longer to complete.

This difference in hardware for issuing instructions is greatly magnified as hardware designers increase the parallelism in a processor. In a vector architecture, because sequentially-issued vector instructions allow operations to be initiated in parallel, the number of instructions handled by the interlock logic *remains at one* even as datapath parallelism grows. In contrast, as Figure 3.5 shows, the total number of pair-wise hazard checks required in a superscalar processor increases as the square of the increase in datapath parallelism. This quadratic growth in hardware was described as early as 1970 by Tjaden and Flynn [111]. Recently, Johnson described techniques for reducing this hardware cost but such techniques have the adverse effect of complicating the hardware design [66]. Extra logic for handling hazards and the increase in debugging complexity also magnifies at the same rate as that of the interlock hardware.

3.2 The Effectiveness of Vector Architectures

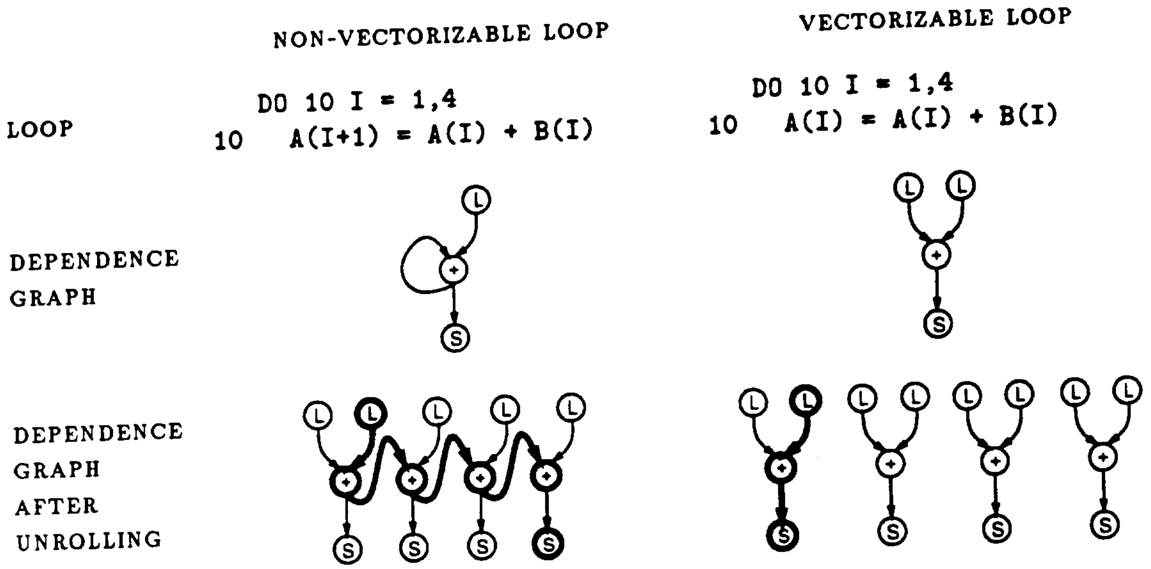
Another common misconception about vector architectures is that they are effective for only a small set of programs, and then only for those portions of a program containing loops that have no self-dependent statements. Moreover, proponents of superscalar architectures believe that, for less cost, a superscalar architecture can take advantage of vectorizable parallelism as well as non-vectorizable parallelism. I believe the opposite to be true. In this section, I address the issue of effectiveness, arguing that a vector architecture is, in fact, highly effective at using fine-grain parallelism.

There are three parts to my argument. In Section 3.2.1, I present data from a parallelism study that suggests that vectorizable programs have an abundancy of parallelism and that only a minuscule amount of parallelism is available elsewhere for *any* architecture. This data also suggests that vectorizable programs are likely to be the more time-consuming ones in a workload. Traditional analyses of this data, which is based on reducing execution time, tend to downplay time consumption. In Section 3.2.2, I discuss an alternate measure of improvement, based on increased workload, that highlights the effectiveness of using parallelism. Nonetheless, Amdahl's Law reminds us that ignoring non-vectorizable program fragments completely would be unwise. In Section 3.2.3, I show how a combined superpipelined and vector architecture can take advantage of both the limited parallelism that is available in non-vectorizable program fragments and the abundancy of parallelism that is available in vectorizable ones.

3.2.1 Where Is the Parallelism?

An understanding of the properties of a vectorizable program fragment shows intuitively that a vectorizable loop intrinsically has more parallelism than a non-vectorizable one. Although many hardware and software techniques are used to expose the parallelism in a loop, it is the presence or absence of self-dependent statements that determines *how much* parallelism is present in a loop. For example, while unrolling a loop is a software technique for exposing more parallelism, unrolling a vectorizable loop, which has no self-dependent statements, produces more parallelism than unrolling a non-vectorizable one that has a comparable number of operations. This is because the dependence graph of an unrolled vectorizable loop has a path of maximal length, known as a *critical path*, much shorter than a critical path in the dependence graph of an unrolled non-vectorizable loop. As Figure 3.6 illustrates, a self-dependent statement results in a critical path whose length is proportional to the number of iterations executed. In contrast, the absence of a self-dependent statement produces a critical path whose length is proportional to the number of operations executed in an iteration. An indication of the amount of parallelism available in a loop is the ratio of the number of operations executed for the loop and the number of operations in a critical path of the loop. Because the number of iterations executed for a loop is typically greater than the number of operations executed for one iteration, the critical path of a vectorizable loop will be shorter than that of a non-vectorizable loop that has a comparable number of operations. Hence, in theory, a vectorizable loop has more parallelism than a non-vectorizable one.

In support of this intuitive explanation, I use data from a study performed by Wall



```
   DO 10 I = 1,4
10   A(I+1) = A(I) + B(I)
```

```
   DO 10 I = 1,4
10   A(I) = A(I) + B(I)
```

Figure 3.6: Intrinsic Parallelism in Non-vectorizable and Vectorizable Loops

This figure demonstrates the difference in the intrinsic parallelism in a non-vectorizable loop and a vectorizable one with a comparable number of operations. The dependence graph that results from unrolling a loop illustrates the intrinsic parallelism of that loop. Unrolling a non-vectorizable loop, which has a self-dependent statement, produces a dependence graph with a chain of dependent operations that are executed for different iterations. This chain of dependent operations, outlined in bold, is typically the critical, or longest, path in the execution of such a loop. Although transformations, such as tree-height reduction or cyclic reduction [74] can be made to eliminate some dependences in a non-vectorizable loop that has been unrolled, not all dependences between iterations can be removed and the resultant dependence graph will remain connected. In contrast, the dependence graph of a vectorizable loop, after unrolling, contains disjoint subgraphs, each of which represents one iteration of the loop. A critical path of this loop, outlined in bold, is limited to the operations in one iteration and is, hence, shorter than the critical path in the non-vectorizable loop.

that indicates that programs with vectorizable loops do have plenty of parallelism while non-vectorizable program fragments, for the most part, have relatively little parallelism, even under favorable hardware and software conditions [115]. Although Wall's study provides data that is favorable to vector architectures, the purpose of his study is to search for any type of parallelism. Moreover, because Wall does not assume a vector architecture nor rely on a vectorizing compiler to generate his data, this study provides independent evidence that parallelism, when it exists in quantity, is suitable for such an architecture.

Wall measured, under a variety of hardware and software conditions, the parallelism available in 17 programs representative of those that would be executed on a workstation. Wall identifies and varies three parameters that affect how much parallelism can be extracted from a program:

1. the level of branch/jump prediction to find parallelism across multiple basic blocks;
2. the number of registers for renaming purposes to eliminate false register dependences; and
3. the level of dependence analysis (called *alias analysis* by Wall) to identify when two memory references access the same location.

Figure 3.7 lists some of the parameter values used by Wall. Another parameter, multiple functional-units, is fixed at 64 for this study. Varying the value of these parameters results in different models of computation. The model of computation that is closest to what a vector architecture can provide today has the following parameter values:

- static branch/jump prediction, which chooses the most frequent target based on a profile from an identical run;
- 256 integer registers, 256 floating-point registers; and
- perfect dependence analysis of stack and global references, and instruction inspection to identify memory dependences among heap references (called *alias analysis by compiler* by Wall).

Although Wall did not include data for this particular computational model, a reasonable approximation is the one that uses perfect dependence analysis because the parallelism demonstrated for most of the computational models using *compiler analysis* is comparable to what is exhibited for the corresponding computational model using *perfect analysis.*

In Figure 3.8, I have reproduced the parallelism data for five models of computation:

1. *bNone,jNone,r256,aNone* shows how much parallelism is extractable using basic scalar execution and a generous supply of registers;
2. *bNone,jNone,r256,aPerfect* indicates the parallelism available within basic blocks;
3. *bStatic,jStatic,r256,aPerfect* approximates how much parallelism can be obtained by a vector compiler and processor;

LEVEL OF BRANCH/JUMP PREDICTION	
bNone,jNone	branch/jump targets are not predicted at all
bStatic,jStatic	a branch/jump is predicted to go to its most frequent target as determined by a profile from an identical run
bInfinite,jInfinite	the target of a branch/jump is dynamically predicted based on a two-bit counting scheme in which the table holding the branch histories is infinitely large
bPerfect,jPerfect	branch/jump targets are always correctly predicted

NUMBER OF REGISTERS	
r256	256 integer registers and 256 floating-point registers dynamically allocated in an LRU fashion
rPerfect	an infinite number of registers to completely eliminate all false register dependences

LEVEL OF DEPENDENCE ANALYSIS	
aNone	no memory dependences are identified, and all loads and stores are assumed to conflict
aPerfect	all memory dependences are identified, and loads and stores conflict only if they access the same memory location

Figure 3.7: Parameter Values for Models of Computation

This table gives the details of the parameter values used for the computational models displayed in Figure 3.8. Some amount of branch/jump prediction, in effect, increases the basic block sizes and hence the number of instructions that can be considered for parallel execution. A more accurate prediction scheme results in fewer wasted cycles that occur when the processor flushes any pending instructions after incorrectly predicting a branch/jump. Register renaming is used to reduce the number of false register dependences that arise because the executable code is compiled for an architecture with 32 registers. The number of registers provided by the hardware determines the number of false dependences that can be eliminated. Because dependence analysis (called ***alias analysis*** by Wall) identifies instructions that access the same memory location, the ***level*** of dependence analysis affects how many instructions can execute in parallel. For identifying memory dependences, I use the two extremes of dependence analysis.

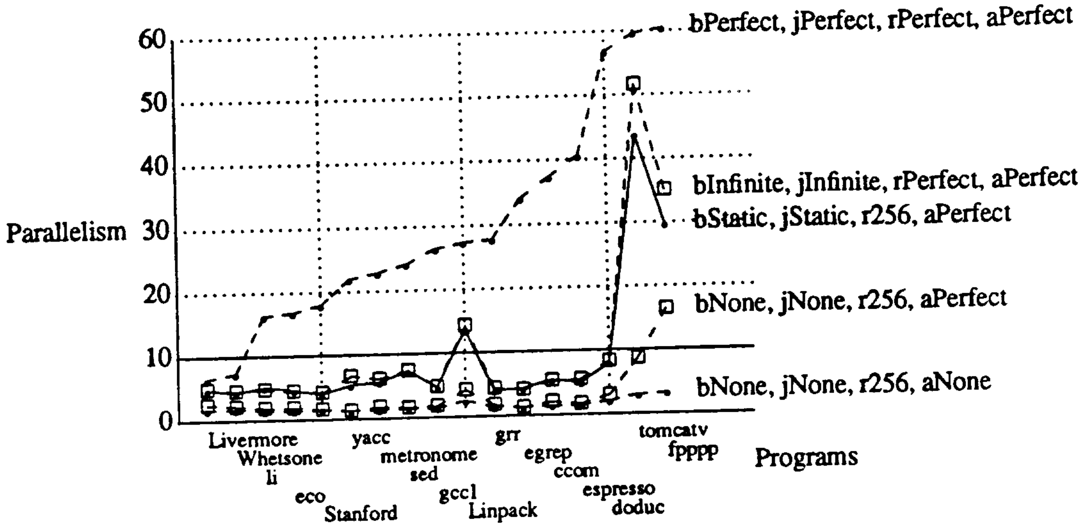


Figure 3.8: Measured Parallelism under Various Hardware and Software Conditions

This graph reproduces some of Wall's data [115] showing the amount of parallelism that can be extracted under five different models of computation. These models vary in the number of available registers, and in their ability to do branch/jump prediction and identify dependences that involve memory. Figure 3.7 gives details on the values of individual parameters. The inputs for all these models are the executable codes of programs compiled for a DECStation 5000 that uses a MIPS R3000 processor, a scalar architecture with 32 registers.

4. *bInfinite,jInfinite,rPerfect,aPerfect* indicates how much parallelism is accessible under nearly impossible conditions; and

5. *bPerfect,jPerfect,rPerfect,aPerfect* gives the intrinsic parallelism in a program.

For the vector computational model, only the 3 programs *fpppp, tomcatv,* and *linpack* have parallelism above 10, whereas the other 14 programs have parallelism between 4 and 8. Of the three programs, *tomcatv* and *linpack* contain loops vectorizable by *cft77*, the vectorizing compiler developed by Cray Research, Incorporated. More importantly, these loops constitute the bulk of the time it takes to execute each program.

The program *fpppp* is unique in that it contains large basic blocks [30] that have parallelism in quantity. This is unusual because large basic blocks do not guarantee copious amounts of parallelism as evidenced by the generally low levels of parallelism exhibited under a nearly impossible computational model (labeled *bInfinite,jInfinite,rPerfect,aPerfect* in Figure 3.8), which assumes aggressive branch/jump prediction techniques to enlarge basic blocks. The large basic-blocks of *fpppp*, however, do contain an unusual abundance of parallelism as demonstrated by the fact that only *fpppp* has parallelism greater than 10 when there is no branch/jump prediction, a generous supply of registers, and perfect alias analysis, the model of computation (labeled *bNone,jNone,r256,aPerfect* in Figure 3.8) that measures the parallelism available exclusively in a basic block. Although current vectorizing compilers are unable to identify and express this parallelism in vector terms, perhaps with more research *fpppp* will be vectorizable in the near future.

Of the 14 programs that have parallelism between 4 and 8, only the *Livermore Loops* benchmark is known to contain any vectorizable loops. Extracted from actual scientific applications, this benchmark is a set of representative loops [84], half of which are loops with self-dependent statements. Hence, this data bolsters my claim that there is meager parallelism to be found in non-vectorizable loops. Wall states that the parallelism for each individual loop in this benchmark ranges from 2.4 to 29.9 with a median around 5. The reason a parallelism of only 4.9 is demonstrated over all the loops is a consequence of Amdahl's Law, an issue I will address shortly.

Providing increasingly more resources for branch/jump prediction and registers does not significantly change this parallelism profile. Even under nearly impossible conditions (Wall's *Great* model which is labeled *bInfinite,jInfinite,rPerfect,aPerfect*in Figure 3.8), only *tomcatv* and *fpppp* show a significant increase in parallelism over the vector model. A major change in the parallelism profile occurs when perfect branch and jump prediction is assumed (Wall's *Perfect* model which is labeled *bPerfect,jPerfect,rPerfect,aPerfect* in Figure 3.8). Under such impossible conditions, all but *Whetstones* and *Livermore Loops* demonstrate parallelism greater than 15. Hence, although there is parallelism intrinsic in non-vectorizable program fragments, it is not as easily extracted as parallelism in vectorizable loops, which require only a good dependence analyzer in the compiler — a technology that is already available.

3.2.2 The Effectiveness of Parallelism

In the previous subsection, I demonstrated that there is abundant parallelism in vectorizable programs. In this subsection, I will quantify how effectively this parallelism can be used to improve the performance of a workload as typified by the one in Wall's study.

The standard reason for using parallelism is to reduce execution time but, because of Amdahl's Law, measuring improvement in this fashion tends to downplay the overall benefits of parallelism. For example, using the instruction counts listed in Figure 3.9 as approximations to execution times, the overall speedup is:

$$\frac{\text{execution time of total workload}}{\sum \frac{\text{execution time of program } i}{\text{parallelism of program } i}} = 9.81$$

In other words, the total execution time of Wall's workload can be reduced by a factor of less than 10 despite the fact that the program that accounts for 44.5% of the executed instructions exhibits the largest amount of parallelism (44). This rather low speedup results because *li*, the second longest running program, exhibits little parallelism (5.2).

As an alternative, a different reason for using parallelism is to execute larger problems in the same amount of time, thus providing a different measure of improvement based on increasing workload. Gustafson has shown quantitatively the importance of increasing the size of a workload to provide more parallelism [54]. To facilitate acceptance of this relatively new measure of performance, Gustafson *et al.* have constructed the SLALOM[3] benchmark, which compares the performance of computers by measuring the number of polygons a computer can generate in one minute [53]. To use this measure for Wall's data, I assume that the time contributed by each program remains the same (an assumption I will discuss shortly). Thus, the overall increase in Wall's workload using parallelism is:

$$\frac{\sum(\text{execution time of program } i \times \text{parallelism of program } i)}{\text{execution time of entire workload}} = 24.11$$

In other words, about 25 times as many instructions can be executed without increasing the workload's execution time.

These two ways of measuring improvement rely upon slightly different concepts of what a workload is. A workload, when improvement is measured by reduced execution time, is characterized by a set of programs and their corresponding *inputs*. A workload, when improvement is measured by its enlargement, is characterized by a set of programs and their *time contributions* to the workload.

The dramatic improvement when measured by increased workload is due to the fact that the most time-consuming program in Wall's workload also happens to exhibit the most parallelism, an important characteristic of any workload if significant gains from parallelism are to be obtained. Let me demonstrate the importance of this characteristic with a fictitious workload as a counter-example:

> By switching the parallelism numbers for *tomcatv* and *sed*, the least time-consuming program now exhibits the most parallelism. The improvement of this fictitious workload when measured by reduced execution time is 6.6, 30% less than the corresponding improvement of Wall's workload. More significantly, the improvement when measured by increased workload is only 8, a *factor of three* less than that for Wall's workload.

Hence, using the size of a workload as a measure of improvement serves to highlight the effectiveness of parallelism, particularly if a workload contains highly parallel programs that are also the most time-consuming.

When measuring improvement by increased workload, I assumed that the time contributed by each program remains the same. This is equivalent to assuming that programs with more parallelism are the ones that a user wishes to execute the most, a reasonable assumption given the context in which I am making my case. In Wall's workload, the programs with the most parallelism come from the scientific and engineering domain. The push for higher performance computers, such as workstations and MPPs, comes from this application domain. Two major benchmarking efforts reflect this: all 13 programs of the Perfect Club Suite [12] and six out of the 10 programs in the SPEC suite [106] come from scientific and engineering applications.

Given the fact that Wall's workload represents about three minutes of execution time (assuming a 25 MHz clock frequency, which is used in a SUN SPARCstation, and ideal cache behaviour), increasing workload seems a better use of parallelism rather than simply

[3] *SLALOM* is an acronym for *S*calable *L*anguage-independent *A*mes *L*aboratory *O*ne-minute *M*easurement.

PROGRAM	LINES	EXECUTED INSTRUCTIONS	PARALLELISM
		number (percentage)	
tomcatv	180	1,986,257,545 (44.5)	43.7
li	7000	1,247,190,509 (27.9)	5.2
fpppp	2600	244,124,171 (5.5)	29.7
doduc	5200	284,697,827 (6.4)	8.0
Linpack	814	174,883,597 (3.9)	13.6
espresso	12000	135,317,102 (3.0)	5.0
grr	5883	142,980,475 (3.1)	4.0
metronome	4287	70,235,508 (1.6)	5.7
yacc	1856	30,948,883 (0.7)	5.3
eco	2721	26,702,439 (0.6)	4.7
Whetstones	462	24,479,634 (0.5)	4.4
gccl	83000	22,745,232 (0.5)	4.8
Livermore	268	22,294,030 (0.5)	4.9
Stanford	1019	20,759,516 (0.5)	4.0
ccom	10142	18,465,797 (0.4)	5.5
egrep	844	13,910,586 (0.3)	4.3
sed	1751	1,447,717 (0.03)	7.4
TOTAL			9.81 harmonic mean
WORKLOAD	140027	4,467,440,568 (99.93)	24.11 arithmetic mean

Figure 3.9: Execution Characteristics of Wall's 17 Programs

This table lists some characteristics of the programs in Wall's parallelism study [115]: the number of source-code lines, the number and percentage of dynamic instructions, and the amount of parallelism demonstrated using a model of computation closest to what vector architectures can provide today. The number of instructions executed per program varies by a factor of 1000, ranging from 1.4 million to 2.0 billion with an average of 262.8 ± 533.0 million.

Parallelism for the total workload can be represented by two *weighted* averages, depending upon how performance improvement for the total workload is measured. The weight is a program's percentage of the workload with respect to time, which is estimated by the number of executed instructions. Improvement as measured by reduced execution time is equivalent to the weighted *harmonic* mean of the amount of parallelism, whereas improvement as measured by increased workload is equivalent to the weighted *arithmetic* mean of the amount of parallelism. In either case, using the unweighted mean excludes the time contributed by each program and will not reflect the fact that, in this workload, the most time-consuming program benefits the most from parallelism.

reducing execution time. Continually measuring improvement in terms of reduced execution time will ultimately limit the amount of improvement that is theoretically possible. In contrast, measuring improvement in terms of increased workload has no limitation assuming that the sizes of problems can grow indefinitely.

3.2.3 Addressing Amdahl's Law

I have just argued that parallelism, when available in quantity, can be used effectively by a vector architecture. Nevertheless, some amount of parallelism does exist in non-vectorizable parts. To avoid the consequences of Amdahl's Law, a vector architecture combined with superpipelined hardware can be used to take advantage of the limited parallelism (4–7) in such programs. A superpipelined extension to a vector architecture makes more sense than a superscalar extension because the instruction-issue logic for vector and superpipelined implementations are similar; in particular, both issue only one instruction per clock period. Moreover, Jouppi indicates that superpipelined hardware is more likely to take advantage of parallelism better than superscalar hardware because of nonuniformities in fine-grain parallelism [69]. In addition to using the limited parallelism in non-vectorizable fragments, which a basic vector architecture cannot take advantage of, superpipelined hardware can also be used in conjunction with vector hardware to execute vectorizable loops. In this subsection, I present data showing how effective this combination works on non-vectorizable and vectorizable programs.

Combining vector and superpipelined hardware can provide good scalar performance in a vector processor as evidenced by the Cray machines. Figure 3.10 compares the performance of the scalar MIPS R2000 in the DECstation 3100 and the superpipelined scalar portion of the Cray Y-MP executing *spice*, a circuit-simulation program that does not vectorize. As a point of reference, the clock frequencies of these two machines differ by a factor of 10: the Cray Y-MP has a cycle time of 6 ns whereas the MIPS R2000 has a cycle time of 60 ns [24, 92]. Because I want to emphasize the superpipelined aspect of the processors and not the implementation technology, this discussion is based solely on counts of clock periods. As Figure 3.11 shows, the Cray Y-MP has much longer latencies in terms of clock periods than does a basic scalar processor, such as the MIPS R2000, which range from 3 times longer for floating-point operations to 8 times longer for a memory operation. On the other hand, the MIPS R2000 has a much shorter memory latency because it uses a data cache. Despite longer latencies, the CPI (cycles per instruction) of the Cray Y-MP (4.13) is only slightly more than two times that of the MIPS R2000 (1.95).

This surprisingly low CPI indicates that some amount of parallelism is being used by the superpipelined hardware. This parallelism can be quantified by comparing the *measured* CPI with the *calculated* CPI, another ratio of cycles-per-instruction. The latter ratio is the weighted average of a processor's operational latencies where the weights are based on a program's operational mix. Although based on dynamic information, this metric does not take into account the interaction of the executed operations and indicates what the CPI would be without pipelined execution. Because the measured CPI does reflect parallel execution, the ratio of the calculated and measured CPIs ($6.77 \div 4.13 = 1.64$) is the amount of parallelism extracted by the superpipelined hardware of the Cray Y-MP.

Because superpipelining already improves performance through parallelism, how

spice < digsr	MIPS R2000	Cray Y-MP
#cycles	1711.7M	2744.9M
#instructions	875.8M	664.8M
	OPERATIONAL MIX	
#memory operations	325.7M (37%)	145.0M (22%)
#floating-point operations	112.5M (13%)	127.5M (19%)
#branches	60.0M (7%)	49.1M (7%)
#other operations	377.7M (43%)	343.1M (52%)
	RATIOS	
calculated CPI	1.64	6.77
measured CPI	1.95	4.13
average parallelism	0.87	1.64

Figure 3.10: Relative Performance of Superpipelined and Scalar Architectures

This table compares the performance of a superpipelined architecture, the Cray Y-MP, with that of a scalar architecture, the MIPS R2000 in the DECstation 3100, executing a non-vectorizable program, *spice*, that is simulating the circuit behavior of a digital-shift register. The operation latencies of these two processors are listed in Figure 3.11.

The dynamic information for the Cray Y-MP was gathered using the hardware performance monitor that is part of the computer. The dynamic information for the MIPS R2000, with the exception of the execution time, was gathered using a MIPS software tool called *pixie* that augments the executable with code to count the number of times each basic block is executed. Because *pixie* does not take into account cache misses, the *time* command in the UNIX operating system was used to determine the execution time.

The calculated CPI (cycles per instruction) for a program is the weighted average of a processor's operational latencies where the weights are taken from the operational mix for the program. This number represents what the CPI would be without pipelined execution. In computing this metric, I assume that the *floating-point operations* are equally divided between adds and multiplies. For the Cray Y-MP, I assume all *branches* are taken and a two-cycle latency for *other operations*. For the MIPS R2000, all *memory operations* are assumed to hit in the cache.

For the Cray Y-MP, the ratio of the calculated CPI and the measured CPI shows how much parallelism is extracted by the hardware because the measured CPI is based on actual execution, and the calculated CPI encompasses all the latencies seen by the processor. For the MIPS R2000, the difference between the calculated and measured CPIs indicates how many cycles per instruction, on average, are due to cache misses.

	MIPS R2000	Cray Y-MP
memory operation	2 CPs	17 CPs
floating-point add	2 CPs	7 CPs
floating-point multiply	3 CPs	8 CPs
branch	2 CPs	2–8 CPs
other operations	1 CP	1–7 CPs

Figure 3.11: Operation Latencies of Superpipelined and Scalar Architectures

This table shows the operation latencies of a superpipelined architecture, the Cray Y-MP, and a scalar architecture, the MIPS R2000.

The *memory operation* latency in the MIPS R2000 is based on a cache hit. *Branch* operations are any that can potentially change the sequential instruction stream, which includes conditional branches, jumps, and calls. The branch latency in the Cray Y-MP depends on whether the branch is taken or not taken. Changing the instruction stream results in an eight-cycle branch, whereas falling through takes two cycles. *Other operations* in the Cray Y-MP include population-count, logical and shift functions.

Most operations in the Cray Y-MP and the MIPS R2000 are delayed operations in that independent ones may execute in the delay slot(s) of an operation that requires more than one clock period to execute. The Cray Y-MP has no data cache and relies on the compiler to find enough operations to fill the delay slots of a *memory operation*. The only operations that are not are *branches* in the Cray Y-MP and *memory operations* resulting in a cache miss in the MIPS R2000.

	NUMBER OF KERNELS	3	+ 4	+ 7	= 14
MFLOPS CONTRIBUTION		$\sum_{i \in S} \frac{1}{R_i}$	$+ \sum_{i \in S?} \frac{1}{R_i}$	$+ \sum_{i \in V} \frac{1}{R_i}$	$= \sum \frac{1}{R_i} = \frac{14}{H}$
COMPILATION TECHNIQUE	scalar, unoptimized	0.600	+ 0.599	+ 1.009	= 2.208
	vector only	0.600	+ 0.599	+ 0.133	= 1.332
	scalar only, unroll 8x	0.301	+ 0.528	+ 0.427	= 1.256
	vector + unroll 8x	0.301	+ 0.528	+ 0.133	= 0.962

all values in seconds per million floating-point operations

R_i is the MFLOPS rate of the i^{th} kernel

H is the harmonic mean of the MFLOPS rates of the 14 kernels

Figure 3.12: Relative Performance of Superpipelined and Vector Architectures

To compare the performance of a superpipelined architecture and a vector architecture with superpipelining, this table shows the contributions of in MFLOPS of different classes of kernels to the harmonic means for different compilation techniques on the Cray-1. Instead of using harmonic means, which are measured in MFLOPS, I use sums of the inverses of the rates, measured in seconds per million floating-point operations, to highlight the contributions. This analysis is based on data collected by Weiss and Smith on the first 14 Livermore Kernels [119].

The set of 14 kernels can be divided into three different classes based on their vectorizability. The three kernels in the set S (called case 2 by Weiss and Smith) are strictly scalar. The four kernels in the set $S?$ (case 3) could possibly be vectorized, but not enough information is provided at compilation time to accurately determine this. The seven kernels in the set V (case 1) are vectorizable.

The data for the *scalar* compilation techniques and the harmonic mean for the *vector* compilation are taken directly from Weiss and Smith. The other numbers are derived as follows. For *vector* compilation, the value for the vector kernels is computed as the difference between the inverse of the vector harmonic mean and the sum of the inverses of the rates of the non-vectorizable kernels using unoptimized scalar compilation. For *vector + unroll* compilation, superpipelined hardware plus scalar unrolling techniques are used for the non-vectorizable kernels, and vector hardware plus vectorization are used for the vector kernels.

much more improvement can vector hardware provide? Weiss and Smith compared the performance of various scalar compilation techniques [119], allowing me to compare the performance of optimized scalar code with that of vectorized code. Although only scalar performance is discussed in this paper, a brief comparison with vector performance is made in the conclusion of the paper. The basis for the comparison is the harmonic mean for the first 14 Livermore kernels executing on a Cray-1S. Using vector code produces a harmonic mean of 10.51 MFLOPS, whereas the best scalar-compilation technique, which unrolls a loop 8 times and uses 64 scalar registers, slightly outperforms the vector version with a harmonic mean of 11.15 MFLOPS. Although this conclusion does not appear highly supportive of vector architectures, I use a more detailed breakdown of these performance summaries, presented in Figure 3.12, to show that a vector processor, *when* it can be used, is about three times faster than a superpipelined scalar processor.

Scalar performance is comparable overall to that of vector performance because this is another instance of Amdahl's law: a smaller average speedup over a larger portion of the workload can result in better performance over the entire workload than a much larger average speedup over a smaller portion of the workload. Unrolling scalar code produces an average speedup of 2 for 10 kernels with only a marginal average speedup of 1.1 for the remaining 4 kernels. In contrast, the vectorized code showed an average speedup of 7.5 for 7 kernels, assuming no improvement over the other half of the kernels. Although this could be interpreted as evidence against the overall effectiveness of vector architectures, the argument could be also be used against superpipelined architectures; that is, seven kernels, *half* the workload, could be improved by a factor of three if vector hardware were to be added.

In addition, the improvement in vector performance in this analysis is somewhat limited by less mature vector-compiler technology and by the Cray-1S implementation, an old vector processor by today's standards with limited chaining capabilities and only one memory port; hence even more improvement could be expected with modern implementations. Finally this data is further evidence that more parallelism is available in vectorizable loops — loops with no self-dependent statements — and limited in non-vectorizable ones.

In summary, using superpipelined hardware for scalar program fragments improves performance by a factor of 1.6 to 2 depending upon the program, while using vector hardware improves vectorizable program fragments by a factor of about 8. To compare the improvement in performance of an entire program when using vector-only hardware and vector hardware combined with superpipelined hardware, I use the following variation of Amdahl's Law

$$\frac{1}{\frac{1-f}{S} + \frac{f}{8}}$$

where f is the percentage of vectorizable code executed by a program, and S is the speedup provided by superpipelined hardware. Vector-only hardware has $S = 1$, while combined hardware has $S = 1.6$ or $S = 2$ depending upon the program. The following table lists the program speedups for a range of values for f and S:

f	$S = 1$	$S = 1.6$	$S = 2$	
0.0	1.0	1.6	2.0	(scalar-only code)
0.2	1.2	1.9	2.3	
0.4	1.5	2.3	2.8	
0.6	2.1	3.1	3.6	
0.8	3.3	4.4	5.0	
1.0	8.0	8.0	8.0	(vector-only code)

This table show how effectively superpiplined hardware in combination with a vector architecture dampens the negative effect of Amdahl's Law when the percentage of vectorizable code is less than 80%.

3.3 Software Advantages of Vector Architectures

In academia, an architecture is often judged only by its hardware and performance. In practice, however, a commercially successful architecture also depends on other aspects that are more software-oriented, such as ease of use and whether a program can execute on implementations which differ in cost but implement the same architecture. Commercial success depends on such issues because they affect how many people can use an architecture. In this regard, a vector architecture holds advantages over a superscalar one, although these advantages are not as easily quantifiable as the advantages in hardware and performance.

First, vector compilation technology is mature, having been in development since before the announcement of the Cray-1 in 1976 [101]. Moreover, if a vector processor were to be used in an MPP, mature compilers and a well-established user community that already knows how to productively use such processors will allow researchers in the compiler and applications community to concentrate on the more important issue of how to efficiently distribute a workload across a large number of processors. By contrast, compilation techniques for superscalar architectures are still in the research and development phase [92]. Using a superscalar processor in an MPP would have the additional burden of developing good compilers for the processor. Although superscalar compilation techniques may be able to extract parallelism from non-vectorizable program fragments, it is unclear that there is much parallelism to extract in such programs (as discussed in the previous section).

One area of concern about vectorization is that it typically takes longer than basic scalar compilation; however, this will also be true for superscalar compilation. A major difference between vectorization and basic scalar compilation is that the former includes a dependence-analysis phase that determines what operations can execute in parallel without changing the functionality of the program. Because dependence analysis must be part of any compiler that generates instructions to execute operations in parallel, superscalar compilation will also include this phase.

A second advantage of vector architectures is that the concept of the vector instruction is easily understood by a wide range of people from high-level language programmers to hardware designers. This conceptual simplicity reduces the chances of implementation errors at the hardware and compiler levels. A simple abstraction model for expressing parallelism will become increasingly important as systems with more parallelism become available. Such simplicity is also advantageous for the end-user who must use not only the computer but also the software that makes the computer easier to use [25]. If a compiler is not yet able to produce vectorized code, the user could still resort to using assembly language with vector instructions and still be able to achieve some amount of parallelism, as did those who used the ETA computer at Florida State University [80]. This would be more difficult to accomplish with a superscalar architecture.

Finally, a vector architecture can provide binary compatibility across different hardware implementations with varying degrees of parallelism more or less as easily as a superscalar architecture does, depending upon the type of compatibility. Binary compatibility allows a program to execute without recompiling on a range of processor implementations that vary in cost and performance. I consider binary compatibility to be a software advantage for an architecture because it minimizes the impact that changes in the hardware can have on existing compiler and application software. Binary compatibility is also a way to amortize the cost and development of a big VLSI chip over a large consumer base by increasing the potential market at both the high and low end of the cost/performance range. There are, in fact, three types of binary compatibility to consider:

- *upward compatibility*, which allows a program compiled for a vector architecture to execute on vector processors with varying degrees of datapath parallelism;
- *scalar compatibility*, which allows a program compiled for a vector architecture to execute on a scalar processor; and
- *backward compatibility*, which improves the performance of a program compiled for a scalar architecture (typically one that already exists) when the program executes on a vector processor.

Upward compatibility allows an architecture to quickly take advantage of improving technology and to increase its cost/performance range at the higher end. A program compiled with vector instructions can be executed on vector implementations with a varying number of functional units because the mapping of a vector instruction to a particular functional unit is part of the hardware implementation and not the instruction set architecture. For architectures that support fine-grain parallelism, more transistors on a single chip, as the result of improving VLSI technology, allows support for greater amounts of parallelism. As I have already discussed in the first part of this section (when I compared the hardware expense of a vector architecture with that of a superscalar one), a vector architecture is not only upwardly compatible with increasing amounts of datapath parallelism, but it also provides this capability at less cost than does a superscalar architecture.

Scalar compatibility increases the range of an architecture at the lower end of the cost/performance scale. Because a scalar instruction is equivalent to its corresponding vector instruction that executes one operation, scalar compatibility can be provided by making each vector register have one element each, although good engineering is needed so that such an implementation has acceptable performance for a vector length of one. In such an implementation, the vector register file becomes in essence a second scalar register file. Viewed this way, scalar compatibility in a vector architecture is easily provided if stripmining is entirely supported by the hardware, as it is in the IBM 3090 vector architecture in which binary-compatible implementations can have different lengths of vector registers [16]. Because scalar compatibility is an issue when cost is more important than performance, the need to provide a scalar-compatible implementation will lessen as larger chips become less costly.

Backward compatibility allows a new implementation to improve the performance of so-called "dusty-deck" programs that have been compiled for a scalar architecture. The motivating factor for providing backward compatibility is to maintain the market share of an already existing architecture that has a large software base that is not likely to be recompiled. To accomplish the same effect as recompilation, which rearranges the execution order of operations to allow parallelism to occur, backward-compatible hardware uses dynamic scheduling, also known as out-of-order instruction-issuing. To find instructions that can overlap in execution, hardware for dynamic scheduling must perform, in each clock period, pairwise checks for dependences among several instructions, in a manner similar to what is done when issuing instructions in a superscalar architecture. However, dynamic scheduling for backward compatibility requires instruction-issue logic that is more complex than that of a superscalar architecture because, to find enough instructions to issue *without* *recompilation*, hardware for dynamic scheduling must examine more than the number of instructions that are actually issued per clock period.

In fact, Wall's study on parallelism provides data showing that hardware for dynamically extracting parallelism is extremely expensive for a relatively small gain in performance. Wall's data, given in more detail in Figure 3.13, shows that a constant increase in the number of instructions issued per clock period requires an exponential growth in the number of instructions examined. For example, to issue an average of 3 instructions per clock period, an average of 4 instructions must be examined per clock period. Doubling the number of examined instructions to 8 produces, on average, only 3 to 5.5 instructions that can issue each clock period. Continuing to double the number of examined instructions finds, at best, one more instruction to issue for programs with little intrinsic parallelism (< 8) and 3 to 11 more instructions for programs rich in parallelism. The hardware expense of dynamic scheduling is actually in the hazard checks made for each possible pair of examined instructions. Hence, because the number of pair-wise hazard checks is proportional to the square of the number of examined instructions, a constant increase in datapath parallelism requires an exponentially-squared increase in the number of pair-wise hazard checks.

Unless the market share for dusty-deck programs is an overriding concern, providing backward compatibility hardly seems like a good cost/performance feature in any architecture, even for programs whose intrinsic parallelism is plentiful. If the main purpose of an architecture is to support fine-grain parallelism, it is best to begin with a new architecture, thus making backward compatibility less of an issue. This is, in fact, what most commercial superscalar implementations have done, with the notable exceptions of Sun's SuperSPARC and possibly a future implementation of the Intel i386, architectures that clearly have a high investment in market share. However, if absolutely necessary, the superpipelined extension of a combined vector and superpipelined architecture could provide backward compatibility, although the simplicity of the instruction-issue logic would be gone because of the many hazard checks required for dynamic scheduling.

3.4 Summary

In this chapter, I presented arguments for why a vector architecture combined with superpipelined hardware is more appropriate for supporting fine-grain parallelism than is a superscalar architecture with respect to hardware, performance, and software issues. Although either architecture could be implemented on a single VLSI chip, much work is focused on superscalar architectures but little attention is being paid to vector architectures as a viable VLSI design. This is because many designers, in part, mistakenly believe that a vector processor is expensive to implement and is effective for only a small set of programs.

I presented data showing that, in fact, when supporting the equivalent amount of datapath parallelism, a vector architecture is no more expensive than a superscalar one and, for some features, is even less costly. One feature that is needed by either architecture is a high-bandwidth memory system because both have a high memory demand. A high-performance cache system could be used for a vector processor as a more cost-effective alternative to an expensive, large, highly-interleaved memory, although further research is

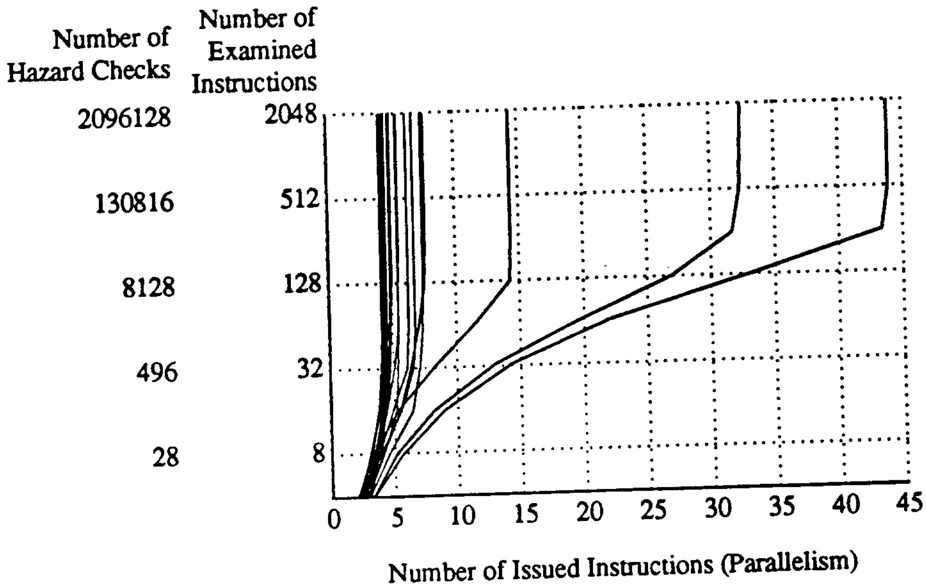


Figure 3.13: Number of Instructions Issued versus Number of Instructions Examined

Based on Wall's parallelism data [115], this graph shows the number of instructions that are examined and the number of hazard checks that are performed *each clock period* in order to find a given number of independent instructions to issue. To show the extent to which the number of examined instructions determines the amount of parallelism under optimistic conditions, I use the computational model that has unlimited hardware resources for branch/jump prediction, 256 registers, and perfect dependence analysis (Wall's *Good* model). The following table summarizes the above graph to emphasize that both the number of examined instructions and the number of hazard checks grow non-linearly for a linear growth in the number of instructions issued per clock period:

NUMBER OF ISSUED INSTRUCTIONS		NUMBER OF EXAMINED INSTRUCTIONS	NUMBER OF HAZARD CHECKS
programs with parallelism < 8	programs with parallelism > 8		
2.0–3.0	2.0–3.0	4	6
3.0–4.5	3.0–5.5	8	28
4.0–6.5	5.0–9.0	16	120
4.0–7.0	8.0–14	32	496
4.0–7.0	11–22	64	2016
4.0–7.5	14–33	128	8128
4.0–7.5	14–43	256	32640

needed to determine an appropriate cache organization. Another feature that is comparable in cost is the register file. Although vector architectures typically use many more registers than do superscalar ones, the area increase of a vector register file is not comparable to its increase in number. In fact, doubling the number of superscalar registers to 64, as Wall's data suggests will be necessary, makes a superscalar register file with 5 read-ports and 3 write-ports comparable in area to a vector register file similar to the one in the Cray Y-MP with 512 registers. A vector architecture also has simpler instruction-issue logic. Finally, these differences in cost favor the vector architecture as hardware designers increase the amount of datapath parallelism in response to further advances in VLSI technology.

I also analyzed data from Wall's parallelism study showing that vectorizable program fragments have copious quantities of parallelism and are, furthermore, most likely to be the more time-consuming programs in a workload [115]. Because of these characteristics, about 25 times as many instructions could be executed if the hardware were to make full use of the intrinsic parallelism in Wall's workload. To lessen the effects of Amdahl's Law, I showed that superpipelined hardware is effective at handling non-vectorizable program fragments and that additional vector hardware provides three times more performance on vectorizable kernels.

Finally, I discussed some of the software advantages of a vector architecture: mature compiler technology, the vector instruction as a simple, elegant abstraction for expressing parallelism, and binary compatibility. These software advantages facilitate use of a vector architecture across a range of implementations that differ in cost. Although often overlooked by academics, in part, because their effects are difficult to quantify, such issues are important to the commercial success of an architecture because they affect the number of people that can use an architecture.

In summary, although superscalar architectures appear to be the current design of choice, I believe that a vector architecture is more suitable in computers, such as workstations and massively parallel processors, that rely heavily on VLSI technology. Vector architectures work effectively for vectorizable programs which have an abundance of parallelism, while non-vectorizable programs appear to contain meager amounts of parallelism. If superscalar architectures are to perform as well as vector ones, contrary to popular belief, the hardware implementation of a superscalar architecture can be more expensive than that of a vector one. In other words, if the main purpose of an architecture is to support fine-grain parallelism, a vector architecture is a better choice than a superscalar one because of the simplicity of a vector architecture's hardware, its natural match to programs rich in parallelism, and its established compiler and application communities.

Chapter 4

Common Experimental Framework

This short chapter describes the common experimental framework — the basic vector hardware, the performance tools, and the workload — that I use in the following two chapters, in which I evaluate the performance impact of changes in a vectorizing compiler and in vector hardware. Other aspects of the experimental framework, such as performance criteria and methodology, differ for the studies I carry out and, hence, are described in the chapter for their respective study.

4.1 Processor Description

The hardware basis for my dissertation is the processor of the Cray Y-MP, which was first announced in 1988. A fully-configured Y-MP computer contains eight processors; the "MP" in the name stands for "multiprocessor." The processor itself is a load/store, superpipelined, vector architecture. The deep pipelines plus the use of rather expensive, bipolar technology result in an extremely high clock frequency: 167 MHz, or equivalently, a 6 ns clock period. As a point of reference, in 1991, most microprocessors have a clock frequency between 25 and 40 MHz with the higher performance ones having 63 MHz (the Hewlett-Packard Snake) and 100 MHz clocks (the MIPS R4000). Following are details on the organization of the registers and functional units that are relevant to my thesis. Other details about the Y-MP processor are available in the Cray Y-MP Computer Systems Function Description Manual [24].

Figure 4.1 lists the register files in the Y-MP processor. The vector register file, which is the focus of this thesis, can be viewed as a partitioned one (see Section 2.2.2) in which each vector register is comparable in organization to a scalar register file. A vector register consists of 64 dual-ported registers that are attached to read and write buses common to the vector register. Because of the separate read and write buses, chaining is possible between any vector instructions. For my thesis, I examine different configurations of the vector register file. In Chapter 5, *Register Usage and Instruction Scheduling*, I experiment with the number of vector registers, and in Chapter 6, *Bus Usage and Register Assignment*, I explore the implications of having more than one vector register share a set

REGISTER FILE	WIDTH	ORGANIZATION	TOTAL NUMBER OF BYTES	FUNCTION
A	32 bits	8 registers	32 bytes	store addresses or integer data
B	32 bits	64 registers	256 bytes	back-up for A register file
S	64 bits	8 registers	64 bytes	store integer/FP scalar data
T	64 bits	64 registers	512 bytes	back-up for S register file
V	64 bits	8x64 registers	4096 bytes	store integer/FP vector data

Figure 4.1: Register Files of the Cray Y-MP Processor

This table shows how the five register files of the Y-MP processor vary in size, organization, and functionality. Note that data and address words differ in size: data are 64-bits and addresses are 32-bits. Because of the limited capacity of the the A and S register files, the back-up register files, B and T respectively, serve as temporary storage that is faster to access than main memory. All the register files are connected to memory ports, which are functional units that serve as the interface between the Y-MP processor and its memory system. The register files, A, S, and V, are also connected to the other functional units, while the back-up ones are not.

of read and write buses.

There are also individual registers for special purposes. The vector length register VL specifies the number of operations that a vector instruction is to execute. The maximum number is 64, which is the number of elements in a vector register. The vector mask register VM is a 64-bit register that is set and used by special vector instructions for conditional selection of data. For example, the instruction V0<-VM?V1:V2 (written with C-like syntax) transfers the i^{th} element of V1 to the i^{th} element of V0 if the i^{th} bit of VM is equal to 1; otherwise the i^{th} element of V2 is transferred. The VM register and its associated instructions permit some loops with conditional statements to be vectorized.

The Y-MP processor has nine special-purpose functional units:

- two load ports
- a store port
- a floating-point adder
- a floating-point multiplier
- a floating-point reciprocal unit
- an integer unit
- a logical unit
- a shifter

The logical unit is used in conjunction with the VM register for conditional selection. A second logical unit is optional but because the simulator does not model this, I choose to ignore it. Division is computed using the reciprocal unit and the multiplier in four steps (one reciprocal approximation and three multiplications).

The Y-MP processor is rich in memory bandwidth with a total of three memory ports, whereas most processors have only one bi-directional memory port. Furthermore, the Y-MP processor has *gather/scatter* hardware that provides a vector version of indexed addressing to allow nested array references to be vectorized. For example, to load the data specified by the array reference `A(I(K))` into vector register `V1`, only two vector memory instructions are executed:

```
A0 <- base address of array I()
A1 <- base address of array A()
V0 <- M[ A0 ]
V1 <- M[ A1+V0 ]
```

To compute the effective addresses for the second memory instruction, gather/scatter hardware adds the elements in the vector register `V0` to the base address in register `A1`. *Gather* refers to memory loads that use a vector register as an index register whereas *scatter* refers to stores.

These functional units are fully-pipelined so that a new operation can begin executing every clock period in each functional unit. Pipelined memory accesses are provided by an interleaved memory system. However, because individual memory banks are not pipelined and have an access time that is greater than one clock period, memory operations that reference the same bank take longer to execute when they are performed in rapid succession. Such access conflicts to a memory bank cause execution delays in the vector memory instructions that generated the references, preventing these instructions and any vector instructions chained to them from achieving full pipelined execution.

Although functional units can execute simultaneously, there are some restrictions on the simultaneous use of the memory ports when using gather or scatter instructions. Even though a gather instruction and a scatter one use different memory ports, only one of these can occur at a time. However, a gather can occur in conjunction with a simple load or store, and a scatter can execute in parallel with one or two loads.

4.2 Performance Tools

To generate the raw data used for my performance studies, I use modified versions of Cray Research's production FORTRAN compiler, which is named *cft77*, and the Cray Y-MP simulator. Figure 4.2 illustrates the relationship of these two tools. Both tools are parameterized so that the number of vector registers can vary up to a maximum of 64.

The *cft77* compiler is a vectorizing one, and performs global and local optimizations on both the scalar and vector code that it generates. I am specifically interested in the instruction scheduling and register assignment phases. An instruction scheduler determines an appropriate order in which operations in a dependence graph execute and which preserves dependences among the operations, and a register assigner determines which register stores the value produced by an operation. In the *cft77* compiler, the scheduling phase occurs before the assignment phase, a sequence that I assume when describing algorithms for these

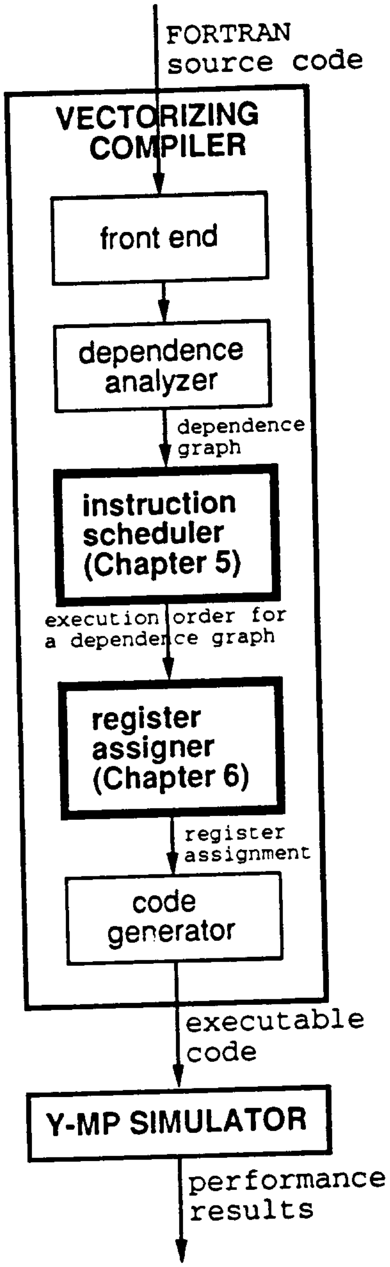


Figure 4.2: Performance Tools

This figures shows how I use Cray Research's vectorizing compiler, called *cft77* and Y-MP simulator in my performance studies. Both tools have been modified so that I can specify the number of vector registers to use, up to a maximum of 64. The instruction scheduler and register assigner are highlighted because these are the phases that I will concentrate on in this dissertation. Although not explicitly shown, inputs to earlier phases are also inputs to later phases. In other words, input to a phase is augmented with more information, all of which is passed on to the next phase.

two phases. Consequently, input to the instruction scheduler is a dependence graph, which is generated by the dependence analyzer, and input to the register assigner is an execution order for that dependence graph.

The simulator emulates every aspect of a Y-MP processor and can keep track of simulated execution time. The behavior of the instruction buffer is accurately modeled. Only simple, memory-bank conflicts, such as those occurring within one stream, are taken into account. Memory conflicts between two independent reference streams are ignored.

4.3 Workload

In addition to performance tools, an appropriate workload is needed for my performance studies. Because my research experiments explore aspects of vector design, I use a set of 36 vectorizable loops that was collected at Cray Research, Incorporated for use by their architects to evaluate future designs. These loops, which I collectively call "the CRI workload," have been extracted from actual applications used by Cray customers and are written in FORTRAN. They include kernel 7 and the second loop of kernel 18 from the Livermore Loops [84] plus several loops extracted from the Perfect Club benchmark suite [12]. In addition, these loops contain several program constructs that are traditionally considered difficult to vectorize. Examples are scalar reductions, array references with nested indices, conditional statements, and calls to intrinsic functions.

Seven of the 36 loops in this workload consist of more than one basic block. One reason for this is that a loop containing a conditional `IF...THEN...ELSE` statement has at least three basic blocks. For example, one loop contains nine `IF...THEN` statements resulting in 27 basic blocks. Vectorizing scalar reductions also produces multiple basic blocks: one for computing a vector of practical sums and another to calculate the final sum. (Section 2.3.1 describes the software transformation for computing a scalar reduction using vector instructions).

In the *cft77* compiler, each basic block is represented by a dependence graph in which a vertex is a vectorizable operation and there is an arc from one operation to another if the first operation produces a result used by the second one. A vectorizable operation is eventually converted into a vector instruction in the code-generation phase of the compiler. The vectorizable operations in this workload comprise a mixture of floating-point and integer operations. Because some loops contain more than one basic block, there are in fact 88 dependence graphs for the 36 loops.

An important characteristic of these loops is the substantial variance in the number of vectorizable operations, which, in turn, results in a wide range in the execution times for one iteration. Figure 4.3 illustrates this diversity. Two-thirds of the loops contain more than 30 vectorizable operations, and the execution time for one iteration ranges from 2 clock periods to about 300 clock periods. This considerable variance is important because, in all likelihood, the larger loops will have more parallelism and hence require more registers. If the workload consisted of loops with less than 30 operations, I could erroneously conclude that eight vector registers is sufficient for the Cray Y-MP functional unit configuration.

Figure 4.3 also shows that, with the exception of one loop, more than one vectorizable operation is executed per clock period, demonstrating that a vector architecture

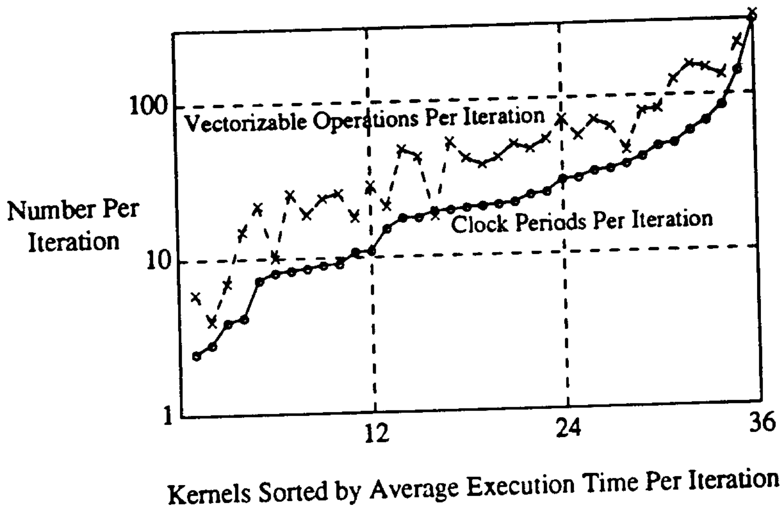


Figure 4.3: Vectorizable Operations and Execution Time of the CRI Workload

This graph, which uses a log scale for the Y-axis, shows the immense range in the number of vectorizable operations and, hence, the execution times of the loops that I use for my evaluation studies.

The number of vectorizable operations for a loop is an indication of the *minimum* amount of work that is executed each iteration. In addition to vectorizable operations, scalar instructions for address and branch computations are executed each iteration. Sometimes there are extra vector instructions to handle register spilling or extra instructions for executing any intrinsic functions such as square root. None of these extra instructions nor the scalar instructions are included in the count of operations plotted above.

The execution time of a loop as plotted above is the average time to execute one iteration of a loop as compiled by *cft77*, Cray Research's vectorizing FORTRAN compiler, for the Cray Y-MP using eight vector registers. This average, which is calculated as the time to execute the entire loop divided by the number of iterations executed, includes any time spent executing loop set-up instructions, strip overhead, or intrinsic functions.

does make use of fine-grain parallelism. The average-per-iteration time for the one loop (number 16 in the graph) includes computing a square root whose instructions are excluded from the count of operations. In fact, because the average-per-iteration times for all the loops includes the execution time for instructions other than the vectorizable operations, the amount of parallelism capitalized on by the Y-MP vector architecture is greater than what is illustrated in this graph.

4.4 Summary

In summary, with the cooperation of Cray Research Incorporated, I have access to benchmarks, a production vectorizing compiler, and a simulator. In the following two chapters, I will modify the Cray compiler and use the benchmarks and simulator to evaluate the performance impact of changes in the Cray Y-MP vector processor and compiler.

Chapter 5

Register Usage and Instruction Scheduling

In Section 2.2 (of Chapter 2, *Fundamentals of Vector Architectures*), I outlined the general hardware requirements needed for supporting fine-grain parallelism. In particular, I stated that both multiple functional units and an appropriate organization for a register file are equally important for allowing parallelism to occur. In support of this statement, I presented data from an independent study showing that increasing the number of registers from 32 to 512 increases the amount of achievable parallelism, which results in better utilization of 64 functional units (see Section 3.2 of Chapter 3, *A Case for Vector Architectures*). Hence, the number of registers must be balanced with the number of functional units, if enough parallelism is to occur to use the hardware efficiently. If there are too few registers relative to the number of functional units, the functional units will not be used to their fullest potential. If there are too many registers, the functional units will be effectively used but the register file will be over-designed. In other words, a hardware designer needs to know the minimum number of registers required to use a given number of functional units effectively across a range of programs. Determining this number requires a study that examines both the performance and cost of implementing a given number of registers.

In this chapter, I focus primarily on the performance aspect of implementing vector registers in the Cray Y-MP vector architecture, and defer the cost analysis to the next chapter. There are 8 vector registers connected to 9 special-purpose functional units in the Y-MP processor. Because I want *both* of these components to be well utilized, I begin my investigation by asking "Would more vector registers significantly improve performance?" and if so, "How many more vector registers are needed before performance no longer improves?" These questions form the primary goal of this study, which is to determine the minimum number of vector registers that can effectively use the 9 special-purpose functional units in the Cray Y-MP vector processor. To produce the desired results, I need to also demonstrate that the instruction-scheduling algorithm, which is used in the code-generation phase of a compiler, has a major impact on the performance of the generated code. This secondary goal was not obvious at the outset of this study and became apparent only after I had analyzed some preliminary performance data.

For this study, I vary only the number of vector registers, leaving the *maximal* *vector length*, which is the number of elements per vector register, fixed at 64. Vector length, which is the amount of unrolling provided by stripmined code (described in Section 2.3.2, pages 24 to 29), is used to hide both the latency of vectorizable operations and the execution of scalar operations, an effect that is more dominant when the number of iterations executed for a loop is greater than the maximal vector length. A maximal length of 64 is sufficiently long to effectively hide the latencies of the operations implemented in the Cray Y-MP. Shortening vector length to be less than 64 is likely to either not affect performance or even decrease it because operational latencies can no longer be effectively overlapped with the execution of other operations. Increasing vector length beyond 64 is unlikely to improve performance. Furthermore, in order to be effective, a longer vector length would require more iterations to be executed for a loop, a factor which is influenced more by an application rather than a compiler. On the other hand, increasing the number of vector registers does improve performance significantly as I will demonstrate in this chapter. Moreover this performance improvement, although somewhat dependent upon characteristics of an application, is also influenced by a compiler's algorithm for scheduling instructions.

The version of the *cft77* compiler used for this chapter is the one that was available to me during my work term at Cray Research, Incorporated in the fall of 1990. Since that time, a newer version of *cft77* has been released that uses a scheduling algorithm similar to the one I developed [62]. Nonetheless, for the sake of brevity, I use the term "*cft77*" interchangeably with the phrase "the 1990 version of *cft77*."

To explain how I formulated the goals for this investigation, I show how more vector registers can improve performance and present some initial performance data that suggest a scheduling algorithm different from the one used in the *cft77* compiler is needed to use more registers[1] effectively. Then, after describing the performance criteria and methodology I use to perform my experiments, I contrast the scheduling algorithm used in the *cft77* compiler with an algorithm that I developed and that is a variant of list scheduling. Finally, I present a set of performance data showing that the scheduling algorithm does have a major impact on performance and another set of data that determines a cost-effective number of registers for the Cray Y-MP vector processor.

5.1 More Registers and A Different Scheduling Algorithm

In this section, I show how the number of vector registers and the scheduling algorithm affect performance. First, I work through an example to show how more registers improve performance and use this example as the inspiration for the primary goal of this study. Additionally, this example demonstrates how determining the appropriate balance between the number of registers and functional units involves understanding the interaction among vector hardware and two aspects of the code-generation phase of a compiler, the instruction scheduler and the register assigner. In the second part of this section, I present an initial performance study whose apparently conflicting results inspire the secondary goal of this study.

[1] Because the scheduling algorithms described in this chapter can treat vector registers and registers analogously, I use the terms *vector register* and *register* interchangeably.

5.1.1 Why More Registers?

Because using more registers requires increasing the number of register cells in hardware, it must be justified by a significant reduction in execution time. Using more registers reduces execution time in two ways. First, more registers allow more aggressive scheduling which, in turn, allows more parallelism to occur. More parallelism causes more intermediate results to be generated at any one time and hence, more registers are needed to hold these intermediate results. Reducing execution time in this way also allows the functional units to be used more frequently.

A second way that more registers reduce execution time is to reduce the number of *register spills* generated by a compiler. Because the number of results can outnumber the number of registers available in hardware, a compiler generates extra memory instructions when all registers are used to save the contents of a register so that another value can be stored in it. In a scalar architecture, executing these extra instructions will increase the execution time because only one operation can be executed during each clock period. With a vector architecture, in contrast, these extra instructions are, in fact, vector instructions that save and restore the contents of vector registers. However, in contrast to a scalar architecture, the time to execute these extra instructions can be overlapped with the original instructions. Reducing register spills also lessens the demand on memory, but this is less of an issue when an abundance of memory bandwidth is provided in the implementation, as is the case for most vector architectures.

In short, reducing the need for register spills has minimal impact on execution time in vector architectures, and the better reason for adding more registers is to allow more parallelism to occur. Later in this chapter, I present data that supports both of these claims.

To demonstrate how register usage affects how much parallelism occurs, I use the vectorizable loop in Figure 5.1, which is represented by the dependence graph also shown in that figure. Once the vectorizable operations of a loop are identified, a vectorizing compiler determines an order in which these operations can execute. (Although this order corresponds to an instruction sequence, I use the term *execution order* to emphasize the fact that these are still operations that are eventually translated into instructions. Moreover, I prefer the term *execution order* to *evaluation order*, which is more commonly used by the compiler community [2], because hardware does not evaluate operations but instead executes them.) Any execution order is permissible as long as the loop's functionality does not change, which will happen if the data dependences among the vectorizable operations are preserved. In other words, a correct execution order is one in which all ancestors of an operation in a dependence graph are executed first.

Although a dependence graph of a loop specifies a partial order for correct functionality, an enormous number of execution orders satisfy that partial order. To take a somewhat trivial example, the first seven operations of a correct execution order for the dependence graph in Figure 5.1 could be the seven loads. Because these loads do not depend on any other operations, they can be executed in any order, which results in at least 7! or 5040 different correct execution orders for the first 7 operations alone. All of these different execution orders are *exactly* equivalent in functionality because the dependence graph, in addition to specifying a partial order, also specifies how the *results* of the operations are

`DO 40 I=1,N`	
`   RA = W(I,1)+X(I,1)`	$LOAD_1$ $+_2$ $LOAD_3$
`   RB = W(I,2)+X(I,2)`	$LOAD_4$ $+_5$ $LOAD_6$
`   RC = W(I,3)+X(I,3)`	$LOAD_7$ $+_8$ $LOAD_9$
`   FS(I) = SX(I)*RA*RB*RC`	$STORE_{10}$ $LOAD_{11}$ $*_{12}$ $*_{13}$ $*_{14}$
`40 CONTINUE`	

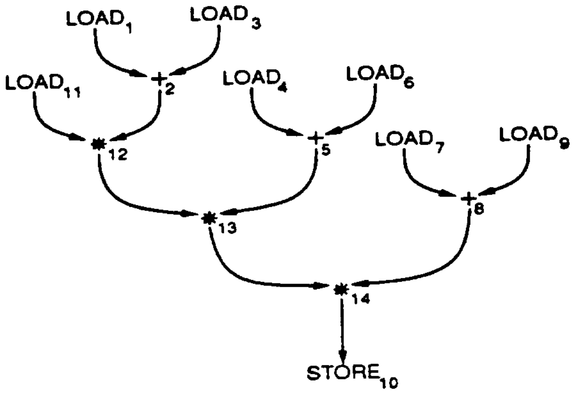


Figure 5.1: Source Code and Dependence Graph for Sample Loop

This figure presents the vectorizable loop that I use extensively in this chapter and in the next one to motivate the studies I perform. This loop is a modified version of one from the CRI workload. On the left is the FORTRAN source code and on the right are the corresponding vectorizable operations for each FORTRAN statement. Each operation is identified by the type of operation it executes and a unique number as a subscript to the operational type. At the bottom is a dependence graph that shows the data dependences among the vectorizable operations. A vertex in such a graph is a vectorizable operation, and an arc is a dependence where the direction of the arc is from the producer of the dependent value to the consumer of the value. In Chapter 2, *Fundamentals of Vector Architectures*, I discuss data dependences in greater detail and also describe how a vectorizing compiler identifies vectorizable operations and constructs a dependence graph.

to be combined. In contrast, Figure 5.2 shows two dependence graphs that are, in theory, functionally equivalent but which combine the results of operations in different orders. This figure also explains why, in practice, such dependence graphs are not exactly equivalent.

There are two main differences between all the correct execution orders for a dependence graph:

1. the time needed to execute the order, and
2. the minimum number of registers needed to execute the order without having to spill registers.

For example, Figure 5.3 shows two correct execution orders for the dependence graph in Figure 5.1. Because all the dependence arrows point downward, both these orders satisfy the partial order specified by the dependence graph. Nonetheless, these two orders differ in their execution times and their minimal register requirements.

To demonstrate that the execution times of the two orders in Figure 5.3 are different, I use a technique called *chime counting* to provide a quick estimate of execution time. A *chime*, which originally was an abbreviation for "*ch*ain t*ime*",[2] is a unit of time that is approximately equal to the time it takes to execute one vector instruction. Instructions that use different functional units can execute in the same chime in the absence of any access conflicts among registers. For example, for the execution orders in Figure 5.3, the first two loads and the addition execute in the same chime because the Cray Y-MP has two load ports and chaining hardware. Conversely, instructions that use the same functional unit must execute in different chimes. For example, the fourth operation (LOAD_4) must execute in the second chime because the load ports are already each executing a vector load instruction.

Continuing in this fashion, we see that the execution order on the left executes in 6 chimes, and the one on the right executes in 4 chimes. Because each operation of a vector instruction corresponds to an iteration of a loop (see Section 2.3 in Chapter 2, *)*, executing a loop with vector instructions in t chimes corresponds to executing one iteration of that loop in approximately t clock periods. Chime counting, in fact, provides an optimistic estimate of the per-iteration execution time because the temporal impact of loop- and strip-overhead is ignored. To verify that this estimate is reasonable, I executed these two orders on a Cray Y-MP for 100 iterations each; the per-iteration time of the order on the left is 7.5 clock periods, and that on the right is 5.7 clock periods. This is a difference of about 2 clock periods as predicted by chime counting.

In addition to differing in their execution times, these orders also differ in the minimum number of registers needed to execute the operations without spilling registers. To demonstrate this, I must first explain how a compiler uses registers and then how to determine the minimal register requirement of an execution order. A compiler uses a vector register to store a vector of values produced by a vectorizable operation. If a compiler assigns each result to a different register, too many registers will be used; fewer registers could be used without increasing the execution time predicted by chime counting.

[2] My thanks to James E. Smith for this etymological fact.

```
DO 40 I=1,N
   RA = W(I,1)+X(I,1)
   RB = W(I,2)+X(I,2)
   RC = W(I,3)+X(I,3)
   FS(I) = {{ {SX(I)*RA} *RB} *RC}
40 CONTINUE
```

```
DO 40 I=1,N
   RA = W(I,1)+X(I,1)
   RB = W(I,2)+X(I,2)
   RC = W(I,3)+X(I,3)
   FS(I) = {SX(I)*RA} * {RB*RC}
40 CONTINUE
```

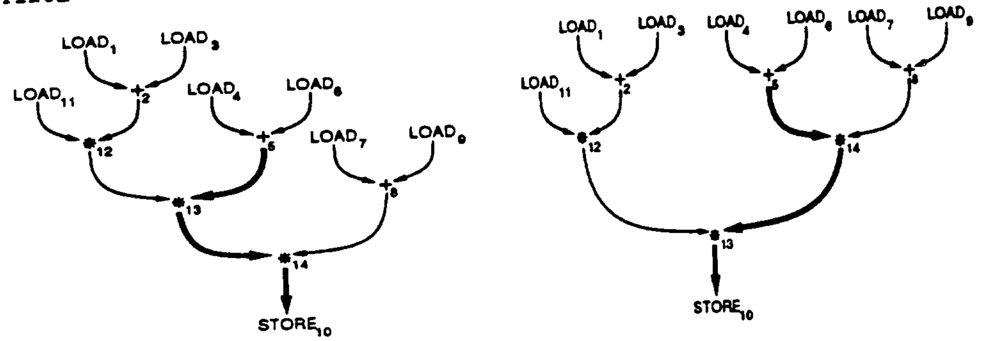


Figure 5.2: Two Dependence Graphs for the Sample Loop

These dependence graphs, both of which represent the loop in Figure 5.1, differ in how they combine the results of the multiplications, as indicated by the arcs that are highlighted in bold. This difference is explicitly shown with braces in the corresponding source code. Although these two dependence graphs combine the results in a different order, they are, in theory, functionally equivalent under the arithmetic law of associativity. In practice, however, the equivalence of such graphs is *not exact* because floating-point arithmetic does not guarantee associativity; instead, these graphs are considered equivalent within the limits of rounding error.

Because both graphs combine the results in different orders, they also specify different partial orders and, hence, different sets of correct execution orders. For example, $*_{12}*_{13}*_{14}$ is the only order in which the three multiplications can be placed in any correct execution order for the dependence graph on the left, and $*_{12}*_{14}*_{13}$ or $*_{14}*_{12}*_{13}$ are the only orders for the dependence graph on the right.

By taking as input the source code shown in Figure 5.1, which does not have parentheses to explicitly group operators, a FORTRAN compiler would produce the dependence graph on the left, in accordance with FORTRAN semantics, which specify that a series of operators of the same class is grouped from left to right. On the other hand, an optimizing compiler may produce the dependence graph on the right to increase the amount of parallelism available; two multiplications can be executed in parallel using the dependence graph on the right, whereas all three multiplications are executed sequentially using the dependence graph on the left. There is little point in generating the more parallel graph, however, unless the hardware includes at least two multipliers to take advantage of the extra parallelism. Although more than one dependence graph can represent a loop, for my study, I use those dependence graphs that most closely follow the arithmetic conventions of the FORTRAN language.

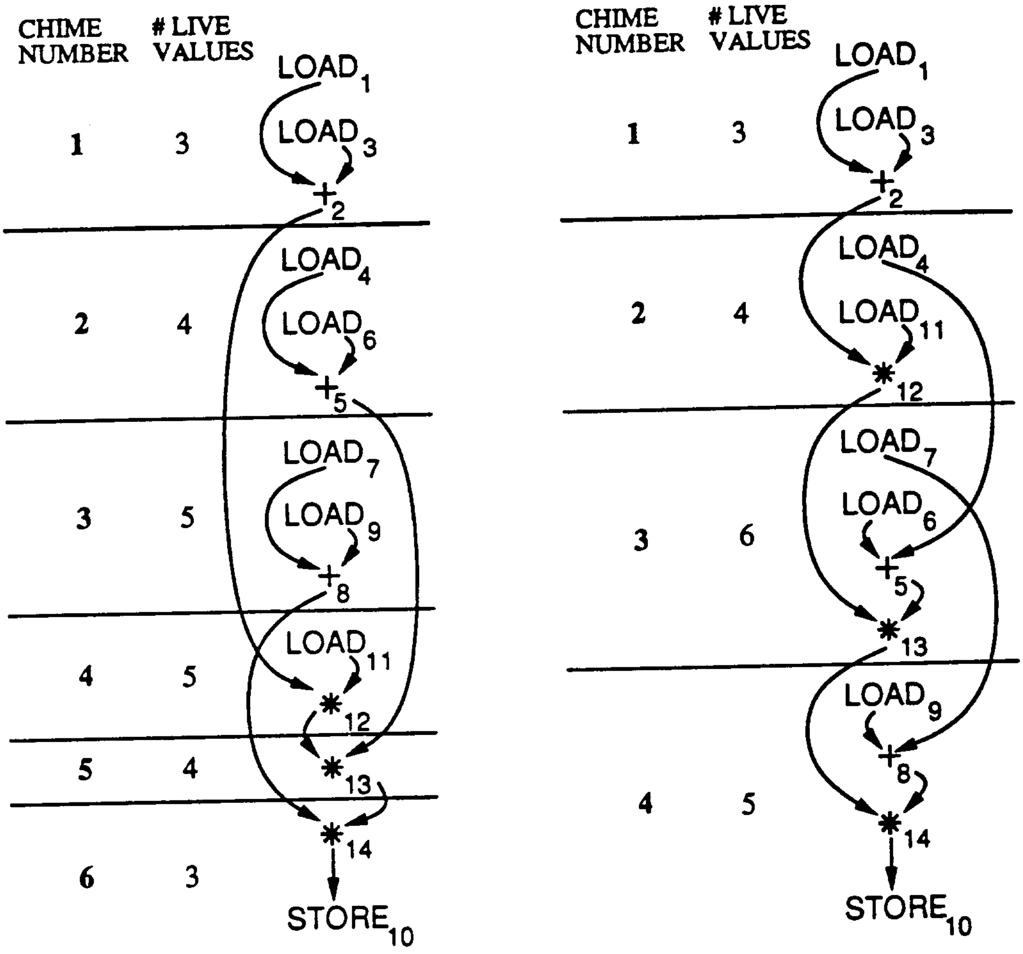


Figure 5.3: Two Execution Orders for the Example Loop

These are two execution orders that satisfy the partial order specified by the dependence graph in Figure 5.1. For each order, I have shown which operations execute in the same chime when the hardware has chaining, two load ports, one adder, one multiplier, and one store port. I have also listed, for each order, the number of values that are live in each chime. The lifetime of a value is indicated by the dependence arrow that connects the producer and the consumer of that value, and the number of live values in each chime is equal to the number of dependence arcs that appear in, or pass through, that chime.

To explain how, I consider the dependence graph of a single basic block. A value is said to be *live* from the time of its production to its last use. In Figure 5.3, the lifetime of a value is indicated by the dependence arrow that connects the producer and the consumer of that value. Two values that are live at *different* times can be assigned to the same vector register. For example, for both execution orders in Figure 5.3, the values produced by the operations LOAD_1 and LOAD_4 can be stored in the same vector register. In contrast, two values that are live at the *same* time, such as the values produced by the operations $+_2$ and LOAD_4 in Figure 5.3, must be stored in different vector registers to avoid generating extra instructions that would transfer the two values between memory and a shared register.

Although the execution of register-spill code, if done at a judicious moment, may *not* increase execution time, I assume, for the sake of simplifying this example, that it does. The impact of register spilling on execution time and register usage is taken into account later in this chapter in Section 5.4, the quantitative part of this study. To avoid generating any code for spilling registers and thereby increasing execution time, all simultaneously live values must be assigned to different registers. Hence, a compiler needs to use only a number of registers that is equal to the maximum number of simultaneously live values, a number which is called the *critical register quantity* by Eisenbeis, Jalby, and Lichnewsky [32]. This is, in fact, the *minimum* number of registers that can be used without spilling registers. For the purposes of this example, it is sufficient to know that this minimum is achievable. An assignment algorithm that is able to match this minimal requirement is described in Section 6.2.3 (on page 123 in Chapter 6, *Bus Usage and Register Assignment*).

Now that I've explained what is the minimum number of registers needed to execute an order, I can now show that the two orders in Figure 5.3 have different minimal register requirements. To do this, I must first know what values are live at the same time; these are the values that are used by operations executing in the same chime because operations in the same chime execute simultaneously. Thus, counting the number of live values in each chime reveals the maximum number of simultaneously live values. In Figure 5.3, the number of live values in each chime is equal to the number of dependence arcs that appear in, or pass through, that chime. Based on this method for determining minimal register requirements, the execution order on the left requires 5 vector registers to avoid generating spill code and that on the right requires 6 vector registers.

In addition to differing in execution time and minimal register requirements, these orders differ in two other respects. First, the execution order on the right exhibits more parallelism; there are two chimes in which four operations are executed, while at most three operations are executed in a chime in the execution order on the left. This is because both orders execute the same number of operations, but the one on the right executes in less time. The second difference is that the execution order on the right uses the functional units more effectively. In every chime, at least one load port is always used; in other words, during the execution of this loop using the order on the right, a load operation is initiated every clock period. In contrast, there is no single functional unit that is always in use when the order on the left is executed.

In summary, Figure 5.3 demonstrates how using more registers allows more parallelism to occur, which in turn reduces the time to execute a loop and results in more effective use of the functional units. In this case, the Cray Y-MP provides enough vector registers to accommodate the faster execution order. However, one example does not prove sufficiency in general. One goal of this study is to determine a cost-effective number of registers, which I believe is more than the 8 vector registers currently provided in the Cray Y-MP vector processor.

5.1.2 Why a Different Scheduling Algorithm?

To substantiate my hypothesis that more registers are needed to improve performance, I must determine the minimal number of registers required for maximal parallelism in each loop of the CRI workload. A minimal register requirement is associated with a particular execution order of a dependence graph, as was explained in the previous subsection, and an execution order is chosen by an instruction scheduler, which is part of a compiler. Hence, the algorithm used by an instruction scheduler affects both execution time and register usage of a loop. What is not obvious is *how much* a scheduling algorithm affects performance and register usage. In this subsection, I present two sets of apparently contradictory data that together suggest that the performance impact of a scheduling algorithm can be significant and that a scheduling algorithm different from the one used in the 1990 version of the *cft77* compiler is needed to prove my hypothesis.

The first set of data indicates that execution time could possibly be reduced by a significant amount. This data is based on a static lower bound for the per-iteration execution time of a loop. An important characteristic of this lower bound is that it is calculated using only the frequency of operational types in a dependence graph and the number and types of functional units in the hardware. This provides a method for quantifying the maximal improvement to performance without having to generate an actual execution order that achieves this improvement. Because a loop can consist of one or more dependence graphs (for example, vectorizable loops with conditional statements or scalar reductions), the static lower bound for the per-iteration execution time of a loop is, in fact, based on a lower bound for the per-iteration execution time of a dependence graph, and is equal to the sum of the lower bound for each of its dependence graphs. Short of actually executing a loop, there is no information about the execution frequency of each dependence graph. Consequently, this lower bound for a loop's execution time is a *static* one because it does not accurately account for dynamic information and, instead, assumes that all dependence graphs in a loop are executed the same number of times.

A lower bound on the execution time for a dependence graph is equal to the number of times a critical resource is used, where a critical resource is a functional unit that is used most frequently to execute operations in that dependence graph [107]. For example, the load port is a critical resource for the dependence graph in Figure 5.1. The following statements summarize the relationships that establish the lower bound for the execution time of a dependence graph:

a lower bound for the per-iteration execution time of a dependence graph, G
= the number of times a critical resource is used in G
$\leq$ the number of chimes needed to execute G
$\leq$ the number of clock periods needed to execute one iteration of G

The number of times a critical resource is used is a lower bound on the execution time of a dependence graph because it must be less than or equal to the number of chimes needed to execute a dependence graph; otherwise, a critical resource would be used twice in the same chime, which is impossible. In turn, the number of chimes needed to execute one iteration of a dependence graph must be less than or equal to the number of clock periods because the chime count only considers vectorizable operations, whereas the clock-period count also includes loop- and strip-overheads.

Thus, determining the lower bound for the execution time of a dependence graph is a simple matter of counting the different types of operations in such a graph, dividing the frequency of each operational type by the number of functional units that execute that operational type, and determining the maximum of these quotients. In other words, a lower bound for the execution time of a dependence graph is equal to:

$$\max_{\mathcal{T}} \left(\frac{\text{the number of operations of type } \mathcal{T}}{\text{the number of functional units that execute the operational type } \mathcal{T}} \right)$$

For example, the lower bound for the dependence graph in Figure 5.1 is four chimes when the hardware has two load-ports and a store-port. This lower bound is greatly affected by the configuration of functional units. For example, if there were only one memory-port in the hardware, the lower bound for the dependence graph in Figure 5.1 would be eight chimes.

A static lower bound on execution time gives a static upper bound on the improvement in performance relative to that of the *cft77* compiler using eight vector registers. Figure 5.4, which summarizes this relative performance, shows that there is a possibility for substantial improvement for almost all the loops and that the performance of the workload can be improved by up to 37%. Although an upper bound on relative performance difference should always be positive, there are three data points that show a negative difference. This is because these loops do not conform to the assumption that all dependence graphs in a loop are executed the same number of times. For these loops, a scalar reduction is computed, and the vectorized version consists of two dependence graphs: one to compute partial sums and the other to compute the final sum (for a full explanation of this transformation see Section 2.3 on page 17 in Chapter 2, *Fundamentals of Vector Architectures*). The number of times these dependence graphs are executed is different; the first is executed several times, and the second is executed only once. Without this dynamic information, the static lower bound places equal emphasis on both dependence graphs, thus over-estimating the execution time of the loop and showing a paradoxical performance degradation. Despite these three misleading data points, the overall data indicate that there is a possibility for significant improvement in performance.

Unfortunately, the next set of data appears to contradict the optimistic promise shown by the static upper bound. In addition to the obvious question of "Do execution orders exist that can achieve this upper bound?", the question that is more pertinent to this study is "What is the minimum number of registers needed to achieve this upper bound?" My theory hypothesizes that more than eight vector registers are needed. If this hypothesis is true, then increasing the number of registers should, on average, reduce the execution time. Because the *cft77* compiler can be directed to use any number of vector registers, I can easily test my hypothesis by comparing the performance using 8 vector registers with the performance using 64 registers, the latter number of registers being sufficiently large to avoid adverse effects on performance due to limited register capacity. This performance

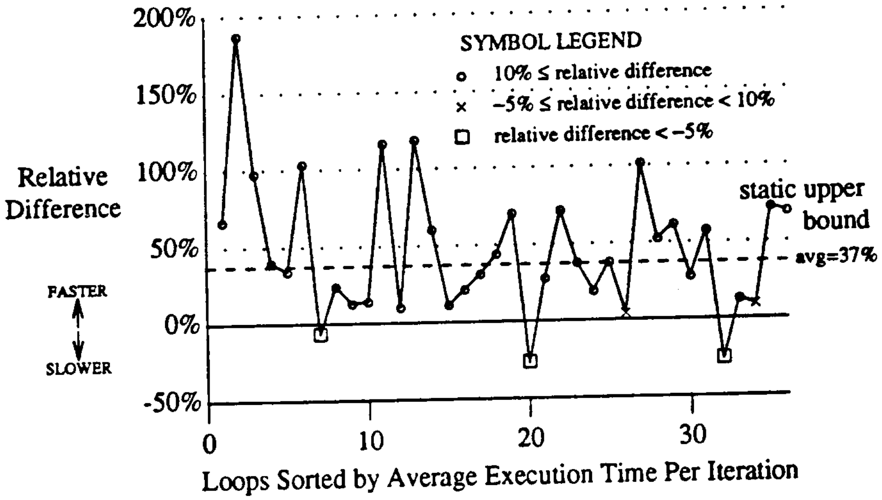


Figure 5.4: Static Upper Bound Vs. *Cft77* Scheduler Using 8 Registers

This graph shows the maximal improvement in performance over that of the 1990 version of the *cft77* scheduler using 8 vector registers, indicating that there is a possibility for significant improvement in performance. Section 5.2.1 describes the performance metrics and the basic layout of this graph, and Figures 5.18 and 5.19 list the execution times plotted in it.

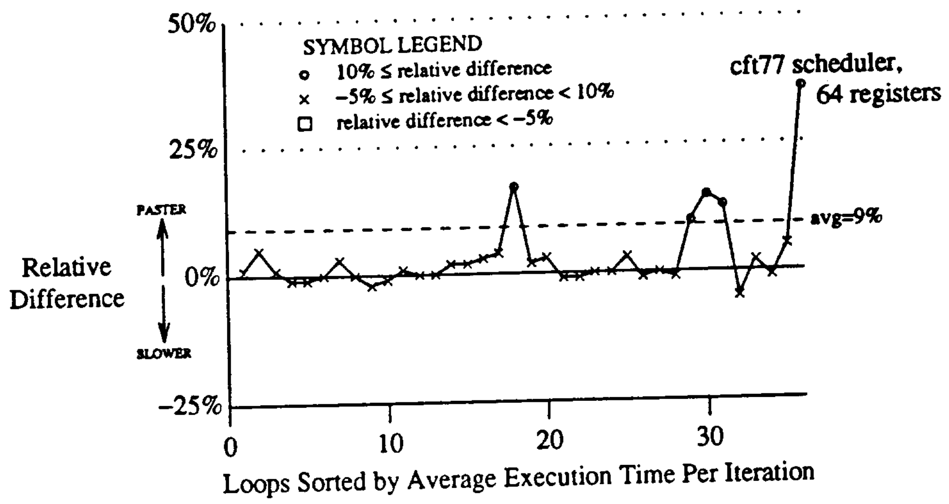


Figure 5.5: *Cft77* Scheduler Using 64 Registers Vs. *Cft77* Scheduler Using 8 Registers

To test my hypothesis that more than 8 vector registers are needed to improve performance, this graph compares the performance of the 1990 version of the *cft77* scheduler using 64 vector registers to that of the same scheduler using 8 vector registers. Section 5.2.1 describes the performance metrics and the basic layout of this graph, and Figures 5.18 and 5.19 list the execution times plotted in it.

data, gathered using the 1990 version of the *cft77* compiler, is summarized in Figure 5.5.

Because more registers are provided, using 64 vector registers should always be faster than using 8. However, a few loops execute slower with 64 registers, the worst case being 5% slower. After hand-examining the assembly code for several of these loops, I determined that the only difference between using 64 registers and using 8 for the same loop is that the data is placed in different locations in memory, possibly causing conflicts to occur in different memory banks. Because memory-bank conflicts can cause a 5% variance in performance, I believe that relative differences less than 5%, including negative ones, are due to differences in memory access patterns and should be considered insignificant.

Overall, the relative improvement to performance for the entire workload when using 64 vector registers instead of 8 registers is about 9%, a reasonable gain in performance. A great disappointment, however, is the meager distribution of the individual differences in relative performance: only 5 out of the 36 loops show more than a 10% improvement in performance, and the rest show less than a 5% performance improvement. Not enough loops show a significant performance improvement to warrant increasing the number of registers beyond eight. As a result, adding more registers appears not, on average, to reduce the execution time significantly.

One possible explanation I considered for these apparently contradictory results is that the upper bound is unrealistic, the *cft77* scheduler already provides the best achievable performance, and 8 vector registers are enough. In particular, the static lower bound does not use any information about the *structure* of a dependence graph, which may actually prevent the lower bound from being achieved. I didn't believe this to be true because I had experimented with some of the loops by hand in a manner similar to that in the example in the previous subsection and estimated that their execution time could be improved by adding more registers. Another possible explanation is that more than 64 vector registers are needed to achieve the upper bound. I didn't believe this second explanation either because the execution orders I produced in my hand-experiments used fewer than 64 registers to improve performance. A third possibility is that more sophisticated compiler optimizations, such as loop unrolling or the transformation made in Figure 5.2, are needed to expose more parallelism. Although such techniques will eventually be needed when there are more functional units in the hardware, I did not believe that this was the case here because I did not have to resort to such techniques in my hand-experiments to show an improvement to performance.

A fourth explanation, and the one that I accept, is that a scheduling algorithm different from the one used by the 1990 version of the *cft77* compiler is needed. This is because different scheduling algorithms can produce different execution orders for the same dependence graph, and different execution orders can require not only different amounts of time to execute but also different numbers of registers, as shown by the example in the previous subsection. Even though the *cft77* scheduler does not use more registers to improve performance, this does not preclude a different scheduling algorithm from doing so. Hence, in addition to this study's primary goal of determining a cost-effective number of vector registers, a secondary goal is to show that a different scheduling algorithm uses more registers more effectively than does the *cft77* scheduling algorithm. In the rest of this chapter, I compare, both qualitatively and quantitatively, the impact the *cft77* scheduler and a different scheduler have on performance to show that this fourth explanation reconciles the disparate results presented above.

5.2 Experimental Framework

In this section, I describe the performance criteria and the methodology I use to carry out the studies throughout this chapter. Other aspects of the experimental framework, such as the architectural platform, the performance tools, and the workload are described in Chapter 4, *Common Experimental Framework.*

5.2.1 Performance Criteria

As part of my investigation, I compare the performance of different combinations of scheduling algorithms and number of vector registers. The raw data for these comparisons are the times needed to execute each loop in the CRI workload using various combinations of scheduling algorithms and number of registers. Unless stated otherwise, all execution times are measured using the Cray Y-MP simulator. In this subsection, I describe how this raw data is used to determine whether the performance of two configurations differs significantly. First, I describe the performance metrics used to represent the time it takes to execute a loop and the time it takes to execute the entire workload when using a particular configuration. I then describe how I compare the performances among various configurations and give the criteria for acceptable performance improvement.

As the performance metric for a loop, I use the average time t to execute an iteration of that loop:

$$t = \frac{\text{the time to execute the entire loop}}{\text{the number of iterations executed for the loop}} = \frac{T}{L}$$

An obvious alternative to using per-iteration time as a performance metric for a loop is using the time it takes to execute the entire loop. Loop-execution time is influenced by factors arising from a program and a compiler, whereas per-iteration time is more influenced by just the compiler because the number of iterations executed for a loop, which is influenced by a program, is factored out. This makes per-iteration time the preferred metric because it emphasizes the differences in performance among various execution orders, differences that I want to measure. Moreover, whereas loop-execution time includes the times to execute loop- and strip-overheads, per-iteration time also includes these overheads because it is calculated from loop-execution time. Although using per-iteration time assumes that all loops in a workload are executed the same number of times, I address this assumption below when I discuss the criteria for evaluating acceptable improvements to performance.

Another alternative to per-iteration time is the MFLOPS[3] rate of a loop, which is a standard metric for measuring the performance of scientific workloads [55, 84]. Because the MFLOPS rate r of a loop is computed from the same data as the per-iteration time t, these two metrics are, in fact, inversely proportional to each other. Just as t is the average time it takes to execute one *iteration* of a loop, $\frac{1}{r}$ can be interpreted as the average time it takes to execute one *floating-point operation* of that loop. In other words, t is equal to the product of $\frac{1}{r}$ and f, the number of floating-point operations executed in one iteration of that loop. This equality can be shown algebraically as follows:

$$t = \frac{T}{L} = \frac{T}{L \times f} \times f = \frac{1}{r} \times f$$

I use per-iteration time instead because not all the loops in the CRI workload contain floating-point operations. Furthermore, using per-iteration times shifts the emphasis from floating-point operations to execution time, which is more directly related to performance.

As the performance metric for the entire workload, I use the sum of per-iteration times of all loops in the workload. Just as per-iteration time is proportional to the inverse of MFLOPS rate, this sum is proportional to the inverse of the weighted harmonic mean of MFLOPS rates of each loop, where the weight for a loop is the number of floating-point operations executed in one iteration of that loop. This equality can be shown algebraically as follows:

$$\sum t_i = \sum \frac{T_i}{L_i}$$

[3] *MFLOPS* is an acronym for "*M*illions of *F*loating-point *O*perations *P*er *S*econd."

$$\begin{aligned}
&= N \times \frac{\sum \frac{T_i}{L_i f_i} \times f_i}{N} \\
&= N \times \frac{\sum \frac{1}{r_i} \times f_i}{N} \\
&= N \times \frac{1}{\text{weighted harmonic mean of MFLOPS rates}}
\end{aligned}$$

where N is the number of loops in the workload
T_i is the time to execute the i^{th} loop
L_i is the number of iterations executed for the i^{th} loop
t_i is the per-iteration time for the i^{th} loop
r_i is the MFLOPS rate for the i^{th} loop
f_i is the number of floating-point operations
executed in one iteration of the i^{th} loop

Although the harmonic mean is a standard metric for summarizing the performance of a workload, I use the sum of per-iteration times for the same reasons I cited in the previous paragraph.

A critical aspect of my investigation is to determine which of two configurations is faster. To do this, I examine the relative difference in performance of both individual loops and the entire workload. For an individual loop, I use the performance metric for a loop to calculate the difference in performance of a new configuration relative to a base configuration

$$\frac{t_i^{\mathrm{B}}}{t_i^{\mathrm{N}}} - 1$$

where t_i^{B} is the per-iteration time for the i^{th} loop using the base configuration, and t_i^{N} is the per-iteration time for the i^{th} loop using the new, and presumably faster, configuration. For a summary of the individual relative differences, I use the performance metric for a workload to calculate the relative difference in performance of the entire workload:

$$\frac{\sum t_i^{\mathrm{B}}}{\sum t_i^{\mathrm{N}}} - 1$$

Although I examine many different configurations, I directly compare only two at a time by plotting their relative performance differences in a graph that has a specific structure. Not only does such a graph provide a visual way to compare the performances of two configurations, but multiple graphs with this common structure allow the performances of several configurations to be compared simultaneously. Figure 5.5 on page 82 (in the previous section) shows an example of such a graph. Loops are plotted along the X-axis, sorted by execution time. In other words, the leftmost loop executes in the fewest clock periods per iteration when using the *cft77* scheduler and 8 vector registers, and the rightmost loop executes in the greatest number of clock periods. The difference in performance of a new configuration relative to that of a base configuration is plotted along the Y-axis. The new configuration is listed first in the caption title (for example, "*Cft77* Scheduler Using 64 Registers" in Figure 5.5), and the base configuration is listed last. Positive values for the relative performance difference indicate that the new configuration is faster than the base one, and the larger the value, the faster it is. A solid curve connects the relative performance differences of individual loops. To readily identify which loops perform significantly better or worse when using a new configuration, the relative difference in performance of a loop is plotted using one of three symbols:

ο 10% ≤ relative difference
× −5% ≤ relative difference < 10%
□ relative difference < −5%

A dashed horizontal line gives the relative difference in performance for the entire workload.

The reason for comparing the performance of two configurations is to choose one of them to implement; a significant difference in performance justifies implementing the faster configuration, which presumably is the new one. An improved performance of a new configuration over that of a base one is considered *significant* if the following two criteria are satisfied:

1. the improvement in performance for the entire workload is greater than 10%, and
2. the majority of individual loops in the workload show a performance improvement greater than 10%.

In more mathematical terms, implementing the new configuration is justified only if:

1. $\sum t_i^B / \sum t_i^N - 1$ is greater than 10%, and
2. the median of the values $t_i^B / t_i^N - 1$ is greater than 10%.

Any performance improvement of less than 10% is considered insignificant because other factors in implementation, such as a shorter clock period, can easily eclipse such a small improvement. Furthermore, as I will show in the next chapter, the minimal increase in hardware is 10%. Hence, any improvement in performance must be at least 10% to justify the increased cost.

Each criterion is needed for a different reason. The first criterion avoids the negative consequences of Amdahl's Law. In other words, large improvements in individual loops must have a significant impact on the execution time of the entire workload. The second criterion prevents loops with longer per-iteration times, which have more influence on the performance of the entire workload, from dominating the decision-making process. Moreover, this criterion compensates somewhat for using per-iteration time rather than loop-execution time as the basis for my performance analysis. A few paragraphs back, I noted that using per-iteration time assumes that all loops in a workload are executed the same number of times, an assumption that using loop-execution time would avoid. But *relative* per-iteration time and *relative* loop-execution time are equal. Hence, the second criterion can be computed from either metric.

To see why both these criteria are needed, I present two examples using hypothetical sets of performance data for the CRI workload (all execution times are rounded to the nearest 10 clock periods):

> The base configuration, which is the *cft77* compiler using 8 vector registers, executes the CRI workload in 1270 clock periods.
>
> For the first example, suppose that a new configuration improves the performance of the 20 loops with the shortest per-iteration times by 45% but has no effect on the performance of the other loops. Then, the median performance improvement for this example is 45%. However, because the sum of per-iteration times of the 20 loops is 240 clock periods when using the base configuration, the performance improvement for the workload is only $\left(\frac{1270}{1270-240+\frac{240}{1.45}} - 1\right) 100$ = 6%.
>
> For the second example, suppose that a different configuration improves the performance of the 2 loops with the longest per-iteration times by 45% but has no effect on the performance of the other loops. Because the sum of the per-iteration times of these 2 loops is 440 clock periods when using the base configuration, the performance improvement for the workload is $\left(\frac{1270}{1270-440+\frac{440}{1.45}} - 1\right) 100 = 12\%$. However, because the remaining 34 loops show no improvement in performance, the median performance improvement is 0%.

These two hypothetical examples show a huge variation in performance improvement among individual loops. In both cases, a select subset of loops shows a significant performance improvement of 45% whereas the rest of the loops show none. In the first example, this select subset is a significant portion of the loops in the workload but does not represent a significant enough portion of the time to execute it. The select subset in the second example represents a significant portion of the time to execute the workload but is not a significant portion of the loops within it. An actual example of this situation is illustrated in Figure 5.5, which shows the performance of the *cft77* scheduler using 64 registers relative to that of the same scheduler using 8. In both cases, because only one criterion is satisfied, implementing the new configuration is not worthwhile despite large performance improvements among individual loops.

5.2.2 Methodology

I considered two methods for achieving the goals of this chapter. These two methods offer a tradeoff between providing a definitive answer and being able to produce an answer in a reasonable amount of time.

The first method solves the following optimality problem: determine the shortest time to execute a dependence graph using some fixed number of registers. By comparing the optimal performances using a different number of registers, I can then definitively say that *the most* cost-effective number of registers is the one for which:

1. using fewer than that number decreases performance significantly, and
2. using more than that number does not increase performance significantly.

Unfortunately, the major disadvantage of this method is that this optimality problem is in a class of *precedence constrained problems*, which are known to be NP-hard for an arbitrary dependence graph [46]. Such problems are so computationally intensive that obtaining an answer for just one instance can require *years* of computational time; moreover, the existence of methods that are less computationally intensive is currently thought to be unlikely. This optimality problem, however, is no longer NP-hard when the dependence graph is a *tree*, which is a specially-structured dependence graph with no common subexpressions and where each operation has only one dependence; the dependence graphs illustrated in this chapter are trees (in Figures 5.2, 5.12, and 5.13). In fact, there are algorithms that take advantage of a tree's regular structure to generate a minimal execution-time order in polynomial time [87, 98, 99, 3]. But because 75% of the dependence graphs in the CRI workload are not trees, this problem is NP-hard for the majority of this workload and, in particular, for the larger dependence graphs.

An obvious method for finding an optimal order for such an NP-hard problem is to examine all possible orders for a dependence graph. The time it would take to compare all possible orders of a dependence graph with N operations is proportional to $N!$, which is a function that grows more than exponentially with a constant increase in the number of operations in a dependence graph. Despite this daunting super-exponential growth rate, today's computers can exhaustively compare the orders of a dependence graph if the number of operations is "small" enough. Moreover, characteristics of the problem can be used to reduce the number of orders that are examined. For example, because not all orders satisfy the partial order specified by a dependence graph, we can ignore any order whose prefix does not satisfy the partial order, such as orders beginning with the operation STORE_{10} for the dependence graph in Figure 5.1 (on page 74 of the previous section). Other rules based on execution time and register usage can also be used to further prune the number of examined orders. If these pruning rules allow most of the dependence graphs in the CRI workload to be exhaustively compared in a reasonable amount of time, then this method could still be used to provide a definitive answer.

To determine whether such a method is feasible, I timed how long a Sun SPARCstation 1 takes to exhaustively compare the orders of progressively larger and larger dependence graphs. Figure 5.6 shows the results of these timings. Optimal solutions for dependence graphs with less than 30 operations can be found in less than a minute. Unfortunately, the time it takes to find an optimal solution grows extremely quickly even when pruning rules are used. Based on this data, Figure 5.7 lists estimates for the comparison times of the larger dependence graphs in the CRI workload. Even computers in the near future are unlikely to improve this situation substantially because their performance is progressing by only a factor of at most two every two years, whereas an increase of just two operations in a dependence graph requires that search time be increased by a factor of 2.4.

In summary, although this method could provide a definitive value for the most cost-effective number of registers, it is infeasible because finding an optimal execution order for the larger dependence graphs in the CRI workload requires years of computational time. Moreover, this method is impractical from a compiler standpoint because the time it takes to find an optimal order is far greater than the time to execute the resultant code for the larger dependence graphs, regardless of how frequently the code is executed. This method could, however, be used if dependence graphs with more than 50 operations were excluded, but doing so is unacceptable not only because 36% of them in the CRI workload contain

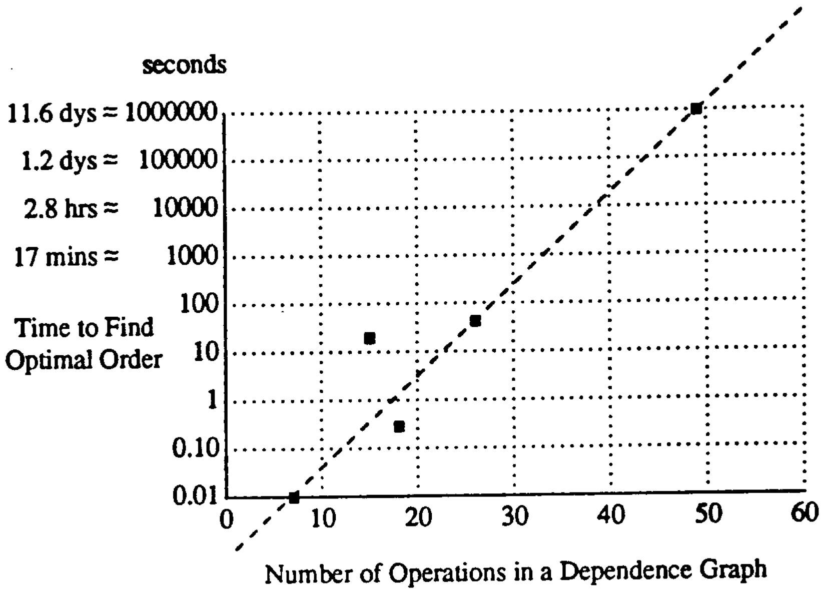


Figure 5.6: Completion Times for Exhaustive Comparisons

This graph shows how long a Sun SPARCstation 1 took to find an optimal execution order for each of five dependence graphs from the CRI workload by exhaustively comparing all the execution orders of a dependence graph. The five dependence graphs have 7, 15, 18, 26 and 49 operations, respectively. The dashed line shows that the comparison time grows exponentially relative to a constant increase in the operations in a dependence graph.

NUMBER OF OPERATIONS IN A DEPENDENCE GRAPH (N)	ESTIMATED TIME FOR EXHAUSTIVE COMPARISON ($10^{\frac{4.35}{23}N-3.27}$ seconds)
49	10^6 seconds ≈ 11.6 days
54	10^7 seconds ≈ 100 days
60	10^8 seconds ≈ 3.2 years
65	10^9 seconds ≈ 32 years
70	10^{10} seconds ≈ 3.2 centuries
80	10^{12} seconds ≈ 320 centuries

Figure 5.7: Estimated Times for Exhaustive Comparisons

This table lists estimated times for finding an optimal execution order for some of the larger dependence graphs in the CRI workload. The dashed line in Figure 5.6 is used to calculate these estimates, which are rounded to the nearest order of magnitude.

more than 50 operations, but also because I expect these graphs to be the ones with larger register requirements.

A second, more practical method is to use scheduling algorithms that execute in polynomial time and then to compare the performance of their execution orders using a different number of registers. In addition to its computational practicality, another advantage to this method is that an algorithm that results in a significant improvement to performance can be adopted by a compiler with only a minimal increase in compilation time. The main disadvantage to this method is that it cannot provide a definitive answer because a scheduling algorithm chooses an execution order based on heuristics and, although heuristics are designed to minimize some aspect of an execution order, optimality cannot be guaranteed. Nonetheless, because I am beginning with a configuration that already exists — namely, the Cray Y-MP with 8 vector registers and the *cft77* scheduling algorithm — I can at least show that a different scheduling algorithm can significantly improve performance when using more registers. The fact that such an algorithm exists is not necessarily obvious as indicated by the initial performance results presented in the previous section. Because this method is computationally practical and the first one infeasible, I use the former method to achieve the goals of my study.

5.3 A Comparison of Two Scheduling Algorithms

In this section, I describe and contrast the two scheduling algorithms I use in my study. One was used in the 1990 version of the *cft77* compiler, and the other, which I developed to emulate what I had accomplished in my hand experiments, is a variant of list scheduling. In the first part of this section, I highlight similarities between these algorithms by comparing a scheduling algorithm for a vector architecture to ones for other architectures and by describing, in general, how a vector scheduling algorithm works. In the second half of this section, I describe the different aspects of the scheduling algorithms and provide a detailed description of each.

The function of any scheduling algorithm, regardless of the underlying architecture, is to generate an order in which to execute the operations of a given dependence graph. Understanding how the underlying architecture affects a scheduling algorithm allows an algorithm designed for one architecture to be more readily adapted to another one. For example, the underlying architecture determines what the operations of a dependence graph represent. For a scalar or VLIW architecture, there is a one-to-one correspondence between an operation in a dependence graph and an operation to be executed. For a vector architecture, in contrast, an operation in a dependence graph represents many independent operations to be executed.

Despite this disparity in representations, scheduling vectorizable operations is comparable to scheduling them for a VLIW architecture; scheduling vectorizable operations is least like scheduling ones for a scalar architecture, even though both issue instructions sequentially. For a scalar architecture, a scheduling algorithm must contend with delayed as well as deeply pipelined operations [52]. In contrast, a scheduling algorithm for a VLIW architecture focuses on grouping together operations that can initiate in parallel [34, 76]. To simplify the description of how such groupings are done, operations are often assumed to execute in unit time, and an abstract machine model is used to handle deeply-pipelined and delayed operations. Similarly, a vector scheduler is concerned with grouping vectorizable operations that can execute in parallel, which in turn causes individual ones to initiate in parallel.

A major difference between schedulers for these architectures and one for a vector architecture is that the unit of time for the latter is one chime rather than one clock period. In other words, a VLIW scheduler groups operations into a VLIW instruction, which executes in one clock period. A vector scheduler, however, groups vectorizable operations that can execute in one chime. But rather than actually grouping these operations into one instruction, a vector scheduler merely specifies the *order* in which these operations are to execute to produce the parallelism found by the scheduler.

To facilitate scheduling vectorizable operations, I use a *chime table* to keep track of when operations are scheduled to execute on which functional unit. A comparable table is also used in schedulers for VLIW architectures, and could be called generically a *functional unit reservation table*. In the context of vector scheduling, a chime table is a matrix where each row represents one chime, each column represents a functional unit in the hardware, and the ij^{th} entry represents an operation that is to be executed in the i^{th} chime by the j^{th} functional unit. To generate an order, a scheduling algorithm places operations into a chime table according to a set of rules that vary from scheduler to scheduler. Once all operations are placed into a chime table, they are removed from the table in chime order, so that their dependences are still preserved. The order of removal is the execution order generated by a scheduling algorithm.

As with the examples in Figure 5.3 (on page 77), an estimate of the execution time is given by the number of rows (or chimes) with at least one scheduled operation, and an estimate of the minimal register requirements is given by the maximum number of live values in a chime. These are estimates only because counting by chimes ignores both the latencies of deeply-pipelined operations and the execution of any scalar operations, both of which are part of the loop and strip overheads. Although this omission simplifies a vector scheduling algorithm, these estimates are not unduly accurate. This is because in any one chime, the latency of *only one* operation is actually exposed even though many operations are executed, and an operational latency typically lasts only a fraction of a chime.

The type of scheduling algorithms I consider are called *simple vector schedulers* [107], ones that schedule operations from the *same* iteration only. Other algorithms, such as polycyclic scheduling,[4] trace scheduling, and loop unrolling, schedule operations from *different* iterations to increase the amount of parallelism that occurs. For example, the Cray-2 uses polycyclic vector scheduling to increase the amount of parallelism to compensate for the lack of chaining hardware, which prevents flow-dependent operations from executing in parallel [108, 32, 26]. In my study, I consider only simple vector schedulers because, as the quantitative results of the next section show, significant improvement to performance is still possible without having to resort to more complex scheduling algorithms.

The main difference between the two scheduling algorithms I use is how operations are placed into a chime table. When examining these placement rules, there are three issues to consider:

[4] Polycyclic scheduling is also known as *software pipelining* [76] or *overlapped loop scheduling* [28].

1. the *goal* of the placement rules,
2. the *order* in which operations are processed, and
3. the *strategy* for finding a time slot in which an appropriate functional unit and register(s) are available

Operations are first sorted into order by priorities and then placed into a chime table one at a time in the sorted order.[5] What this placement strategy and these priorities are depends largely upon what the goals of the scheduling algorithm are: whether to minimize execution time or minimize register usage. Because minimizing execution time tends to increase register usage and vice versa (as the examples in Figure 5.3 demonstrate), these cannot be goals of equal priority. All three aspects — the goals, order, and strategy — of placing operations into a chime table are different for the two scheduling algorithms I use.

Details about each algorithm are given in Figures 5.8 and 5.9. These figures explain how the priorities and strategy exploit some aspect of the problem to achieve the primary and secondary goals of the algorithm. Figure 5.8 describes the scheduling algorithm used by the 1990 version of the *cft77* compiler [62]. Figure 5.9 describes the algorithm that I developed, which is a variant of list scheduling, a technique originally used for assigning a set of partially-ordered tasks to a fixed number of processors in a parallel processing system [1]. List scheduling is also used for scheduling scalar instructions [48] and in trace scheduling, which is used for VLIW architectures [34]. Variants of list scheduling differ in the priorities used to determine the order of scheduling, but the strategy is always the same: after scheduling the $(i-1)^{st}$ operations, schedule the i^{th} one as early as possible under the constraints of the partial order and availability of hardware resources. In choosing an order and a strategy, I designed the list-scheduling algorithm to emulate what I had been doing when I hand-scheduled several dependence graphs.

The major differences between the *cft77* and list schedulers are summarized in Figure 5.10. The differences in order and strategy between these two schedulers directly reflect the differences in their goals. The *cft77* scheduler emphasizes register usage over execution time because it must generate production code for a vector architecture with only eight vector registers. Nonetheless, the number of registers provided in hardware is an input parameter to this algorithm to avoid needlessly increasing execution time in an attempt to use fewer registers. In contrast, the list scheduler I developed does not consider limiting register usage to match the number of registers provided in hardware because I am more interested in examining the best performance with no constraints imposed by limited register capacity. Although the final step is designed to reduce register usage, this step should not increase the execution time of the resultant order.

Because the goals of both schedulers are different, we would expect the list scheduler to produce orders that execute in less time than those of the *cft77* scheduler, as demonstrated by the examples in Figure 5.3. What this qualitative description does not indicate is *how much* better the list scheduler is, nor does it indicate *how often* the list scheduler

[5] Not all scheduling algorithms have such a clear separation between order of processing and placement. For example, in the algorithm described by Gibbons and Muchnick, the order of processing is affected by the strategy for placement in that the criteria for choosing the next operation to be scheduled is based on what has already been scheduled [48].

Indicate that all the functional units are being used in the first chime of the chime table so that operations without any predecessors (typically LOADs) can be scheduled properly.

Given a dependence graph, schedule each operation in order of

1. appearance by statement in the source code, and
2. within a statement, decreasing maximal path distance from an operation with no successors (typically a STORE)

by choosing the first chime c from the end of the chime table for which one of the following is true:

1. the number of live values in chime $c-1$ is equal to the number of vector registers in the hardware,
2. a predecessor has been scheduled in chime c,
3. all appropriate functional units are being used in chime $c-1$, or
4. an operand to the operation is being used by an operation already scheduled in chime $c-1$.

Figure 5.8: *Cft77* Scheduling Algorithm

This figure describes how the scheduling algorithm used by the 1990 version of the *cft77* compiler places operations of a dependence graph into a chime table. The primary goal of this scheduling algorithm is to minimize register usage; its secondary goal is to minimize execution time. As an example of its application, this scheduling algorithm produces the execution order on the left in Figure 5.3 for the dependence graph in Figure 5.1.

The first enumerated list indicates how priorities are assigned to operations to determine the order in which they are scheduled. The goal of this list is to minimize register usage, which is done by keeping the lifetimes of many values as short as possible. Executing operations in order of appearance in the source code causes values to be used shortly after they are produced.

The second enumerated list outlines the strategy for placing operations into a chime table. The goal of this list is to minimize execution time without unduly increasing register usage. The first item prevents register usage from exceeding the physical capabilities of the hardware, and the last three items minimize execution time by interleaving the execution of operations from adjacent statements. The fourth item of this list reflects the fact that the execution of operations that use the same value is necessarily sequential in the Cray Y-MP, regardless of the availability of functional units. Sequential execution is necessary because a vector register has only one read port, whereas parallel execution of multiple successors requires a vector register with multiple read ports. The fourth item shows this sequential execution from the viewpoint of one of the successors.

Given a dependence graph, schedule each operation in order of

1. decreasing maximal path distance from an operation with no successors (typically a STORE), and
2. for operations with the same maximal path distance, decreasing number of successors

by choosing the *first* chime *c* such that

1. all ancestors have been scheduled before or in chime *c*,
2. an appropriate functional unit is available in chime *c*, and
3. operands to the operation are not being used by an operation already scheduled in chime *c*.

Reschedule operations without any predecessors (typically LOADs) by choosing the *latest* chime *c* such that

1. all successors have been scheduled after or in chime *c*, and
2. an appropriate functional unit is available in chime *c*.

Figure 5.9: List Scheduling Algorithm

This figure describes how a list-scheduling algorithm places operations of a dependence graph into a chime table. The primary goal of this scheduling algorithm is to minimize execution time; its secondary goal is to minimize register usage. As an example of its application, this scheduling algorithm produces the execution order on the right in Figure 5.3 for the dependence graph in Figure 5.1.

The first enumerated list indicates how priorities are assigned to operations to determine the order in which they are scheduled. The goal of this list is to minimize execution time, which is done by giving the highest priority to operations that many other operations indirectly depend upon. For operations with the same maximal path distance, higher priority is given to those with more directly-dependent operations. In other words, higher priority is given to an operation whose value is used by more operations.

The second and third enumerated lists outline the strategy for placing operations into a chime table. The goal of the first of these lists is to minimize the execution time. The reason for item number three has to do with the single read port of a vector register, and the explanation for its inclusion is the same as the one given for the *cft77* scheduling algorithm in Figure 5.8. The goal of the last list is to shorten the lifetimes of values in order to reduce register usage. This is done by placing an operation as close as possible to the operations that use its value. Only operations without any predecessors are rescheduled because they have the most flexibility when placed in a partially-filled chime table. This rescheduling should not affect the execution time of the resultant code.

An algorithm similar to this one has been implemented in a version of the *cft77* compiler that is more recent than the one used for my studies. In addition to the above, the newer *cft77* version also takes into account register usage to avoid generating an excessive number of register spills.

	cft77 SCHEDULER	LIST SCHEDULER
GOALS	1. minimize register usage 2. minimize execution time	1. minimize execution time 2. minimize register usage
ORDER	as operations appear in the source code	based on properties of the dependence graph
STRATEGY	start at end of chime table and work backwards, taking into consideration the number of vector registers available in hardware	start at beginning of chime table and work forwards

Figure 5.10: Comparison of *Cft77* and List Scheduling Algorithms

This table summarizes the major differences between the *cft77* and list scheduling algorithms. Details about each algorithm are given in Figures 5.8 and 5.9, respectively.

is better. As a result, quantitative performance data is needed to justify changing the algorithm used in the *cft77* compiler.

5.4 How Many Vector Registers?

Up until now, I have described qualitatively why more registers and a scheduling algorithm that is different from the one used in the 1990 version of the *cft77* compiler are needed to effectively use the functional units in the Cray Y-MP. In this section, I present data that not only substantiates these observations but, more importantly, shows *how many* registers are needed and *how much* of an improvement to performance is possible with more registers and a different scheduling algorithm. The primary goal of this study is to find a cost-effective number of registers and the secondary goal is to show that a different scheduling algorithm uses more than 8 vector registers better than the *cft77* scheduler does. The initial performance results presented in Section 5.1.2 suggest that the secondary goal must be achieved before the primary one can be found. Consequently, I first present data to show that the list-scheduling algorithm in Figure 5.9 uses more registers more effectively than does the *cft77* scheduler. I then present data by which to choose a cost-effective number of registers, and then I discuss the impact of using this number of registers on the the performance of vectorizable loops and an entire program. Finally, I explain how the data shows that larger loops are more likely to need more registers for greater performance and that register spills have minimal impact on execution time. All the presentations of data use the basic graph structure described in Section 5.2.1, and the execution times used in these graphs are listed in Figures 5.18 and 5.19 at the end of this section.

From the performance results described in Section 5.1.2, we already know that the *cft77* scheduling algorithm described in Figure 5.8 (in the previous section) does not use

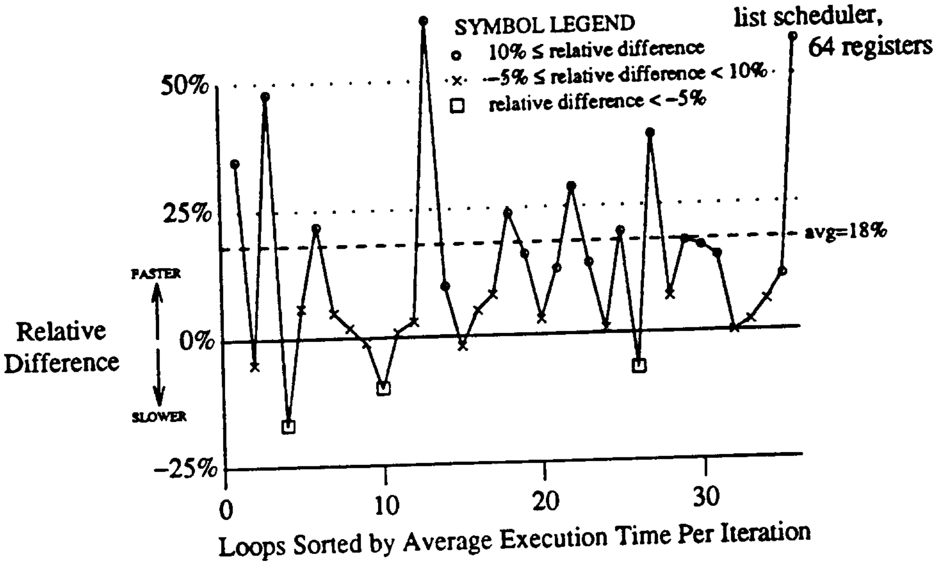


Figure 5.11: List Scheduler Using 64 Registers Vs. *Cft77* Scheduler Using 8 Registers

To show that a different scheduler can use more registers more effectively than the scheduler used in the 1990 version of the *cft77* compiler, this graph compares the performance of the list scheduler using 64 vector registers to that of the *cft77* scheduler using 8 vector registers. Section 5.2.1 describes the performance metrics and the basic layout of this graph, and Figures 5.18 and 5.19 list the execution times plotted in it.

more than 8 vector registers effectively because little improvement to performance resulted when using 64 registers, a number that is sufficiently large to avoid adverse performance effects due to limited register capacity. Hence, the secondary goal of this study is easily achieved by demonstrating that the list scheduling algorithm in Figure 5.9 of the previous section can use 64 registers to significantly improve performance over that of the base configuration. To generate the raw performance data, I replaced the scheduler in the *cft77* compiler with the list scheduler. Figure 5.11 summarizes the improvements to performance that result.

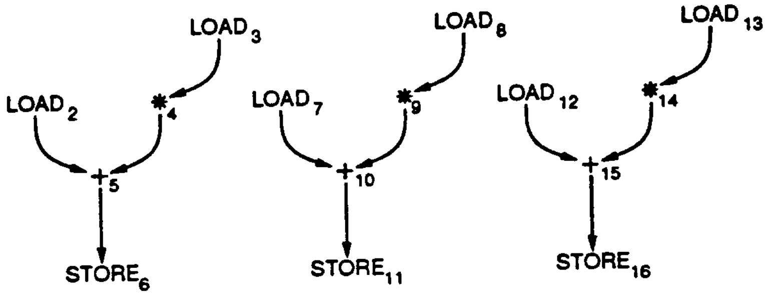
We would expect that using 64 vector registers should be at least as fast as using 8 registers, regardless of the scheduling algorithm. Yet, there are three loops that perform worse than 5% when the list scheduler is used rather than the *cft77* one. These data points emphasize the fact that scheduling algorithms rely on heuristics, which cannot guarantee that the best execution order is generated for *every* dependence graph. For the two worst cases, which are 13% and 10% slower, the order for *scheduling* operations causes the chime estimates to be one chime longer than that of the corresponding orders produced by the *cft77* scheduler. For the loop with the worst performance, Figure 5.12 illustrates how the heuristic used by the list scheduler to choose the scheduling order just does not work, whereas the heuristic used by the *cft77* scheduler is ideal. For the loop with the second worst performance, two operations are scheduled in the "wrong" order because they had equal priorities for scheduling, and I schedule such operations in a random order. Figure 5.13 demonstrates that using another heuristic to schedule operations with equal priority allows list scheduling to generate an order that executes in a time comparable to one generated by the *cft77* scheduler.

Nevertheless, despite these poor data points, the list scheduler improves performance significantly for the entire workload and for almost half the loops. These gains are all the more impressive because the execution times include the time to execute loop and strip overheads, for which the heuristics in the list scheduler did nothing special to compensate. Overall, the relative improvement to performance for the entire workload is about 18% when a different scheduler is used and 64 vector registers instead of 8. In addition, 21 of the loops improve their performance by more than 5%, with 17 of these achieving a performance improvement greater than 10%. Although one of the performance criteria is not met (the median performance difference is only 8%), the distribution of the individual relative differences is significantly better than the distribution resulting from the *cft77* scheduler using 64 registers (shown in Figure 5.5), where performance improved by more than 10% for only 5 of the 36 loops. Hence, Figures 5.5 and 5.11 show that the list-scheduling algorithm uses more than 8 registers better than the *cft77* scheduler does, and that, at the very least, changing the scheduling algorithm in the *cft77* compiler is warranted.

I now have the necessary tools with which to obtain the primary goal, which is to determine a cost-effective number of vector registers. Although I have shown that a significant improvement to performance is possible with 64 vector registers, fewer registers or even just a change in the scheduling algorithm could produce comparable performance results. In Figure 5.11 above, I compare two *scheduler&register* combinations where all the components differ; some combination between these two extremes could perform as well as the list scheduler using 64 registers. To better judge what a cost-effective combination is, I compare, in successive order, several pairs of *scheduler&register* combinations, Figure 5.14 summarizes these comparisons, which progressively compare the relative performance of the following *scheduler&register* combinations:

cft77 scheduler and 8 vector registers,
list scheduler and 8 vector registers,
list scheduler and 16 vector registers,
list scheduler and 32 vector registers, and
list scheduler and 64 vector registers.

The first graph, Figure 5.14a, compares the performances of the two scheduling algorithms, when both use 8 vector registers. This graph shows that the *cft77* scheduler does not always effectively use 8 registers; the list scheduler provides more than a 10% improvement for 7 loops. Hence, some of the improvement to performance seen in Figure 5.11 is attributable to a change of scheduling algorithm. Nevertheless, Figure 5.14a, in combination with Figure 5.11, demonstrates that much of the improvement to performance is attributable to both changing the algorithm *and* increasing the number of registers. Figure 5.11 shows that using more registers significantly improves performance, both overall as well as for individual loops. Figure 5.14a shows that using only 8 registers with the



Cft77 SCHEDULER

Scheduling Order
(statement number)

LOAD$_2$	(1)
LOAD$_3$	(1)
×$_4$	(1)
+$_5$	(1)
STORE$_6$	(1)
LOAD$_7$	(2)
LOAD$_8$	(2)
×$_9$	(2)
+$_{10}$	(2)
STORE$_{11}$	(2)
LOAD$_{12}$	(3)
LOAD$_{13}$	(3)
×$_{14}$	(3)
+$_{15}$	(3)
STORE$_{16}$	(3)

LIST SCHEDULER

Scheduling Order
(maximal path distance)

LOAD$_3$	(3)
LOAD$_8$	(3)
LOAD$_{13}$	(3)
LOAD$_2$	(2)
×$_4$	(2)
LOAD$_7$	(2)
×$_9$	(2)
LOAD$_{12}$	(2)
×$_{14}$	(2)
+$_5$	(1)
+$_{10}$	(1)
+$_{15}$	(1)
STORE$_6$	(0)
STORE$_{11}$	(0)
STORE$_{16}$	(0)

Operations Placed Into Chime Table

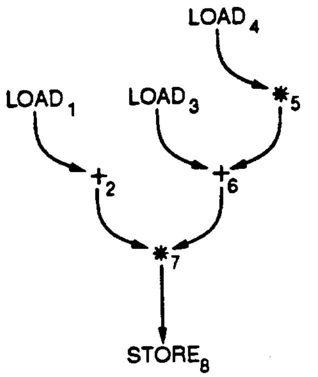
LOAD$_2$	LOAD$_3$	×$_4$	+$_5$	STORE$_6$
LOAD$_7$	LOAD$_8$	×$_9$	+$_{10}$	STORE$_{11}$
LOAD$_{12}$	LOAD$_{13}$	×$_{14}$	+$_{15}$	STORE$_{16}$

Operations Placed Into Chime Table

LOAD$_3$	LOAD$_8$	×$_4$		
LOAD$_{13}$	LOAD$_2$	×$_9$	+$_5$	STORE$_6$
LOAD$_7$	LOAD$_{12}$	×$_{14}$	+$_{10}$	STORE$_{11}$
			+$_{15}$	STORE$_{16}$

Figure 5.12: Example Showing that List Scheduler Uses Wrong Heuristic

This figure demonstrates that the *cft77* scheduler produces an execution order for the illustrated dependence graph that is better than the execution order produced by the list scheduler because the *cft77* algorithm schedules operations in order of statement number rather than by maximal path distance.



Cft77 SCHEDULER

Scheduling Order
(maximal path distance)

LOAD$_4$	(4)
* LOAD$_3$	(3)
* LOAD$_1$	(3)
$\times_5$	(3)
$+_6$	(2)
$+_2$	(2)
$\times_7$	(1)
STORE$_8$	(0)

Operations Placed Into Chime Table

LOAD$_4$	LOAD$_3$	$\times_5$	$+_6$	
LOAD$_1$		$\times_7$	$+_2$	STORE$_8$

LIST SCHEDULER

Scheduling Order
(maximal path distance)

LOAD$_4$	(4)
* LOAD$_1$	(3)
* LOAD$_3$	(3)
$\times_5$	(3)
$+_6$	(2)
$+_2$	(2)
$\times_7$	(1)
STORE$_8$	(0)

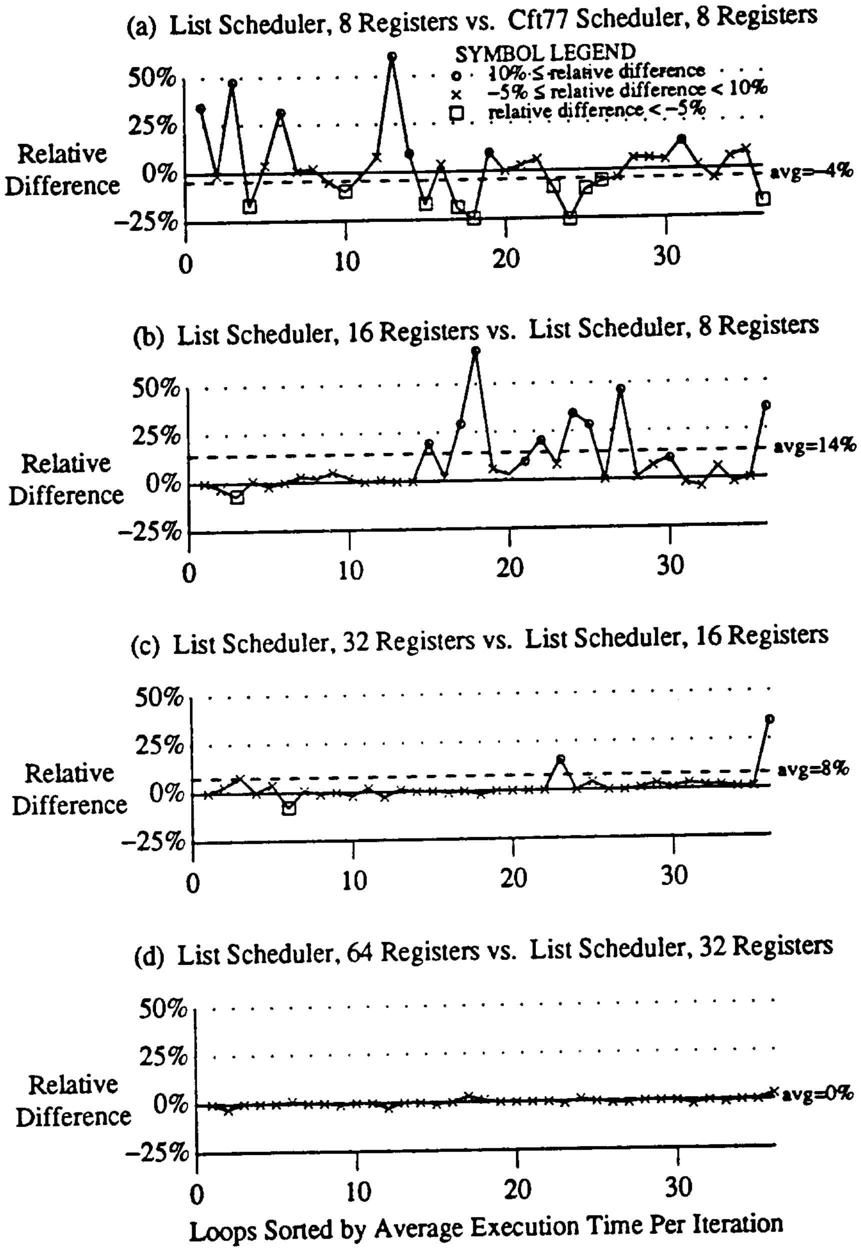
Operations Placed Into Chime Table

LOAD$_4$	LOAD$_1$	$\times_5$		
LOAD$_3$			$+_6$	
		$\times_7$	$+_2$	STORE$_8$

Figure 5.13: Example Showing that List Scheduler Needs a New Heuristic

This figure demonstrates that another heuristic is needed to determine a scheduling order among operations with equal priority. Because the illustrated dependence graph represents one statement, the scheduling order used by both the *cft77* and list schedulers is based on maximal path distance. The operations marked with an asterisk (*) have the same maximal path distance but are scheduled in reverse order for each algorithm. As a result, the execution order produced by the *cft77* scheduler is better than the one produced by the list scheduler.

Operations can be grouped into a ***chain of operations*** which can be executed in one chime using chaining hardware despite RAW dependences. For example, the operations LOAD$_3$, LOAD$_4$, $\times_5$, and $+_6$ form such a chain. The scheduling order used by the list scheduler, however, does not allow this chain to form whereas the order used by the *cft77* scheduler does. A new heuristic that gives higher priority to an operation in a chain that is already partially scheduled would allow the list scheduler to generate the better execution order.


(a) List Scheduler, 8 Registers vs. Cft77 Scheduler, 8 Registers

(b) List Scheduler, 16 Registers vs. List Scheduler, 8 Registers

(c) List Scheduler, 32 Registers vs. List Scheduler, 16 Registers

(d) List Scheduler, 64 Registers vs. List Scheduler, 32 Registers

Figure 5.14: Performance Comparisons of Various *Scheduler&Register* Combinations

These graphs compare the performances of various *scheduler&register* combinations whose results are used to choose the most cost-effective one. The loops in graphs (b) and (c) that show a performance degradation of 8% and 9% (as indicated by the □'s) are unexpected because using more registers should result in a relative difference in performance no worse than -5%. A closer examination of the assembly code for these two loops revealed that the only difference between using different numbers of registers is that the data is placed in different locations in memory, possibly causing conflicts to occur in different memory banks. Although memory-bank conflicts can cause up to a 5% variance in performance, these two data points indicate that sometimes the variance can be greater. Section 5.2.1 describes the performance metrics and the basic layout of these graphs, and Figures 5.18 and 5.19 list the execution times plotted in them.

list scheduler produces more than a 5% performance *degradation* for 10 loops, 8 of which show a performance degradation of 10% or more. Because other loops show significant *improvement*, the relative difference in performance for the entire workload is small (−4%), despite the significant degradation in performance of many individual loops. Thus, Figures 5.11 and 5.14a together show that, as long as an appropriate scheduling algorithm is used, more than 8 vector registers are needed to effectively use the functional units of the Cray Y-MP.

The last three graphs in Figures 5.14 demonstrate how the performance of the list scheduler is affected by the number of registers in the hardware. Increasing the number of registers from 8 to 16 improves the performance of the workload by 14% and improves by more than 10% the performance of 10 out of the 36 loops. Doubling the number of registers from 16 to 32 results in a fair improvement (8%) in the workload performance but provides more than 10% performance improvement for only 2 of the loops. Finally, using 64 registers instead of 32 results in relatively little improvement to performance in either the workload or individual loops. Because the greatest gain in performance is obtained by increasing the number of registers from 8 to 16, I conclude that 16 vector registers is enough to effectively use the functional units in the Cray Y-MP.

Figure 5.15, which is a complementary graph to those displayed in Figure 5.14, shows the improvement in performance of the list scheduler using 16 vector registers relative to the *cft77* scheduler using 8. The graphs in Figure 5.14 show the relative change in performance as *scheduler&register* combinations progressively change from the *cft77* scheduler using 8 registers to the list scheduler using 64. Whereas this illustrates *which* intermediate combination achieves the greatest gain in performance, Figure 5.15 indicates the actual improvement to performance over the *cft77* scheduler using 8 registers. The overall improvement (9%) of the list scheduler using 16 registers is not as high as that (18%) of the same scheduler using 64 registers. This is because the largest loop, which represents 24% of the execution time of the workload, can still benefit tremendously by using more than 16 registers. However, the overall improvement of the list scheduler using 16 registers is the same as that of the *cft77* scheduler using 64. Moreover, the number of individual loops (15) showing significant improvement in the former combination is noticeably higher than that (5) of the latter combination, and almost matches that (17) of the list scheduler using 64 registers. Consequently, although the list scheduler using 16 registers falls just short of the performance criteria for justifying more hardware, this *scheduler&register* combination comes reasonably close and requires the least increase in hardware for the greatest gain in performance.

The improvements to performance reported in this study are only for the vectorizable portions of a program. To determine the improvement in performance of an entire program, we can use Amdahl's Law to calculate a program's speedup S_k as a function of both vector speedup k and fraction of time spent executing vectorizable code f:

$$S_k = \frac{1}{1 - f + \frac{f}{k}}$$

Under the assumption that the CRI workload is representative of the vectorizable portion of a program, I use the improvement in performance over the entire workload (1.09) as the

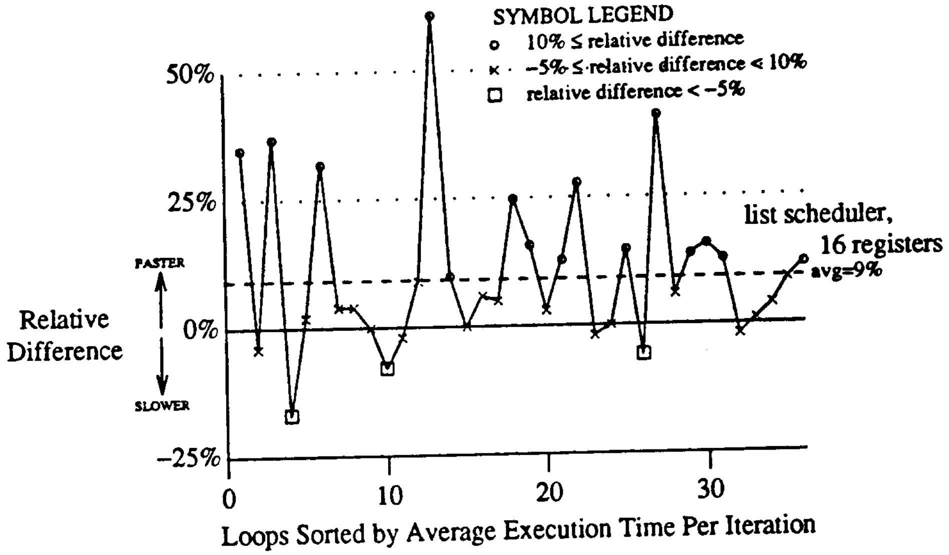


Figure 5.15: List Scheduler Using 16 Registers Vs. *Cft77* Scheduler Using 8 Registers

To show the improvement to performance of a cost-effective number of registers, this graph compares the performance of the list scheduler using 16 vector registers to that of the *cft77* scheduler using 8 vector registers. Section 5.2.1 describes the performance metrics and the basic layout of this graph, and Figures 5.18 and 5.19 list the execution times plotted in it.

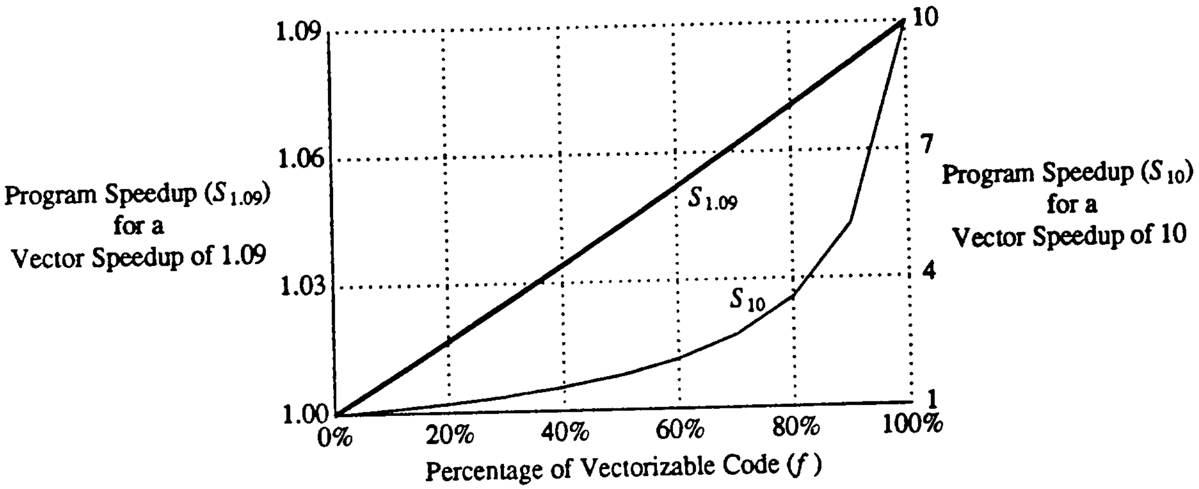


Figure 5.16: Performance Improvement of a Program

Using Amdahl's Law, this graph shows that, although the performance improvement of a program declines with decreasing portions of vectorizable code, the decline in performance for a small vector speedup is not as rapid as the decline for a large vector speedup. The curve labeled S_{10} shows the program speedup when vector speedup is 10, a speedup which typically results when using vector hardware rather than scalar hardware. The curve labeled $S_{1.09}$ gives the program speedup when vector speedup is 1.09, which is the average improvement in performance of the CRI workload when using the list scheduler with 16 vector registers rather than the *cft77* scheduler with 8 registers.

For a large vector speedup of 10, program speedup is *inversely proportional* to $1 - f$, the amount of *non-vectorizable* code, which results in a rapid decline in overall program speedup as the amount of vectorizable code decreases. In contrast, for a small vector speedup of 1.09, program speedup is *linearly proportional* to f, the amount of *vectorizable* code, which results in only a near-linear decline in overall program speedup as f decreases.

vector speedup. Figure 5.16 shows that the improvement in program performance when using the list scheduler and 16 vector registers degrades only linearly as the amount of vectorizable code in a program drops. Because a near-linear decline in program speedup occurs for vector speedups of less than 2, this rate of decline would still occur even if the vectorizable portion of a program consisted mainly of loops with the larger improvements. This is in contrast to typical applications of Amdahl's Law that show a precipitous drop in program speedup because such analyses use the much higher vector speedup of 10. As a result, although the improvement indicated by the study in this chapter is small relative to typical applications of Amdahl's Law, its impact on a program's overall improved performance does not drop off as rapidly with decreasing amounts of vectorizable code.

Finally, when I explained the issues involved in this study, I made two observations that my performance data also substantiates. First, in justifying the inclusion of large loops in the CRI workload, I stated that these were the ones more likely to require more registers for enhanced performance because these are more likely to have more operations that can execute in parallel. Figure 5.14b supports this intuition in that only the loops on the right-hand side of the graph show any significant improvement when the number of registers is doubled from 8 to 16. These loops are the larger ones because loops with more operations will tend to execute longer, and longer-executing loops are plotted on the right-hand side of the graphs.

Second, in explaining why more registers are needed, I stated that avoiding register spills is not an adequate reason because, once enough registers are provided, execution of register-spill code has little impact on the execution time. To demonstrate this, I reproduce Figures 5.14b and 5.14c in Figure 5.17 and use boxes to mark loops whose minimal register requirement is greater than the base configuration for that graph. Because I did not include any mechanism for matching register usage with what the hardware provides, the same execution order and hence, the same minimal register requirement for a loop are produced by the list scheduler, irrespective of the actual configuration of the register file. If more registers are required than the hardware can provide, the *cft77* compiler will generate extra instructions to spill registers. For example, as Figure 5.17 illustrates, register-spill code is generated for 17 loops when only 8 vector registers are provided in the hardware, and for 5 loops when 16 registers are available. Execution of this extra code could not be effectively hidden when using only 8 registers, as indicated by the drastic improvement in performance of several loops when the number of registers is doubled to 16. In contrast, execution of register-spill code has little impact on performance for 3 of the 5 loops when using 16 registers, as indicated by the lack of improvement in performance for these loops when the number of registers is doubled to 32. Hence, once enough registers are provided, register spills can be accommodated with little impact on performance, and the better reason for adding more registers is to allow more aggressive scheduling so that more parallelism occurs.

Figures 5.18 and 5.19 give the per-iteration execution times and minimal register requirements for each loop. These data are used in the various graphs I presented in this section.

5.5 Related Work

Three other groups of researchers have investigated scheduling algorithms for vector architectures that implement a restrictive form of chaining and for ones that implement fully flexible chaining. (Chaining hardware is described in Chapter 2, *Fundamentals of Vector Architectures* on page 11). The Cray-1 is an example of a vector architecture with restricted chaining and the Cray X-MP and Y-MP are examples of ones with fully flexible chaining. Arya modeled the problem of finding an optimal execution order as an integer programming problem, a technique which is expected to take considerably longer to execute than heuristic approaches such as list scheduling [7]. Bernstein, Boral, and Pinter extended the work of Aho and Johnson to apply to vector architectures and presented an optimal algorithm that always generates an order that executes in the shortest time for a given number of vector registers in hardware [10, 3]. However, this algorithm is applicable only for dependence graphs that are trees, which have no common subexpressions. Finally, Tang

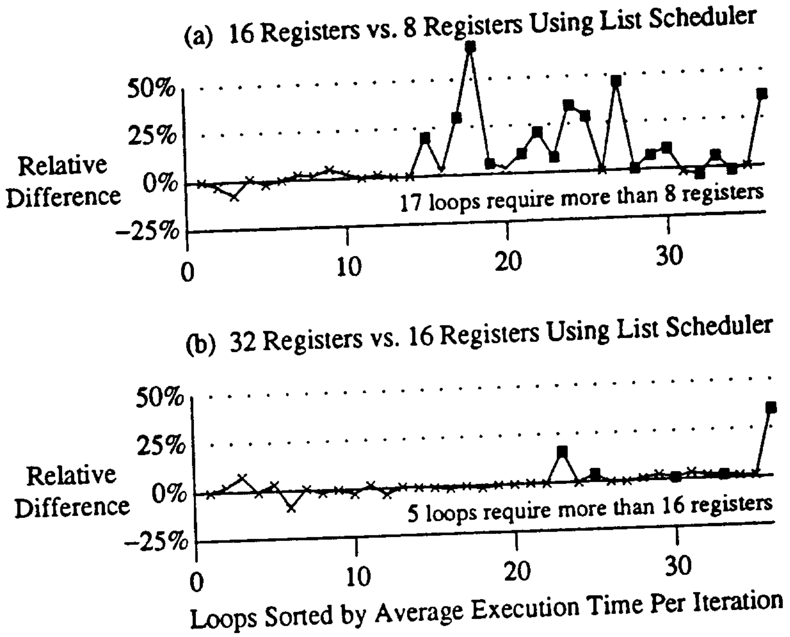


Figure 5.17: Register Usage of the List Scheduler

These graphs show when execution of code for register spilling can and cannot be effectively hidden. A box (■) marks a loop whose minimal register requirement exceeds the register capacity of the base configuration of a graph; a cross (×) marks all other loops. Section 5.2.1 describes the performance metrics and the basic layout of these graphs, and Figures 5.18 and 5.19 list the data used in them.

LOOP	*cft77* SCHEDULER		LIST SCHEDULER				$\mathcal{R}^\dagger$	STATIC
NUM-	# Vector Registers		Number of Vector Registers					UPPER
BER	8	64	8	16	32	64		BOUND
1	2.50	2.48	1.85	1.85	1.85	1.85	3	1.5
2	2.88	2.75	2.92	3.00	2.92	3.02	4	1.0
3	3.96	3.94	2.68	2.89	2.68	2.68	4	2.0
4	4.20	4.23	5.09	5.04	5.06	5.04	6	3.0
5	7.36	7.42	7.05	7.20	6.95	6.95	6	5.5
6	8.17	8.17	6.19	6.19	6.75	6.69	5	4.0
7	8.39	8.14	8.31	8.09	8.00	8.01	6	9.0
8	8.66	8.70	8.49	8.33	8.45	8.45	7	7.0
9	9.07	9.26	9.56	9.11	9.11	9.20	7	8.0
10	9.12	9.19	10.08	9.86	10.08	10.08	7	8.0
11	10.86	10.78	11.03	11.03	10.80	10.80	4	5.0
12	11.00	11.02	10.20	10.06	10.38	10.66	5	10.0
13	15.36	15.34	9.56	9.56	9.49	9.49	6	7.0
14	17.67	17.35	16.04	16.06	16.06	16.10	8	11.0
15	17.73	17.33	21.36	17.74	17.81	18.05	12	16.0
16	19.43	18.95	18.66	18.26	18.49	18.43	5	16.0
17	19.72	18.92	24.43	18.75	18.75	18.27	10	15.0
18	20.20	17.30	27.02	16.11	16.37	16.27	13	14.0

†minimal register requirement

Figure 5.18: Performance Data for the 18 Shortest Loops

For the 18 shortest loops in the CRI workload, this table gives the average execution time per iteration using two scheduling algorithms with a different number of vector registers in the hardware. Descriptions of the *cft77* and list schedulers are given in Figures 5.8 and 5.9 (on pages 93 and 94), respectively. The column labeled $\mathcal{R}$ lists the minimal register requirement of the execution order produced for a loop by the list-scheduling algorithm. The rightmost column gives the minimal execution time for a loop based on functional-unit usage. Data for the other half of the CRI workload is given in Figure 5.19.

LOOP NUMBER	*cft77* SCHEDULER		LIST SCHEDULER				$\mathcal{R}^{\dagger}$	STATIC UPPER BOUND
	# Vector Registers		Number of Vector Registers					
	8	64	8	16	32	64		
19	20.58	20.09	18.73	17.72	17.73	17.73	10	12.0
20	21.19	20.58	21.13	20.60	20.60	20.65	5	29.0
21	21.70	21.95	21.12	19.16	19.11	19.17	11	17.0
22	24.11	24.36	22.79	18.77	18.77	18.73	14	14.0
23	24.77	24.66	27.14	25.18	21.64	21.81	18	18.0
24	29.82	29.70	40.28	29.81	29.81	29.61	13	25.0
25	30.35	29.49	33.90	26.29	25.24	25.24	20	22.0
26	33.26	33.54	35.50	35.51	35.51	35.80	7	32.0
27	34.39	34.50	35.85	24.43	24.47	24.77	13	17.0
28	36.69	37.10	34.77	34.59	34.10	34.25	10	24.0
29	40.51	36.91	38.06	35.47	34.47	34.47	9	25.0
30	47.32	41.19	45.24	40.83	40.30	40.41	18	37.0
31	49.13	43.35	42.61	43.45	42.00	42.68	6	31.0
32	58.82	61.80	57.75	60.06	58.83	58.83	9	80.0
33	67.59	66.28	70.98	67.00	65.78	66.32	23	60.0
34	85.17	85.89	80.26	81.51	80.46	80.71	9	78.0
35	141.59	135.17	129.33	129.34	127.80	127.27	6	82.0
36	303.41	223.14	369.38	269.98	199.53	193.31	57	178.0
WORK-LOAD	1266.68	1160.97	1325.34	1158.83	1076.15	1071.80		924.0

†minimal register requirement

Figure 5.19: Performance Data for the 18 Longest Loops and the Entire Workload

For the 18 longest loops in the CRI workload, this table gives the average execution time per iteration using two scheduling algorithms with a different number of vector registers in the hardware. Descriptions of the *cft77* and list schedulers are given in Figures 5.8 and 5.9 (on pages 93 and 94), respectively. The column labeled $\mathcal{R}$ lists the minimal register requirement of the execution order produced for a loop by the list-scheduling algorithm. The rightmost column gives the minimal execution time for a loop based on functional-unit usage. Data for the other half of the CRI workload is given in Figure 5.18.

and Davidson investigated the impact on performance of vector scheduling techniques and architectural features of the Cray-1 and Cray X-MP [107].

Two other research groups, Tang, Davidson, and Tang as well as Eisenbeis, Jalby, and Lichnewsky have both presented polycyclic vector scheduling algorithms for the Cray-2, which does not implement chaining [108, 32]. This scheduling technique compensates for the lack of chaining to improve performance but at the expense of using more registers. Both research groups indicate that more than 8 vector registers would improve the performance of the Cray-2 but do not investigate how many more would be enough.

The above studies differ from mine in that the researchers have emphasized the algorithm aspect and have assumed that the hardware is fixed. In contrast to these studies, Mangione-Smith, Abraham, and Davidson compared the performance of a scheduling algorithm using different hardware configurations [81]. The purpose of their study, which is a continuation of the work by Tang *et al.* and Eisenbeis *et al.*, is to determine the dimensions of a vector register file that allows all the loops in their workload to execute in the shortest time when using a polycyclic schedule. Because their technique for determining the minimal register requirement of a polycyclic schedule relies on the special structure of a dependence graph that is a tree, the loops used in their study are all trees. In contrast, 75% of the dependence graphs in the CRI workload contain common subexpressions. To determine the minimum number of vector registers needed for a given loop, their algorithm implicitly enumerates all polycyclic schedules for that loop. Based on these exhaustive searches, the researchers conclude that optimal performance for their workload is possible when using a vector register file with 32 registers and 4 elements each, or one with 16 registers and 16 elements per register.

5.6 Summary

In this chapter, I answered the question posed in the opening paragraph of this dissertation:

> How many vector registers are enough to effectively use
> the functional units of the Cray Y-MP vector processor?

To do this, I examined how different numbers of vector registers affect the performance of the Y-MP. I did not experiment with the number of elements per vector register because I hypothesized that this would not affect performance significantly. An undertaking for the future is to verify that increasing the number of vector registers improves performance more than does increasing the number of elements per vector register. In addition to experimenting with register capacity, I also examined how different algorithms for scheduling instructions affect performance.

An integral aspect of this investigation was the interaction between theoretical knowledge and experimental data. The former guided the investigation by providing hypotheses that were verified by experimentation. For example, when the *cft77* scheduler performed so poorly using 64 registers, I correctly hypothesized that a different scheduler was needed rather than erroneously concluding that 8 registers were enough because I understood how scheduling instructions affects register usage and execution time. Moreover, this knowledge led me to hypothesize that more registers could improve performance by allowing more parallelism to occur. The computational impracticality of enumerating all possible execution order for a loop was another hypothesis that was verified by experimentation, which in turn led to the use of scheduling algorithms that are based on heuristics. Correctly interpreting some results is another example of the interaction between knowledge and data. For example, The fact that the scheduling algorithms I used are based on heuristics explains why the list scheduler does worse than the *cft77* scheduler on some loops.

Theoretical knowledge alone, however, does not provide enough information to make decisions; quantitative results are also needed. (Such results must be obtained by experimentation because the current state of the art in performance analysis does not include analytical formulas that can produce such data.) For example, knowing that finding a definitive answer to the above question is an NP-hard problem does not necessarily rule out choosing this method. However, after producing experimental data that showed that current computers or even ones in the near future would require *years* to find the definitive answer for the majority of the loops in the CRI workload, I chose an approach that is computationally practical but cannot give a definitive answer.

Although theoretical knowledge provides a qualitative ranking, experimental data is necessary to quantify *how much* better and *how often* performance is improved. For example, the descriptions of the two scheduling algorithms indicate that the list scheduling should outperform the *cft77* scheduler because of their differing goals, which are summarized in Figure 5.10 (on page 95). Nonetheless, both algorithms rely on heuristics, which can be either exploited or thwarted by different dependence graphs; Figures 5.3 (on page 77), for example, demonstrates how the list scheduler generates a better execution order than does the *cft77* scheduler, and Figure 5.12 (on page 98) illustrates the reverse. What these qualitative descriptions and examples do not indicate is *how much* nor *how frequently* one scheduling algorithm outperforms the other.

Another example demonstrating how quantitative results sharpen a qualitative ranking concerns the usage of registers. Although more registers should allow more parallelism to occur which, in turn, allows the functional units to be used more often, a quantitative study is needed to indicate not only *how many* registers improve the performance by *how much* but also *how frequently* such performance improvement occurs. To justify changing the scheduling algorithm in the *cft77* compiler or increasing the number of registers, a quantitative study using a representative set of loops is needed to show not only that the improvement to performance is large enough, but also that the improvement occurs often enough.

Figure 5.20 summarizes the experimental data I produced to justify the proposed changes. This figure shows the performance of various *scheduler&register* combinations relative to that of the *cft77* scheduler using 8 vector registers. A *scheduler&register* combination improves performance *significantly* if it improves performance for *both* the entire workload and a majority of the individual loops. The rationale for prescribing both these criteria is given in Section 5.2.1. Based on these criteria for acceptable improvement in performance, the *cft77* scheduler does not improve performance significantly for the CRI workload even with an abundance of vector registers, whereas the list scheduler using only 16 regsiters does. Although the list scheduler using 32 registers further improves the performance of the entire workload, the distribution of improvement in performance for individual loops does

SCHEDULING ALGORITHM AND NUMBER OF REGISTERS		RELATIVE DIFFERENCES IN PERFORMANCE (ρ)				FIGURE FOR PERFORMANCE GRAPH
		DISTRIBUTION FOR INDIVIDUAL LOOPS			ENTIRE WORK-LOAD	
		$\rho < -5\%$	$-5\% \leq \rho < 10\%$	$10\% \leq \rho$		
cft77	64	0	31	5	9%	5.5 (p. 82)
list	8	10	19	7	-4%	5.14 (p. 100)
list	16	3	18	15	9%	5.15 (p. 102)
list	32	3	16	17	18%	—
list	64	3	16	17	18%	5.11 (p. 96)
static upper bound	∞	3	2	31	37%	5.4 (p. 81)

Figure 5.20: Summary of Performances for Various *Scheduler&Register* Combinations

This table summarizes the performance of various *scheduler&register* combinations relative to that of the scheduling algorithm implemented in the 1990 version of the *cft77* compiler using 8 vector registers. The raw data for this performance summary are given in Figures 5.18 and Figures 5.19, and Section 5.2.1 (pages 83 to 87) explains the metrics used to compare the performance of a *scheduler&register* combination to that of the *cft77* scheduler using 8 registers. Descriptions of the *cft77* and list scheduling algorithms are given in Figure 5.8 and 5.9 (on pages 93 and 94), respectively. Because these two scheduling algorithms are based on heuristics, it is possible that either one outperforms the other for individual loops. For example, when using 64 registers, the *cft77* scheduler outperforms the list scheduler by more than 5% for 3 loops, whereas the list scheduler outperforms the *cft77* scheduler for other loops. Section 5.1.2 (pages 79 to 83) describes how the static upper bound is calculated and also explains why this "upper bound" has worse performance than the *cft77* scheduler for some of the loops.

not change significantly for this *scheduler&register* combination relative to the distribution for the same scheduler using 16 registers. Hence, because the improvement in performance does not justify doubling the number of registers from 16 to 32, the answer to the question posed in the opening paragraph of this dissertation is 16 vector registers, as long as an appropriate scheduling algorithm is used.

In this chapter, I emphasized performance in order to show that more than 8 vector registers can improve performance by more than 10%. Because increasing the number of registers is a costly endeavor, I will examine the cost of doing so in the next chapter. Furthermore, in order to balance the improved performance with increased cost, I will also investigate a special organization of 16 vector registers that requires only a 10% increase in hardware.

Chapter 6

Bus Usage and Register Assignment

In the previous chapter, I showed that using more than 8 vector registers significantly improves performance. However, because this analysis of performance excludes the cost of implementing more vector registers I explain in this chapter why adding vector registers in a straightforward fashion is ill-advised and investigate a different organization for a vector register file that is more cost-effective.

To introduce what this different organization is, I first analyze the hardware cost of implementing more registers and suggest an organization that is less costly. Implementing a new register file, however, is viable only if it can be used and used without degrading performance. Accordingly, the bulk of this chapter addresses these concerns of utility and performance. After introducing the new organization, I describe an assignment algorithm I developed that uses vector register files with such an organization. I then present data to evaluate how well my algorithm uses such configurations and finish with a discussion about choosing a cost-effective one for the Cray Y-MP vector processor.

6.1 Cost/Performance Analysis

In the previous chapter, I presented data showing that doubling the number of vector registers from 8 to 16 improved performance by 9%. Although implementing more registers obviously increases the cost of hardware, a general rule-of-thumb is that a new hardware configuration that increases the cost of implementation is acceptable if it improves performance by at least as much as it increases the cost of implementation. To determine whether doubling the number of vector registers is, in fact, an acceptable tradeoff between increased cost and improved performance, in this section I first analyze the hardware cost of implementing more, and then I combine this cost analysis with the performance analysis of the previous chapter to examine the tradeoff between increased cost and improved performance for various configurations of a vector register file. Finally, based on this cost/performance analysis, I outline the goals of this chapter's study.

A vector register file actually consists of multiple banks of registers, each of which has two buses: a read bus to an interconnection that sends operands to a set of functional units, and a write bus from an interconnection that receives results from those functional units (see Figure 2.4 on page 13 in Chapter 2, *Fundamentals of Vector Architectures*). Increasing the size of the register file in a straightforward fashion increases not only the number of register banks but also the size of the interconnections. In Chapter 3, *A Case for Vector Architectures*, I argued that the size of these interconnections is negligible when compared to the registers themselves. So the cost of implementing more vector registers in a straightforward fashion is determined mainly by the register cells. But this cost analysis applies only to a *single-chip* VLSI implementation. A different cost analysis is needed for a *multi-chip* implementation, which is how the Cray Y-MP vector processor is built, where the relative sizes of register banks and interconnections are quite different from their relative sizes in a single-chip implementation.

The function of an interconnection is to transfer large amounts of data between vector registers and functional units. Wawrzynek and von Eicken have shown that in a single-chip, CMOS implementation such functionality is provided relatively easily by using metal lines and multiplexors that take up a negligible amount of extra space [117, 118]. On the other hand, in a multi-chip implementation with 64-bit data buses, this functionality requires numerous pins to provide a physical path between vector registers and functional units.[1] For example, each bus of a vector register requires 64 pins and each input or output bus from a functional unit needs 64 pins. Because large numbers of pins are required and because one chip can accommodate only a relatively small number of pins, many chips are needed to implement these two interconnections. In fact, the number of chips needed to implement both interconnections is 1.5 times the number of chips that implement the 8 vector registers in the Cray Y-MP.

Given that 8 vector registers account for 10% of the chips in an implementation of one Y-MP processor and the accompanying interconnections account for 15% of the chips, doubling the size of the vector register file from 8 to 16 registers in a straightforward fashion results in a 25% increase in the chip count. Hence, in a multi-chip implementation, such as that of the Cray Y-MP vector processor, more than half the cost of implementing more registers is due to increasing the size of the interconnections. Because doubling the number of vector registers produces a 9% improvement in performance, using 16 does not appear to be an acceptable tradeoff between increased cost and improved performance.

(Although this analysis covers only the cost of a processor and ignores the cost of the memory and I/O systems of a computer, the main purpose for this cost analysis is to motivate the investigation of a special organization of registers that reduces the cost of adding more registers. As I demonstrate later in this chapter, in addition to being less expensive to implement, this special organization has comparable performance to a traditional one with the same number of registers. Consequently, although the cost of adding more registers relative to an entire computer is less significant, the cost/performance ratio can still be improved when using this special organization.)

One possible solution for improving the tradeoff between increased cost and performance gain is to change the number of elements per vector register, which is also known as the *vector length*. Halving the vector length from 64 to 32 while doubling the number of vector registers from 8 to 16 results in a 15% rather than 25% increase in chip count. Halving the vector length again to 16 results in only a 10% increase in chip count. A reason-

[1] I am indebted to Wei-Chung Hsu for this information.

able vector length, however, is needed to hide both the latency of vectorizable operations and the execution of scalar operations. Moreover, when functional units are reserved for some number of clock periods greater than the vector length, as is the case in the Cray Y-MP processor, the performance penalty of this extra delay can be amortized by using an appropriately long vector length. As a result, although a vector length of 16 appears to be an acceptable tradeoff between increased cost and improved performance, sustainable performance may be adversely affected when the vector length is shortened so much.

Because the size of an interconnection is determined by the number of buses attached to it, another possible solution for improving the cost/performance tradeoff is to have more than one vector register share a bus. As Figure 6.1 shows, although adding more vector registers still increases the hardware cost, the rise in cost is much less for this new configuration than for a traditional one with the same number of registers. Another way to describe this new configuration is that it partitions the vector registers into banks, where each bank has its own read and own write buses. Based on this description, I call such a configuration a *partitioned vector register file.*[2]

Both the Ardent Titan, a commercial vector computer, and the Fujitsu VPU, a VLSI implementation of a vector processor, have implemented a partitioned vector register file. In addition to being partitioned, these vector register files are also reconfigurable in that the number of vector registers and their vector length can be varied under software control. The vector register file in the Ardent Titan can be viewed as a partitioned one with four banks, each of which contains 2048 elements [31]. The number of vector registers in a bank can vary from 1 to 2048, depending upon the vector length which can vary from 2048 down to 1. The vector register file in the Fujitsu VPU also has four banks, each of which contains 256 elements [64]. The number of vector registers per bank can vary from 2 to 16 depending upon the vector length which can vary from 128 down to 16.

Figure 6.2 shows a rudimentary analysis of the tradeoff between increased cost and improved performance among various configurations of partitioned register files, where the performance of a partitioned register file approximates the performance of a traditional one with the same number of registers. Based on this approximation for performance, using 16 registers appears to be an acceptable cost/performance tradeoff if 4 or 2 registers were to share a bus. Likewise, a partitioned register file with 32 registers and 4 or 8 buses, which improves performance by nearly 18%, is nearly an acceptable tradeoff between increased cost and improved performance, although it is not as cost-effective as one with 16 registers. In contrast, using 64 registers will never achieve an acceptable cost/performance tradeoff because, even though performance is improved by about 18%, the relative increase in cost is, at best, more than three times higher no matter how many registers share a bus. One other interesting configuration to consider from a cost standpoint is a partitioned register file with 4 buses and 8 registers. This yields an acceptable cost/performance tradeoff *as long as* performance does not degrade more than 7.5%; the best result is achieved when no performance degradation occurs at all.

Based on this initial cost/performance analysis, using a partitioned register file

[2] Because I always deal with vector objects in this chapter, I use the term *element* to refer to a register in a vector register, the term *vector register* interchangeably with *register*, and the term *partitioned vector register file* interchangeably with *partitioned register file*.

		NUMBER OF REGISTERS					
		4	8	16	32	64	128
	4	-12%	-7%	**3%**	23%	63%	143%
NUMBER	8		0%	**10%**	30%	70%	150%
OF BUSES	16			25%	45%	85%	165%
	32				75%	115%	195%

Figure 6.1: Relative Differences in Chip Count Among Vector Register Files

This table gives the relative difference in chip count among various traditional and partitioned vector register files with different numbers of buses and registers and with varying amounts of partitioning. The number of elements per vector register is kept constant at 64. Figure 6.13, which is presented in a later section, gives a cost analysis of vector register files where vector length is varied.

The difference in chip count is relative to the total chip count (C) for the current implementation of a Cray Y-MP processor, which uses 8 buses and 8 registers. Given that 8 vector registers account for 10% of the total chip count and the number of chips needed to implement the two interconnections is 50% more than the number needed for the 8 vector registers, the relative difference in cost of a partitioned register file with B buses and R registers is:

$$\frac{(\frac{B-8}{8} \times 1.5 + \frac{R-8}{8})\, 0.10C}{C} = \left(\frac{B}{8} \times 1.5 + \frac{R}{8} - 2.5\right) 0.10$$

Because the difference in cost is relative to the cost for 8 buses and 8 registers, using 4 buses and 16 or fewer registers actually results in a decrease in the overall chip count.

Based on data in Figure 6.2, which shows the tradeoffs between increased cost and improved performance for partitioned register files with 16, 32, and 64 registers, the two relative chip counts highlighted in bold above correspond to configurations that could provide an acceptable cost/performance tradeoff.

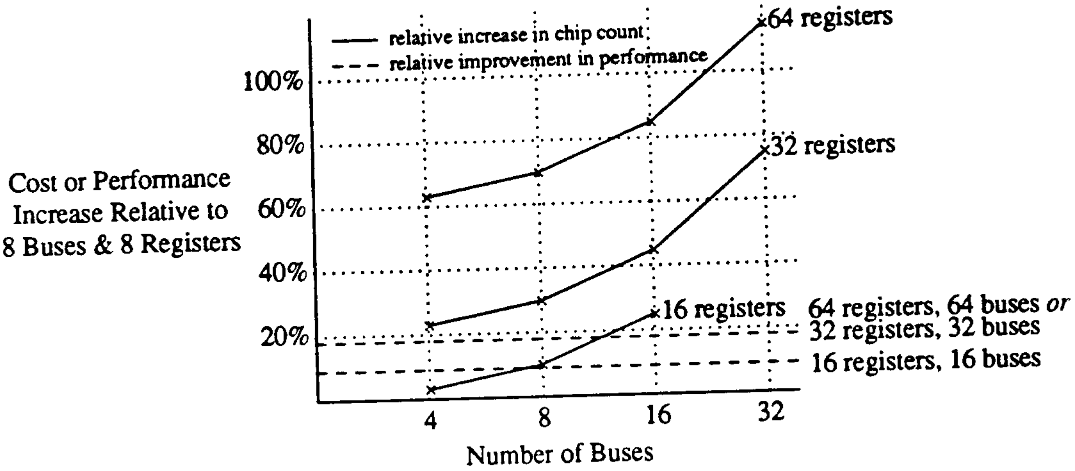


Figure 6.2: Cost/Performance Comparisons for Various Partitioned Register Files

This figure combines cost data from Figure 6.1 and performance data summarized in Figure 5.20 from the previous chapter in order to compare the relative increase in chip count with the relative improvement in performance when implementing more than 8 registers.

In addition to showing the cost of implementing traditional vector register files where each register has its own bus, I have also shown the cost of implementing various partitioned register files where more than one register shares a bus. Because of this sharing, a partitioned register file imposes more restrictions on accessing vector registers. Moreover, because each performance line is based on using a traditional vector register file, the performance in this analysis represents an upper bound on the improvement possible for the indicated number of registers.

A general rule-of-thumb for justifying a more costly implementation is if a comparable increase in performance can also be achieved. The above figure shows that implementing 64 registers is never cost-effective, whereas using 8 or 4 buses with 16 registers appears to be an acceptable tradeoff between increased cost and improved performance.

with 16 registers appears to be an acceptable tradeoff between increased cost and improved performance. But there is little point in implementing a partitioned register file if it can't be used. Moreover, in the cost/performance analysis described above, I assumed that the performance using a partitioned register file is the same as the performance using a traditional one with the same number of registers. This assumption, of course, is not necessarily true. Although a partitioned register file is less costly to implement, such a configuration, because multiple registers share a bus, imposes more restrictions on accessing vector registers. As a result, the performance of a partitioned register file could be less than that of a traditional one with the same number of registers.

These two shortcomings provide several goals for this chapter's study. One goal is to design an algorithm that can use a partitioned register file. Another goal is to evaluate how effective the algorithm is and to determine whether the performance of a partitioned register file is comparable to that of a traditional one with the same number of registers. A third goal of this study, which is appropriate only if the first two are successful, is to provide data that determines whether a partitioned register file with 16 registers, in fact, yields an acceptable tradeoff between increased cost and improved performance.

Finally, to provide an intuitive understanding of my results, I make an important observation that gives a clue to both how and how well a partitioned register file can be used. To use a partitioned register file effectively, I take advantage of the fact that:

> only a subset of simultaneously live values are
> actually used at any given time

For example, in Figure 5.3 (on page 77 in the previous chapter) in the fourth chime of the execution order on the left, only three out of five live values are actually being used. Values that are live at the same time must be stored in different registers to avoid generating spill code. However, if they are never *used* at the same time, they can be stored in registers that share the same bus. For example, using the execution order on the left in Figure 5.3, the values produced by $+_8$ and LOAD_{11} can share the same bus but not the values produced by $+_2$ and LOAD_{11}. In other words, registers store live values, buses transfer values that are being operated on, and there are more live values than active ones at any given time. In the rest of this chapter, I present graphical representations of register and bus usages to model the usage of a partitioned register file, and present data showing that the number of simultaneously active values is sufficiently small that a partitioned register file is an acceptable cost/performance configuration for implementing more registers.

6.2 Assignment Algorithm for a Partitioned Register File

In this section, I describe the algorithm I developed to use a partitioned register file where more than one register shares a bus. Because most people tend naturally to focus on on the execution of operations, they often consider the use of registers (and buses) from the perspective of the operations that use these resources. In other words, an operation reads one or two operands from registers and produces a result that is stored in a different register, which in turn is read as an operand for one or more additional operations.

This point of view about operations, however, does not provide the proper frame of reference for determining how to use registers and buses. To do so, focus must instead be placed upon the operands and results, which are seen as values, each of which is produced by some operation. A value is stored in a register from when it is first produced by an operation to when it is last read by another operation. Moreover, a value is transferred on a bus when it is first produced as a result and for each time that it is read as an operand. This point of view shows that values are assigned to registers and buses to use these resources properly. In other words, using a partitioned register file consists of two assignment problems:

1. assigning values to registers without causing any register conflicts, and

2. assigning values to buses without causing any bus conflicts.

Access conflicts, such as WAR or WAW register dependences, are to be avoided because they add extra chimes to execution time. These two problems are related in that assignments for either one are made under the additional constraint that a fixed number of registers share a bus.

One of the goals of this study is to determine whether the performance of a partitioned register file is comparable to that of a traditional one with the same number of registers. In the context of these assignment problems, this goal is equivalent to determining whether the number of buses must increase with the number of registers in order to avoid any access conflicts. The algorithm I developed thus attempts to minimize the number of buses and registers assigned. Minimizing register usage, however, can increase bus usage, and vice versa. Because increasing the number of buses requires more hardware, higher priority is given to minimizing the number of buses assigned. Hence, my algorithm really consists of two algorithms executed in sequence: the first one assigns values to buses and then, for each assigned bus, the second algorithm assigns values to registers that share a bus.

My algorithm is also influenced by the *cft77* compiler (in which it would execute) and by practical considerations in the hardware. The input to this algorithm is an execution order for a dependence graph, rather than the dependence graph itself, because register assignment occurs after scheduling in the *cft77* compiler. Moreover, because a dependence graph corresponds to a basic block, I consider only local register assignment.[3] Finally, in a practical hardware configuration, the number of registers per bus is fixed and equal to the number of registers divided by the number of buses. This quotient is an independent parameter that allows my algorithm to be used for a partitioned register file with an arbitrary but fixed amount of partitioning. Additionally, to keep the implementation simple, the number of registers per bus is a power of two.

To keep this algorithm simple, I do not consider the possibility of register spilling. Moreover, although the hardware allows assignments of the form `Vi<-Vi op Vj`, my algorithm never uses such assignments; instead, it always assigns a different register for each value read or produced by an operation. I address both these omissions in a later section when I evaluate the effectiveness of this algorithm.

[3] I outline the various types of algorithms for register assignment when I discuss related studies in Section 6.5.

To model the use of a partitioned register file, my assignment algorithm is based on the optimal graph coloring problem, which poses the question:

> What is the minimum number of colors needed to color a graph such that no adjacent vertices are assigned the same color? [46]

The formulation of register assignment as a graph coloring problem was independently developed by Ershov and Chaitin *et al.* for assigning scalar registers globally [35, 18]. When an assignment problem is modeled as a graph coloring problem, the colors represent the resources to be assigned and the graph, called an *interference graph*, indicates when an assignment will result in a conflict. In an interference graph, a vertex is a value and an edge represents a potential resource conflict. In other words, an edge identifies two values that would interfere with each other if they were assigned to the same resource.

To present my assignment algorithm, I first define two interference graphs. Then, I examine the structure of these interference graphs to determine whether this problem is NP-hard. Although coloring an arbitrary graph is known to be NP-hard, coloring interval graphs can be done in polynomial time [46]. After showing that one graph has this special structure and the other does not, I describe how these interference graphs are used to assign values to a partitioned register file, and how the special structure of the one graph guarantees that the minimum number of registers is used per bus.

6.2.1 Two Interference Graphs

In my assignment algorithm, I use two interference graphs: a *live interference graph* for assigning registers, and an *active interference graph* for assigning buses. Figure 6.3 shows a small example of these two graphs and illustrates how coloring them produces an appropriate assignment of values to registers and to buses. Both the live and active interference graphs for a dependence graph have the same set of vertices, namely the values produced by operations in a dependence graph. In the execution order used for the example in Figure 6.3, a value is represented by an arrow, whereas in the corresponding interference graphs a value is identified by the operation that produces it. Although the vertices are the same in associated live and active interference graphs, the set of edges that connect vertices differ.

Whether two vertices are connected in a *live interference graph* depends on register usage. Because two values assigned to the same register will conflict if the values are live at the same time, an edge is placed in a live interference graph between two values that are simultaneously live in order to model how register conflicts can occur. In the execution order of Figure 6.3, all values are live at the same time because their associated arrows occur in the same chime. Hence, there is an edge among all the values in the corresponding live interference graph.

In a similar fashion, whether two vertices are connected in an *active interference graph* depends on bus usage. Because two values assigned to the same bus will conflict if the values are either read or written at the same time, an edge is placed in an active interference graph between two values that are simultaneously read or simultaneously written in order to model how bus conflicts can occur. In the execution order of Figure 6.3, a value is

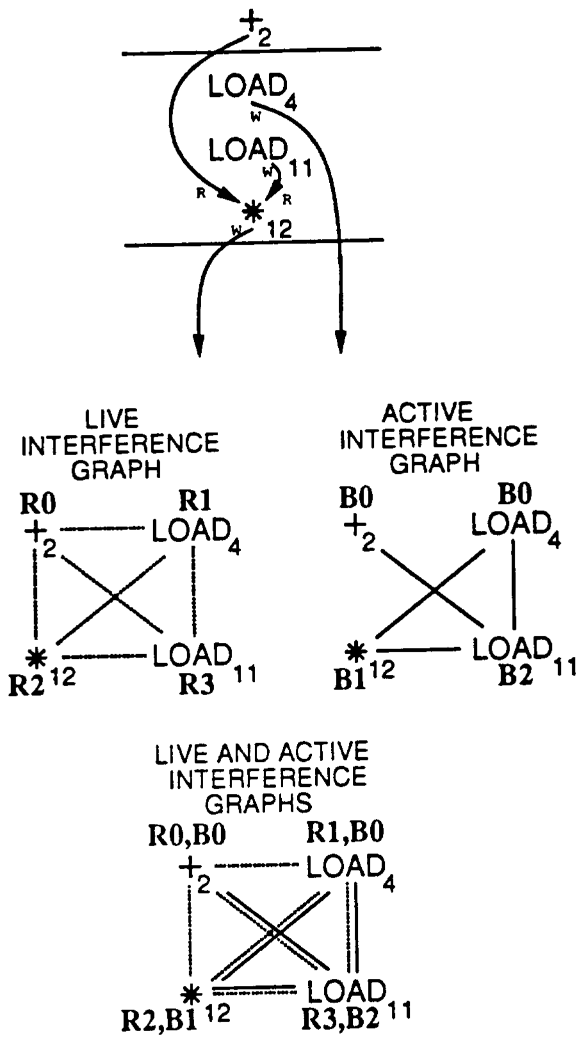


Figure 6.3: Live and Active Interference Graphs

This figure illustrates the part of the live and active interference graphs associated with the second chime of the execution order on the right in Figure 5.3 (on page 77 in the previous chapter). This portion of the execution order is reproduced above. A vertex in an interference graph represents a value, identified in the above figure by the operation that produces it. An edge represents a potential resource conflict between its incidental vertices.

This figure also illustrates how coloring these graphs provides a conflict-free assignment of values to registers and buses, where more than one register can share a bus. The individual interference graphs in the middle of the above figure demonstrate how coloring the live interference graph assigns values to registers (denoted above as the four "colors" R0, R1, R2, and R3) and how coloring the active interference graph assigns values to buses (denoted above as the three "colors" B0, B1, and B2). Once colored, these two graphs together indicate which values can be stored in registers that share the same bus without causing any resource conflicts that increase execution time.

written into a register if the beginning of its arc appears in that chime, and a value is read from a register in a chime if the end of its arc appears in that chime. Because the values $+_2$ and LOAD_{11} are read in the same chime, there is an edge between these two values in the corresponding use interference graph. Furthermore, there are edges between the values LOAD_4, LOAD_{11}, and $*_{12}$ because these are all written in the same chime.

In an active interference graph, there is no conflict between a read that occurs simultaneously with a write. This distinction is important for two reasons. One, it accurately reflects what the hardware can do. Secondly, it reduces the number of edges in the active interference graph, thus increasing the probability that fewer buses can be used.

Figure 6.3 also demonstrates how coloring these two interference graphs gives a conflict-free assignment of values to registers and buses where more than one register shares a bus. To color the live interference graph in Figure 6.3, a minimum of four colors are needed. In other words, a minimum of four registers (denoted R0, R1, R2, and R3) are needed to store these values without any access conflicts among registers. To color the active interference graph in Figure 6.3, a minimum of three colors are needed. In other words, a minimum of three buses (denoted B0, B1, and B2) are needed to use these values without any bus conflicts. Together, the live and active interference graphs indicate which values can be assigned to registers that share a bus. For this example, the values $+_2$ and LOAD_4 can be stored in different registers that share a bus without any register or bus conflicts. Conversely, the values $+_2$, LOAD_{11}, and $*_{12}$ must be stored in different registers that use different buses to avoid access conflicts among registers or buses, a condition that also holds for the values LOAD_4, LOAD_{11}, and $*_{12}$.

Although values that are simultaneously live are not necessarily simultaneously active, the reverse is true; values that are simultaneously active must also be simultaneously live. As a result, the set of edges for an active interference graph is a subset of the edges in the associated live interference graph. This fact allows my algorithm, when it assigns values to buses, to keep track of values that are live at the same time.

When constructing these interference graphs, a compiler needs accurate information about the lifetimes and usage times of values before it can properly assign registers and buses. To obtain this information, a compiler simulates the execution of the vectorizable operations of a dependence graph to determine the time intervals in which values are written and read. In this section, however, I ignore the effects of operational latencies for the sake of simplicity, even though this does not accurately reflect what the hardware actually does. A simulation that uses detailed timing information about the execution of vector instructions produces interference graphs that more closely reflect what actually happens in the hardware, particularly for a processor that supports fine-grain parallelism. More accurate interference graphs in turn allow a compiler to assign registers and buses so that access conflicts will definitely not occur during execution. The accuracy of the length and relative positioning of time intervals does not affect how the assignment algorithm works but rather how many buses and registers are needed to avoid access conflicts. Consequently, an implementation of this assignment algorithm must include the effects of operational latencies when these interference graphs are constructed.

To summarize, live and active interference graphs, which model potential register and bus conflicts respectively, are constructed from an execution order of a dependence graph. These interference graphs have the same set of vertices, which are the values produced by operations in the dependence graph. However, in a live interference graph, there are edges between values that are live at the same time, whereas in an active interference graph there are edges between values that are used in the same way at the same time.

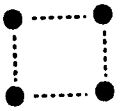
6.2.2 Structure of Live and Active Interference Graphs

Although coloring an arbitrary graph is known to be NP-hard, coloring interval graphs can be done in polynomial time [46]. Hence, to determine whether this particular assignment problem is NP-hard, in this subsection I examine the structure of the interference graphs by using alternative representations of these graphs. The traditional diagram of a graph, which uses symbols for the vertices and lines for the edges, ignores the temporal condition when values are live or active with respect to each other. Alternative representations, which are illustrated in Figure 6.4, capture this timing information to reveal any special structure of these interference graphs.

A live interference graph is an *interval graph*,[4] so named because it can be represented by a set of intervals on the real line:

> Given a collection of intervals $I_1, \ldots, I_n$ on the real line, an ***interval graph*** is a graph where each vertex represents an interval, and there is an edge between vertices i and j if and only if the intervals I_i and I_j overlap [49].

When drawn as an interval graph, the lifetime of the i^{th} value is represented by the interval I_i; when the i^{th} value is produced or last read is indicated by the placement of I_i on the real line. Figure 6.4 shows a live interference graph drawn both in the traditional manner and as an interval graph. Not all graphs can be drawn as interval graphs. For example, the following graph cannot be represented by a set of intervals on the real line in the manner described above nor can it be constructed as the live interference graph for any aependence graph:



An active interference graph, on the other hand, is an ***intersection graph***:

> Given a collection of sets $S_1, \ldots, S_n$ of intervals on the real line, an ***intersection graph*** is a graph where each vertex represents a set, and there is an edge between vertices i and j if and only if some intervals in S_i and S_j overlap [82].

When drawn as an intersection graph, the set S_i contains the time intervals in which the i^{th} value is either written or read, and the placement of these intervals on the real line indicates when the i^{th} value is actually active. Figure 6.4 shows an active interference graph drawn both in the traditional manner and as an intersection graph.

Unfortunately, an intersection graph has no special structure because any arbitrary graph can be described as an intersection graph:

[4] In graph theory, the name of a graph typically reflects some special structure in that graph, whereas, for compilation purposes, the name of a graph commonly reflects what that graph represents.

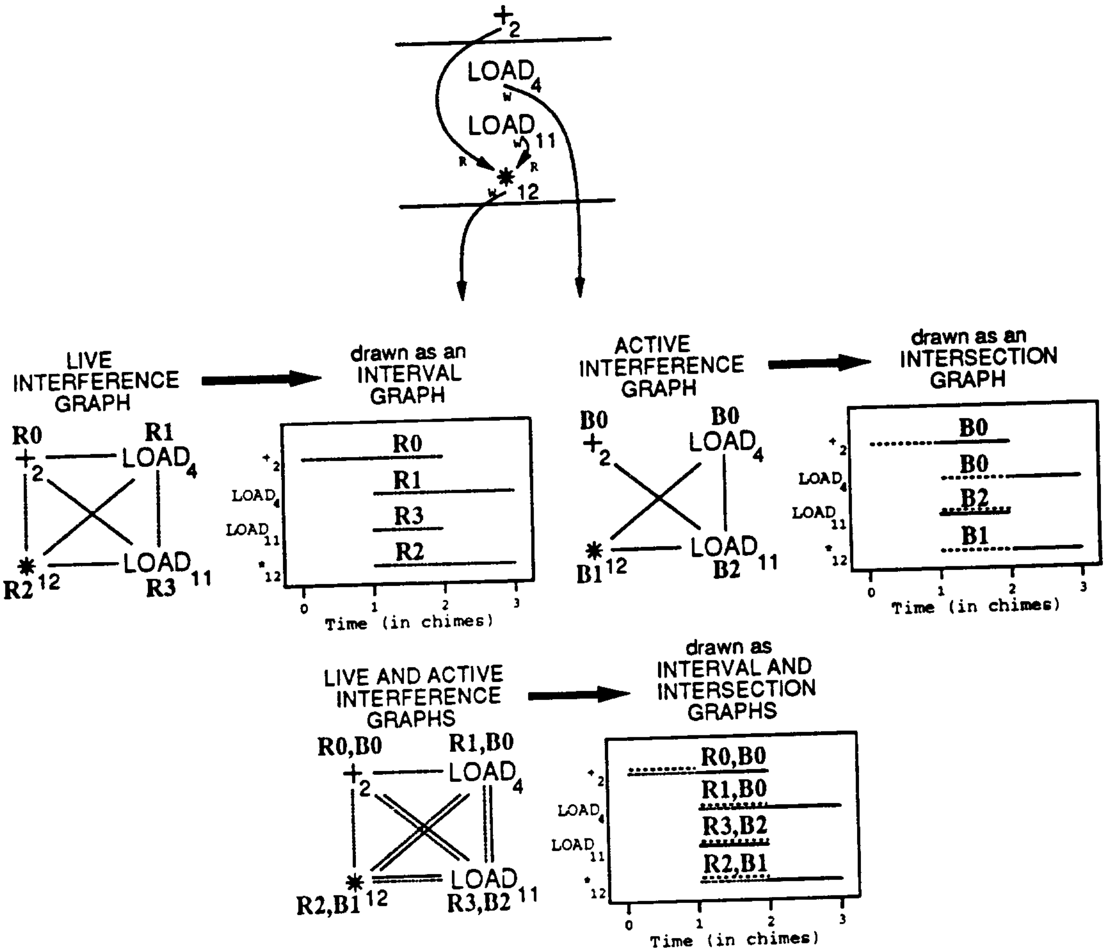
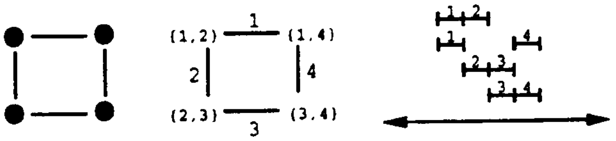

+2
LOAD4
LOAD11
W
R
*12
LIVE INTERFERENCE GRAPH
drawn as an INTERVAL GRAPH
R0
R1
+2
LOAD4
*12
LOAD11
R2
R3
Time (in chimes)
0
1
2
3
ACTIVE INTERFERENCE GRAPH
drawn as an INTERSECTION GRAPH
B0
B1
B2
LIVE AND ACTIVE INTERFERENCE GRAPHS
drawn as INTERVAL AND INTERSECTION GRAPHS
R0,B0
R1,B0
R2,B1
R3,B2

Figure 6.4: Alternative Representations for Live and Active Interference Graphs

This figure shows how the live and active interference graphs in Figure 6.3 can be drawn as a collection of time intervals on the real line. These alternative representations capture the timing information ignored by the traditional drawing of a graph. A line in the graphs illustrated here represents a different component of a graph; a line in the traditional drawing represents an edge, whereas one in an interval or intersection graph represents a vertex. To draw an active interference graph as an intersection graph, I use a dashed interval to represent when a write occurs and a solid interval to represent when a read occurs. Edges in interval and intersection graphs are not explicitly drawn in but are, instead, implicitly represented by overlapping intervals. In the above figure, I ignore the effects of operational latencies for the sake of simplicity and assume that values active in the same chime are active at exactly the same times.

> Any finite graph can be considered the intersection graph of a collection of sets: an edge is an element in the sets and the set associated with a vertex contains the edges incident to that vertex [82].

For example, the following figures represent three different forms of the same graph where a number labels an edge and a set labels a vertex:



The figure on the left represents the traditional way a graph is drawn, whereas the middle one shows how a collection of four sets represents the same graph and the figure on the right represents the graph drawn as a collection of overlapping intervals. To put this in the context of my assignment problem, an active interference graph is an arbitrary graph. Moreover, any intersection graph corresponds to a sequence of loads and stores from which an active interference graph can be built:

> The leftmost interval on the real line in each vertex S_i of an intersection graph corresponds to a load, while the rest of the intervals in S_i correspond to stores that are dependent on S_i's load. The order in which these loads and stores are executed is the same as the order in which the intervals from all the vertices appear from left to right on the real line.

Hence, coloring an active interference graph is NP-hard.

In addition to revealing the structure of my interference graphs, interval and intersection graphs also serve as convenient visual aids for developing an assignment algorithm and debugging its implementation. Although extremely small graphs can be drawn in the traditional fashion, this becomes rather messy when the number of vertices or edges reaches even a modest quantity, such as seven, and many of my interference graphs have substantially more vertices and edges than this; these graphs can be easily drawn as either interval or intersection graphs on a single page. For example, Figure 6.5 shows the live and active interference graphs for one of the loops in the CRI workload that has 47 values.

In this subsection, I emphasized the usage of intervals on the real line to represent my interference graphs. On the other hand, a computer uses yet another representation of a graph that is described in the previous subsection. All of these different representations portray the same graph with varying amounts of information captured in the description. In addition to revealing the structure of these graphs, the conceptual representations described above enable a human to better understand this assignment problem. Furthermore, I use the special structure of the interval graph to prove (in the next section) that a polynomial-time algorithm uses the minimum number of registers possible to assign values to registers.

6.2.3 Assigning Values to Buses and Registers.

In this subsection, I describe the algorithm I developed to use a partitioned register file. The ideal goal of this algorithm is to use the minimum number of buses and registers

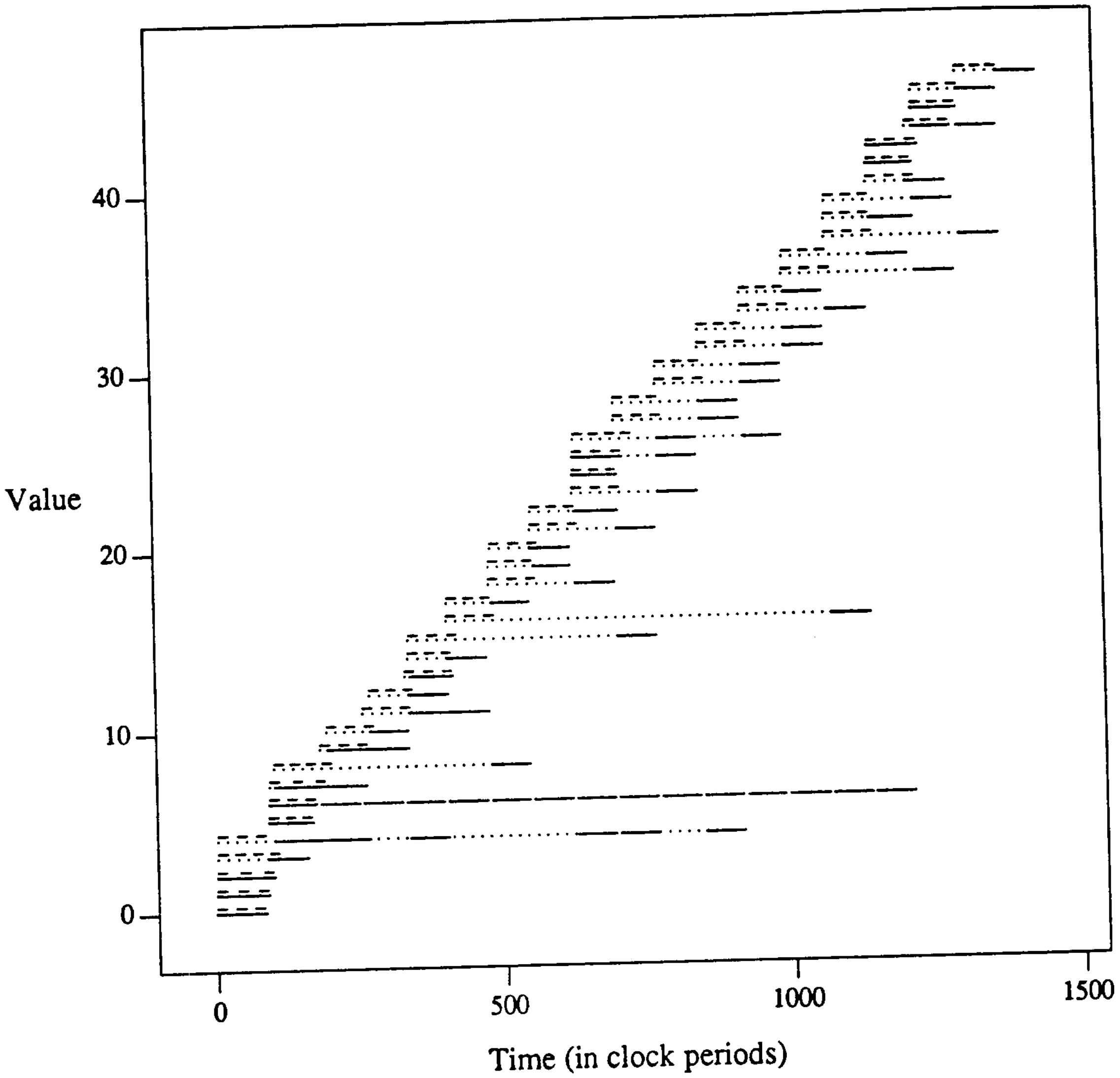


Figure 6.5: A Large Example of Live and Active Interference Graphs

This figure demonstrates large live and active interference graphs that are constructed from a loop in the CRI workload. Each graph contains 47 values. A value in the live interference graph is represented by a dotted line, whereas a value in the active interference graph is represented by a collection of short intervals that are horizontally aligned. A dashed interval indicates when a value is written and a solid interval indicates when it is read. As an indication of the complexity of these graphs, the seventh value has at least 33 edges in the active interference graph to represent potential conflicts in read accesses.

without causing any access conflicts, such as WAR or WAW register dependences, which add extra chimes to execution time. However, because I demonstrated that coloring an active interference graph is NP-hard, finding the minimum number of buses and registers for my assignment problem is also NP-hard.

Nevertheless, as with the scheduling problem described in the previous chapter, algorithms based on heuristics can minimize bus and register usage, although there is no guarantee that the minimum has been achieved. Although I would ideally like to minimize both the number of buses and the number of registers, this is not always possible due to conditions that oppose minimizing each quantity. Moreover, because increasing the number of buses requires more hardware than increasing the number of registers, higher priority is given to minimizing the number of buses.

My assignment algorithm actually consists of two algorithms, executed in sequence. Because minimizing bus usage has higher priority, the first algorithm assigns values to buses and then, for each assigned bus, the second algorithm assigns values to registers that share a bus. To bound their execution time, these algorithms do not backtrack nor do they re-assign buses or registers in an attempt to use fewer resources. The organization of assignment algorithms is similar to that of scheduling algorithms in that values are processed in a particular *order*, and a bus or register is chosen based on some *strategy*. What the order and strategy are depends largely upon what the goal of an algorithm is.

Figure 6.6 gives the algorithm that examines an active interference graph to assign values to buses so that bus conflicts are avoided. Because the number of registers per bus is fixed in the hardware, my algorithm also inspects the associated live interference graph to ensure that no more than that number of registers would be needed per bus to avoid register conflicts. Its basic strategy is to use buses as often as possible and introduce a new bus only if all available ones have been used or if too many values that are simultaneously live have already been assigned to those buses.

Because the goal of this algorithm is to minimize the number of buses that are assigned, it would seem judicious to first process values that have more potential conflicts under the rationale that a value with more edges is more likely to be in a part of the graph that is highly connected. But because this rationale is not always justified, my algorithm instead assigns values in the order they are created, an order that corresponds to the execution order of the operations that produce the values. Although it is not possible to always use the minimum number of buses, assigning values in order of creation time guarantees that the minimum number of registers per bus will be used, which in turn increases the chances of using fewer buses. As I will explain in the following paragraphs, this guarantee is possible because of the special structure of a live interference graph.

Figure 6.7 gives the algorithm that examines a live interference graph to assign values to registers so that access conflicts among registers are avoided. Algorithms similar to this one are described in standard compiler texts [2, 51]. In a fashion similar to the algorithm for bus assignment, the basic strategy for assigning registers is to use registers as often as possible and to introduce a new register only if all currently available ones have been used. An important characteristic of this algorithm is that it will *always* assign the *minimum* number of registers for *any* live interference graph and do so in a time that is polynomially proportional to the number of values in the graph. In contrast, other

Given an execution order of a dependence graph, assign a bus to each value in order of *creation time* by choosing the first bus b such that:

1. no neighbor in the active interference graph is assigned to bus b, and
2. no more than $R/B - 1$ neighbors in the live interference graph are assigned to bus b.

If no such bus exists, choose a new bus that has not yet been assigned.

Figure 6.6: An Algorithm for Assigning Values to Buses in a Partitioned Register File

This algorithm assigns buses to values that are produced and used in a particular order for a dependence graph. The *creation time* of a value is when a value is produced by its operation and, hence, corresponds to when that operation begins executing. The goals of this algorithm are to avoid conflicts due to register and bus accesses where multiple registers share a bus, and, in doing so, to minimize the number of buses and registers used.

Information in the interference graphs is used to avoid access conflicts. Bus conflicts are guaranteed not to occur by choosing a bus not already assigned to a value that is used in the same way at the same time. When values are simultaneously active in the same way is recorded in the active interference graph. Access conflicts among registers are guaranteed not to occur by choosing a bus not already assigned to R/B values that are simultaneously live with the value being processed, where R is the number of registers in hardware and B is the number of buses. Such information is recorded in the live interference graph.

Minimizing the number of buses and registers is accomplished in separate ways. The number of buses used is minimized by choosing a new bus only when all previously assigned ones will cause a bus conflict if chosen. The number of registers used is minimized by processing values in the order in which their associated operations are executed. Assigning values in this order causes the number of values simultaneously live with the value being processed to be a true reflection of the minimum number of registers needed at that point. As explained in the accompanying main text, this true reflection arises as the result of the special structure of the live interference graph. However, unlike the algorithm in Figure 6.7, more than the minimum number of registers may be used because higher priority is given to minimizing the number of buses, which tends to increase the number of registers used.

Given an execution order of a dependence graph, assign a register to each value in order of *creation time* by choosing a register r such that:

1. register r is among those that have already been assigned to at least one of the previously processed values, and
2. no neighbor in the live interference graph is assigned to register r.

If no such register exists, choose a new register that has not yet been assigned.

Figure 6.7: An Algorithm for Assigning the Minimum Number of Registers

This algorithm assigns registers to values that are produced and used in a particular order for a dependence graph. The goals of this algorithm are to avoid conflicts due to register accesses and, in doing so, to use the minimum number of registers.

Register conflicts are guaranteed not to occur by choosing a register that is not already assigned to a simultaneously live value.

The number of registers used is minimized by choosing a new register only when all previously assigned ones will cause conflicts if chosen. In addition, because values are assigned in the order of their creation times, the *minimum* number of registers is *always* assigned, a fact that is proven in the accompanying text.

polynomial-time algorithms do not always use the minimum number of colors. Such an algorithm results from a slight modification of the algorithm in Figure 6.7, where registers are assigned in order of the number of potential conflicts a value has instead of by its creation time. To illustrate that one algorithm is optimal and the other is not, Figure 6.8 shows the assignments produced by these algorithms for the same live interference graph.

The following two paragraphs prove why the modified algorithm is not an optimal one and the algorithm in Figure 6.7 is. These proofs are based on the following condition for optimality:

> If a value, which is about to be processed, is live at the same time as some other values that have already been processed, then these other values must also be simultaneously live with each other.

If this condition is always satisfied by an algorithm, then the number of registers used by such an algorithm equals the maximum number of simultaneously live values, which is the *minimum* number of registers needed for an execution order to avoid any register conflicts (as I explained in Section 5.1.1 of Chapter 5, *Register Usage and Instruction Scheduling*). This condition, however, is sufficient but not necessary for an algorithm to be optimal. For example, Ford and Fulkerson describe a different algorithm that does not satisfy this condition but is nonetheless optimal [43, pages 64–67].

The modified algorithm does not always satisfy the optimality condition because it is possible to process a value *after* assigning registers to values that are simultaneously

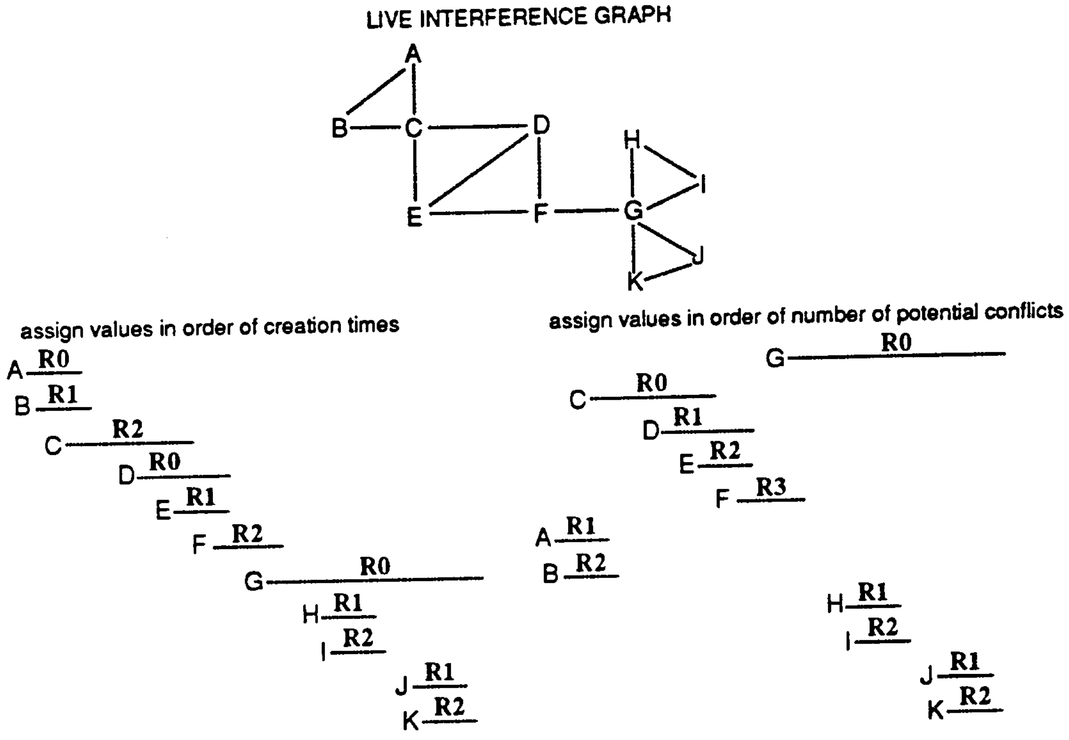


Figure 6.8: One Example of Two Algorithms that Assign Values to Registers

This figure demonstrates how two slightly different algorithms that assign values to registers use different numbers of registers for the same live interference graph. For this example, values are labeled A through K. The assignment on the left is produced by the algorithm listed in Figure 6.7, whereas the assignment on the right is produced by the same algorithm but with a minor modification: the order in which values are assigned is based on the number of potential conflicts a value has rather than its creation time. The number of potential conflicts a value has is equal to the number of edges incident upon it.

Not only does the assignment on the left use one fewer register, but it also uses the *minimum* number possible; using fewer than three registers for this live interference graph would result in register conflicts. Moreover, the algorithm that produces the assignment on the left will *always* use the minimum number of registers for *any* live interference graph, whereas the slightly modified algorithm will not.

live with it but not with each other. Such a situation is illustrated for the value F in the assignment on the right in Figure 6.8. For this example, value F is live at the same time as the values D, E, and G but these other values are not all simultaneously live with each other. Nonetheless, by the time value F is processed, its three neighbors have been assigned to different registers as follows:

> Because value G and another value, C, have the most number of potential conflicts but are not simultaneously live, these two values are assigned to register R0 first. Although value C could have been assigned a register different than R0, recall that a goal of these algorithms is to minimize the number of registers. When value C is assigned its register, there is not enough information to know that a different register could have been used without increasing the overall number of registers. Without this information, which becomes available after processing more values, and given the algorithm's goal, value C is assigned the register R0.
>
> The next values to be assigned registers are values D, E, and F, which all have the same number of potential conflicts. I have arbitrarily chosen to process values D and E first, which are assigned to two new registers, R1 and R2, because these two values are simultaneously live with each other and with value C.

Consequently, because different registers are first assigned to all three values that are live at the same time as value F, value F is assigned to a fourth register even though a different assignment can use only three registers. Although I chose an arbitrary order in which to process values D, E, and F, any assignment order for these three values results in a less than optimal assignment for the same reason: all neighbors of the last of these three values to be assigned are not all simultaneously live with each other but, nonetheless, have been assigned to different registers.

In contrast, the algorithm in Figure 6.7 always satisfies the optimality condition. For example, when the value F in Figure 6.8 is assigned to a register using this algorithm, only two of its neighbors in the graph, values D and E, have already been assigned to registers. More importantly, these two values are not only simultaneously live with value F but also with each other. In general, the optimality condition is satisfied for *any* live interference graph because such a graph is an interval graph. Moreover, because values are assigned to registers in the order of their creation times, if a value has a lifetime that overlaps those of other values already assigned to registers, then the lifetimes of these values must overlap as well. Thus, the optimality condition is always satisfied when processing a value, and this algorithm *always* uses the *minimum* number of registers needed to avoid any register conflicts.

The proofs described above highlight the importance of the relative positions of the *endpoints* of lifetimes, and algorithms that do not use this information tend to be less than optimal. For example, the algorithm based on the number of potential conflicts is not optimal. Although a potential conflict provides information about the relative position of two lifetimes, it says nothing about the relative positions of the endpoints of these lifetimes. Another example of an algorithm that does not use this information is one in which the length of a value's lifetime is the basis for the order of assignment [76]. Such an algorithm will also produce a less than optimal assignment for the live interference graph illustrated in Figure 6.8.

Conversely, several algorithms that make use of the relative positions of the endpoints of lifetimes are optimal, such as the algorithm based on the creation times of values; another is the one described by Ford and Fulkerson. Other examples that make use of the relative position of the endpoints are variations on the assignment order used for the algorithm in Figure 6.7. Other sequences that are possible are:

1. in the reverse order of when values are first produced,

2. in the order of when values are last read, or

3. in the reverse order of when values are last read.

Although different orders are used, the altered algorithms still satisfy the optimality condition and, hence, are optimal.

In summary, assigning values in the order of their creation time always produces an optimal register assignment. Although this does not always produce an optimal bus assignment, assigning values in this order guarantees that, when a value is processed, the number of its neighbors in the live interference graph assigned to a particular bus is also the minimum number of registers needed at that point. This, in turn, increases the chances of assigning fewer buses. Using creation time instead of number of potential conflicts or lengths of lifetimes has implementation benefits as well because values do not have to be sorted into an order different from that of the given execution order. Moreover, because values with more potential conflicts tend to have longer lifetimes and, hence, are executed earlier, an algorithm based on creation time still has some of the benefits of one based on a count of potential conflicts. In a later section, I will present some quantitative data showing that an algorithm based on creation time uses slightly fewer buses than one based on a count of potential conflicts.

I have intentionally given high-level descriptions of my algorithms so that a programmer has as much freedom in implementing these algorithms without affecting their stated goals. In particular, the description of the algorithm for assigning registers is deliberately nonspecific about which register to choose when there are several candidates (see Figure 6.7). The choice of a register is left as an implementation detail because this information is not used when proving that the algorithm is optimal in its register usage. As a result, other reasons can be used to dictate the choice of a register without affecting the number of registers assigned. On the other hand, because little can be shown mathematically about bus usage, the algorithm for assigning buses does specify which bus to choose, namely the first bus that was ever assigned and that satisfies the conditions for avoiding access conflicts (see Figure 6.6). Because the data I produced for showing how many buses are used is based on this algorithm, altering the choice of bus may affect how many buses are used. If a different rule for choosing a bus is preferred for other reasons, further experimentation is needed to determine the impact of bus choice on bus usage.

6.3 Experimental Framework

In this section, I briefly describe the performance metric and the methodology I use to carry out the studies in this chapter. Other aspects of the experimental framework, such as the architectural platform, the performance tools, and the workload are described in Chapter 4, *Common Experimental Framework.*

6.3.1 Performance Metric

For this study I should ideally have compared the execution times of the CRI workload using register files with various degrees of partitioning. But because the Y-MP simulator does not model a partitioned register file, and because I did not have access to the source code for the simulator, I compare instead the number of conflict-free assignments my algorithm produces for the CRI workload using various configurations of vector register files. Unlike execution time, a count of conflict-free assignments cannot indicate how much worse an assignment with conflicts performs relative to one that is conflict-free. Because this performance metric is less informative than execution time, decisions based on counts of conflict-free assignments are more conservative than ones based on execution time.

Nonetheless, a count of conflict-free assignments is a suitable alternative for a performance metric. This is because an execution order for a dependence graph will be executed in the same amount of time using a partitioned register file or a traditional one as long as an assignment for that configuration is conflict-free. Furthermore, this metric can be collected at compilation time unlike at execution time, which requires executing the generated code on the Y-MP simulator. As a result, a count of conflict-free assignments provides a quick method for evaluating the effectiveness of the algorithm and any of its variants.

6.3.2 Methodology

To determine the minimum number of buses needed to effectively use the functional units of the Cray Y-MP vector processor, I use the heuristic algorithm presented in the previous section and compare the performance of its assignments for a different number of registers and buses. Such a comparison cannot definitively give the minimum number of buses needed because this assignment problem is NP-hard; finding the optimal answer for such a problem is practically infeasible for the same reasons that finding the optimal answer for the scheduling problem, another NP-hard problem, is practically infeasible (see Section 5.2.2 in Chapter 5, *Register Usage and Instruction Scheduling*). In additional, a heuristic algorithm has the advantage that it will increase compilation time minimally when it is used in a compiler.

6.4 How Many Buses?

In a previous section, I explained how to assign values to a partitioned register file by using an algorithm that attempts to minimize the number of buses and registers assigned. In this section, I present data to evaluate how well a partitioned register file can be used

		NUMBER OF REGISTERS					
		4	8	16	32	64	128
NUMBER OF BUSES	4	47%	54%	55%	55%	55%	55%
	8		85%	94%	97%	98%	99%
	16			96%	99%	100%	100%
	32				99%	100%	100%

Figure 6.9: Data for Evaluating the Usability of a Partitioned Vector Register File

This table lists the percentage of dependence graphs for which my algorithm produces an assignment with no conflicts for the indicated number of buses and registers. For example, for 94% of the dependence graphs in the CRI workload, my algorithm uses 8 buses or less, and 16 registers or less to produce a conflict-free assignment. Because one of the goals of this study is to determine whether the number of buses can remain at 8, which is the number of buses the Cray Y-MP currently implements, I have highlighted the data for 8 buses.

and to determine how many buses and registers provide an acceptable cost/performance configuration for the Cray Y-MP vector processor.

6.4.1 Performance Evaluation of Algorithm and Partitioned Register Files

Figure 6.9 tabulates the fraction of dependence graphs that have a conflict-free assignment using partitioned register files that have anywhere from 4 to 128 registers and from 4 to 32 buses. This data shows that the count of conflict-free assignments does not change significantly among the various configurations of vector register files with 16 or more registers. From this observation, I conclude that a partitioned register file performs comparably to a traditional vector register file with the same number of registers as long as enough registers are provided.

This data also shows that, with only 4 buses, there are many dependence graphs that do not have a conflict-free assignment, even when an abundance of registers is provided. Hence, 4 buses are clearly not enough to effectively use the functional units of the Cray Y-MP processor. Although a partitioned register file with 4 buses is attractive from a cost viewpoint, such a configuration is never an acceptable choice because of its consistently poor performance. Moreover, because the coverage is so poor, this data also suggests that, in general, a partitioned register file with 4 buses is an inadequate design for a vector processor with a configuration of functional units comparable to the Y-MP. In particular, the partitioned register files for the Ardent Titan and the Fujitsu VPU, which both use 4 buses, may be ineffective for their respective configuration of functional units, although further investigation is needed to verify this hypothesis.

On the other hand, doubling the number of buses from 4 to 8 results in a large increase in the number of conflict-free assignments, and most of the dependence graphs have a conflict-free assignment when 16 or more registers are provided. Moreover, because there is little improvement when the number of buses is doubled from 8 to 16, this data shows that 8 buses are enough to effectively use the functional units of the Cray Y-MP processor.

Although no assignment algorithm can guarantee to use the minimum number of buses, I stated in a previous section that assigning values in the order of their creation time instead of by the number of their potential conflicts will tend to use fewer buses. This is because assigning values by creation time guarantees that at least the minimum number of registers per bus will be used, which in turn increases the chances of using fewer buses. Quantitative evidence of this is given in Figure 6.10, which indicates the number of buses assigned by these two algorithms for a partitioned register file with 8 buses and 1, 2, or 4 registers sharing a bus. Regardless of the amount of partitioning, assigning on the basis of creation time uses fewer than 8 buses for a greater fraction of the dependence graphs in the CRI workload. Although this is not a crucial difference—both algorithms produce a comparable number of conflict-free assignments for 8 buses—assigning values by creation time is nonetheless the better choice not only because it uses slightly fewer buses, but also because it is easier to implement.

6.4.2 Making a Stronger Case for 8 Buses and 16 Registers

From the data in Figure 6.9, it is clear that 8 buses are enough, but the appropriate number of registers—16 or 32—is less obvious. Because these numbers do not reflect execution time, there is no way of knowing whether increasing the number of registers from 16 to 32 would result in a significant improvement in performance. A stronger case for 16 registers and 8 buses, however, could be made by increasing the number of conflict-free assignments for this configuration. To do this in a systematic fashion, in this subsection I examine in greater detail the seven dependence graphs for which my algorithm generates an assignment with conflicts for any clues that would result in more conflict-free assignments. Once I find a method that produces a conflict-free assignment for one of these dependence graphs, I use that method on the rest of the dependence graphs to see whether more conflict-free assignments are produced overall.

For two of the seven dependence graphs, my assignment algorithm uses more than 8 buses for different reasons. For one of these dependence graphs, more than 8 buses are assigned because my algorithm doesn't happen to produce the best assignment, emphasizing the heuristic nature of this algorithm. But assigning values to buses in the order of their final reads or *death times*, rather than their creation times, does produce a conflict-free assignment for 8 buses and 16 registers. Using death time instead of creation time as the basis for assignment for all the graphs in the CRI workload, however, does not change the overall number of conflict-free assignments when using 8 buses and 16 registers. This is because both these algorithms are based on heuristics and neither will always produce the best assignment; although assigning values by death times produces a better assignment for some dependence graphs, assigning values by creation times produces a better assignment for other dependence graphs.

More than 8 buses are assigned for the second dependence graph because, as I discovered upon closer examination, its associated active interference graph contains a complete graph with 9 vertices. Although *no* assignment algorithm can use fewer than

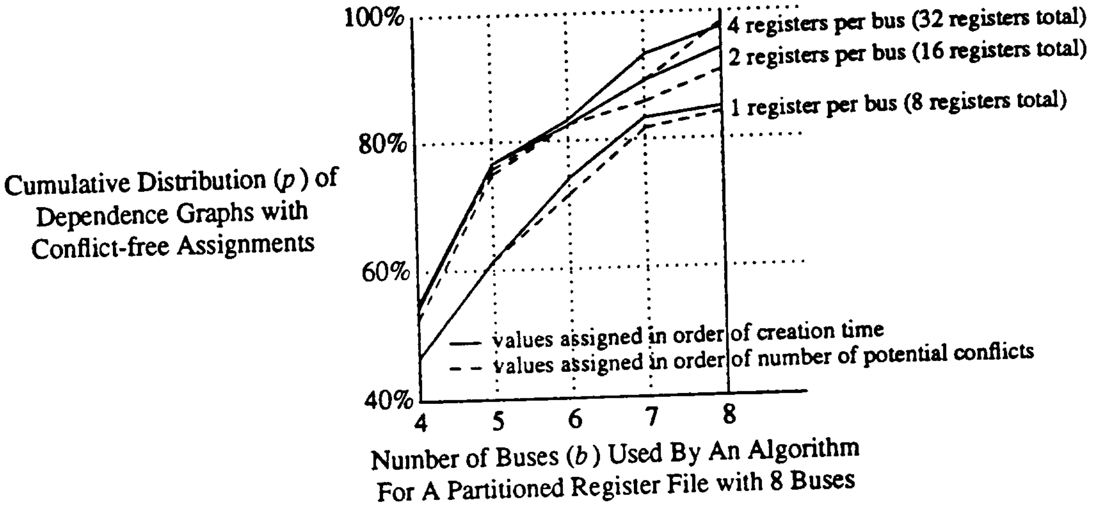


Figure 6.10: Comparing the Effectiveness of Two Assignment Algorithms

This figure compares the effectiveness of two slightly different assignment algorithms by summarizing the number of buses each uses for partitioned register files with 8 buses and 1, 2, or 4 registers sharing a bus.

A point (b,p) on a curve means that a particular algorithm uses b or fewer buses for p percent of the dependence graphs to produce a conflict-free assignment using a specific partitioned register file. Hence, each curve is the cumulative distribution of dependence graphs that have a conflict-free assignment using a particular algorithm for a specific partitioned register file. The solid curves indicate the number of buses used by the algorithm listed in Figure 6.6, and the dashed curves indicate the number of buses used by the same algorithm but with a slight modification: values are assigned based on the number of potential conflicts rather than their creation times. The bottom two curves are the cumulative distributions of the two algorithms when assigning 8 buses and 8 registers, the middle two curves are the distributions when assigning 8 buses and 16 registers, and the top pair of distributions are for 8 buses and 32 registers.

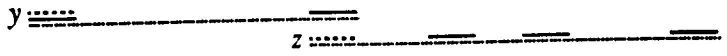
These distributions show that, regardless of the amount of partitioning, assigning on the basis of creation time uses fewer than 8 buses for the most number of dependence graphs in the CRI workload.

9 buses for this interference graph, interference graphs correspond to a particular execution order of a dependence graph, and a different execution order allows my algorithm to produce a conflict-free assignment for this dependence graph using 8 buses. This different execution order is generated by a slight modification to the scheduling algorithm, in which operations are processed in their statement order rather than in the order of their path distance. Although changing the scheduling algorithm may affect the execution time of the loops, this is only relevant if the change increases the overall number of conflict-free assignments when using 8 buses and 16 registers. As with the previous attempt, the heuristic nature of the scheduling and assignment algorithms prevents this method from always producing the best results. In fact, applying this change to all the dependence graphs reduces the overall number of conflict-free assignments by one, thereby suggesting poorer performance for the entire CRI workload.

Although my algorithm uses more than 8 buses for two dependence graphs, it generates assignments with conflicts for the other five dependence graphs because their minimal register requirement is greater than 16. For these five dependence graphs, a conflict-free assignment for 8 buses and 16 registers is possible only if their minimal register requirement is reduced. Because this requirement is associated with a particular execution order, one method of reducing it is to use a different scheduling algorithm to generate different execution orders for the same set of dependence graphs. Using the modified scheduling algorithm described in the previous paragraph thus reduces the minimum register requirement to less than 16 for two of the five dependence graphs. Unfortunately, as I mentioned in the previous paragraph, the total number of conflict-free assignments decreases by one. Using execution orders generated from other variations of the scheduling algorithm also produces comparable results. Hence this alternative is not an improvement.

Another approach to reducing the minimal register requirement is to allow register assignments of the form `Vi<-Vi op Vj` or `Vi<-Vj op Vi`, where two different values, an operand and a result, are assigned to the same register `Vi`. For the sake of brevity, from this point on I use `Vi<-Vi op Vj` to represent both forms of this type of assignment. Although the hardware allows such assignments, my modeling of assignment does not because all values associated with an operation are considered to be simultaneously live and hence are assigned to different registers. Allowing simultaneously live values to share a register could reduce the minimal register requirement and perhaps increase the number of conflict-free assignments when using 8 buses and 16 registers. Although including a feature that recognizes when two simultaneously live values can actually share a register complicates the algorithm somewhat, this additional complexity is justified nonetheless if substantially fewer registers and buses can be used as a result.

An assignment of the form `Vi<-Vi op Vj` should only be used when there is no possibility of causing a register or bus conflict. Because a value is produced by an operation, such as a load or multiply, no conflicts will occur between two simultaneously live values, y and z, only if their associated operations, op_y and op_z, satisfy the conditions listed in Figure 6.11. The relative positions of two such values in the live and active interference graphs look like this:



CONDITION	REASON FOR CONDITION
1. operation op$_z$ is dependent on operation op$_y$	minimum requirement for using Vi<-Vi op Vj
2. op$_z$ is the final read of the value y	to avoid register conflicts
3. op$_z$ is not executed in the same chime as op$_y$	to avoid bus conflicts due to writes
4. any operation reading value z does not execute in the same chime as op$_z$	to avoid bus conflicts due to reads

Figure 6.11: Conditions for Using Vi<-Vi op Vj

This table lists the conditions that identify two simultaneously live values, y and z, that can be used in assignments of the form Vi<-Vi op Vj or Vi<-Vj op Vi without causing any register or bus conflicts. Because a value is produced by an operation, such as a load or multiply, these conditions actually apply to the value's associated operations, op$_y$ and op$_z$.

The important aspects of this diagram are that the last read of the value y (indicated by a solid interval) occurs simultaneously with the write of the value z (indicated by a dashed interval), and all other accesses of either value occur at other times. Although this diagram illustrates the last three conditions in Figure 6.11, it does not indicate that operation op$_z$ is dependent on operation op$_y$, information that is kept in the associated dependence graph.

If the conditions above are satisfied, the value z can be assigned to the same register assigned to the value y without causing any register or bus conflicts, and the instructions for their associated operations look like this:

$$\mathtt{Vi} \quad \mathtt{<-} \quad \cdots \quad \mathtt{op}_y \quad \cdots$$
$$\vdots$$
$$\mathtt{Vi} \quad \mathtt{<-} \quad \mathtt{Vi} \quad \mathtt{op}_z \quad \cdots$$

Two values that satisfy the above conditions can be merged in the associated interference graphs and treated as a single value for the purposes of assignment. Thus, to incorporate assignments of the form Vi<-Vi op Vj into my algorithm, the interference graphs, after being built, are modified by combining values that satisfy the above conditions. The modified graphs are then used as before to assign buses and registers.

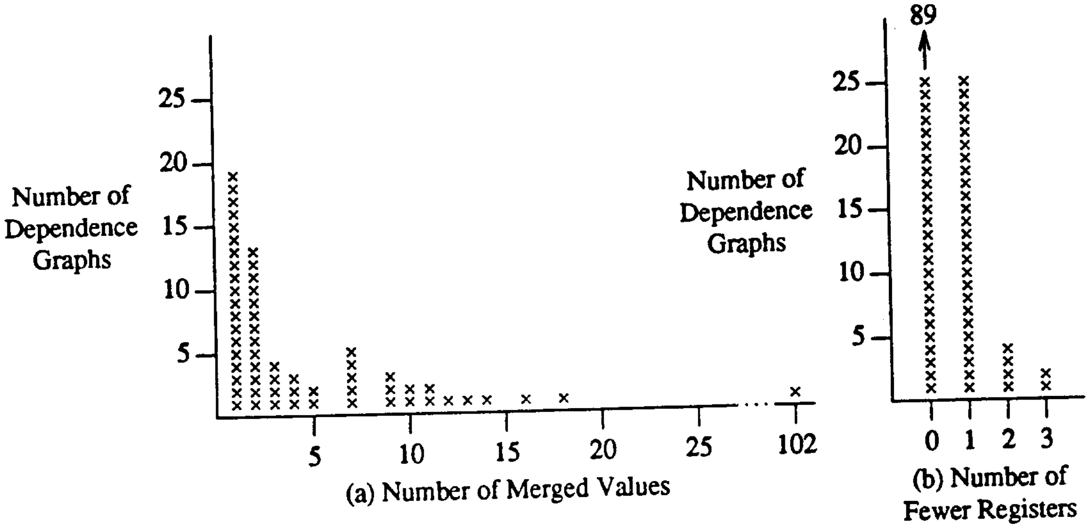
Although combining live values decreases the number that are simultaneously live in the interference graph, this merging also has the negative effect of increasing the connectivity in the active interference graph. The data in Figure 6.12 shows that the reduction in the minimal register requirement is insufficient grounds for countering the negative impact of this increased connectivity. The first set of data in graph (a) shows that such an assignment can be used fairly often, especially for the larger dependence graphs. For example, from 10 to 20 values can be merged for nine dependence graphs and 102 values for one dependence graph. Although this first set of data looks promising, the second set in graph (b) shows that the resultant reduction in the number of registers is not substantial. At best, the minimal register requirement is reduced by 3 for two dependence graphs; most others show no change or at most a reduction of 1 register.

Finally, the third set of data in table (c) shows that using `Vi<-Vi op Vj` actually *decreases* the number of conflict-free assignments when 8 buses and 16 or more registers are uses as well as and when 4 buses and 8 or more registers are used. This is because merged values result in longer lifetimes which in turn reduce the number of registers that become available for re-assignment at any given time. Furthermore, allowing the re-use of a register in this fashion is never better than always using three distinct registers for the values that are read and written by an operation. Although there are other reasons for using `Vi<-Vi op Vj` (for example, in the vector version of a scalar reduction where the operand and result actually represent the same value), I conclude from this set of data that using such an assignment does not improve the register nor bus usage, and hence there is little reason to incorporate such a capability into the assignment algorithm.

A third approach to reducing the minimal register requirement is to spill registers. In other words, rather than combining values in the interference graphs as was done in the preceding method, a value is split into two when registers are spilled to reduce minimal register requirement. However, a judicious choice of which values to spill and when to do so is necessary if this technique is to work. Although the minimal register requirement can be reduced by choosing candidates from the largest set of values that are live at the same time, the actual act of spilling a value requires using a bus, and hence adds more edges to the active interference graph. In order to produce a conflict-free assignment, the reduction in register conflicts must be more than the potential increase in bus conflicts.

Data from the previous chapter suggests that spilling registers is a promising approach. Even though five of the dependence graphs have a minimal register requirement greater than 16, I showed in the previous chapter that with a traditional vector register file 16 are enough to improve performance and that adding more registers results in only a nominal improvement in performance. This is because the algorithm for register assignment in the *cft77* compiler generates code for register spilling to accommodate the requirements of the larger dependence graphs. Moreover, because a vector architecture supports fine-grain parallelism, these extra instructions can execute in parallel with the original instructions without increasing execution time.

Incorporating register spills into my algorithm involves changing the dependence graph to show which values are to be spilled and when the extra spill operations are to be executed. However, this entails extensive changes because more information must be integrated than what was done for the other changes I have described. For example, using a different order for scheduling or assigning is an easy change because it does not require examining the interaction among sets of values. A more complicated change is allowing assignments of the form `Vi<-Vi op Vj`, which requires examining the interaction between pairs of dependent values before the interference graphs can be suitably modified. Introducing register spills is even more complex because it requires examining the interaction



		NUMBER OF REGISTERS					
		4	8	16	32	64	128
	4	47%	*53%*	*53%*	*53%*	*53%*	*53%*
NUMBER	8		85%	*92%*	*93%*	*94%*	*94%*
OF BUSES	16			96%	99%	100%	100%
	32				99%	100%	100%

(c) Fraction of Dependence Graphs with Conflict-free Assignments

Figure 6.12: Impact on Assignments When Using `Vi<-Vi op Vj`

The figures above show three sets of data for evaluating the effectiveness of allowing assignments of the form `Vi<-Vi op Vj`, whereby simultaneously live values that have specific characteristics can be assigned to the same register without causing any conflicts.

Because two such values are treated as a single value for the purposes of assignment, the first histogram, graph (a), indicates how often such an assignment can be used in a dependence graph. Allowing simultaneously live values to share a register could reduce the minimal register requirement, and the second histogram, graph (b), shows the size of this reduction for each dependence graph. Finally, the third set of data, table (c), gives the fraction of dependence graphs for which my algorithm produces a conflict-free assignment when using `Vi<-Vi op Vj` for the indicated number of buses and registers.

The first set of data shows that assignments of the form `Vi<-Vi op Vj` can be used fairly often. Despite this promising result, the second set of data shows that the minimal register requirement for a dependence graph is not reduced substantially. Finally, the third set of data shows that using `Vi<-Vi op Vj` does no better than not using this type of assignment, and actually *decreases* the number of conflict-free assignments when more than one register shares a bus in a partitioned register file with 4 or 8 buses (indicated by the percentages in italics).

among *several* values that are live or active at the same time in order to choose candidates for spilling and to modify the dependence graph appropriately. Because of such extensive changes and because, as I argue in the next subsection, using 32 registers with 8 buses is a reasonable choice for now, I omit a quantitative evaluation of this method, leaving such an undertaking for the future.

6.4.3 Choosing a Partitioned Register File

In summary, the experiments in the previous subsection show that increasing the number of conflict-free assignments for 8 buses and 16 registers in a systematic fashion is a difficult task. Although the data in Figure 6.9 is unable to provide a strong case for either 16 registers or 32 registers when using 8 buses, the relative priorities of performance and cost also influence the choice of which configuration to implement.

If cost is more important than performance, then this data suggests that a configuration of 8 buses and 16 registers is the better choice. Because different heuristics produce a conflict-free assignment for different dependence graphs, performance could be improved by sequentially applying these techniques until a conflict-free assignment is found. The drawback to this approach is a potential increase in compilation time. However, experimental evidence shows that over 90% of the cases would need to use only the original algorithm, and doing so should not significantly increase the average compilation time. Another alternative is to include register spilling into the assignment algorithm, although further investigation is needed to evaluate the effectiveness of this technique.

On the other hand, if performance is more important than cost, then the better choice is a configuration of 8 buses and 32 registers. Based on the cost/performance analysis in Figure 6.2 (on page 115), however, this configuration is not an ideal tradeoff between cost and performance. In fact, a partitioned register file with 8 buses and 32 registers is slightly more costly to implement than a traditional one with 16 buses and 16 registers. Nonetheless, three factors favor the partitioned organization.

First, using 32 registers results in a greater improvement in performance than when using 16 registers (18% versus 9%). Moreover, a partitioned register file with 8 buses and 32 registers has slightly more conflict-free assignments than a traditional register file with 16 buses and 16 registers.

Second, a register file that provides more registers rather than more buses is better able to accommodate a wider range of dependence graphs. Because the number of functional units provided by hardware limits the number of values that can be simultaneously active, minimal bus requirements are influenced more by the configuration of functional units and less by a program's characteristics. In other words, a dependence graph with even an inordinate number of operations is unlikely to require more than 8 buses because the number of simultaneously active values is limited by the number of vectorizable operations that can execute in parallel, which in turn is limited by the number of functional units. Based on the data in Figure 6.9, 8 buses appear to be enough for the current configuration of functional units in the Cray Y-MP.

In contrast to bus requirements, the need for a register depends on how operations interact with each other. This means that minimal register requirements are influenced less by what hardware provides and more by the dependence patterns in a program. Providing

		VECTOR REGISTER FILE			
		8R	16R	16R	32R
		8B	8B	16B	8B
NUMBER OF	64	0%	10%	25%	30%
ELEMENTS PER	32	-5%	0%	15%	10%
VECTOR REGISTER	16	-7%	-5%	10%	0%

Figure 6.13: Cost Analysis of Vector Register Files with Varying Vector Lengths

This table gives the relative difference in chip count of some traditional and partitioned vector register files when varying the number of elements per vector register, which is also known as the *vector length*. This cost analysis is an extension of the one given in Figure 6.1, which assumes a constant vector length of 64. No increase in chip count indicates a configuration where the total number of registers and buses is the same as the current implementation of a Cray Y-MP processor, which uses a total of 512 registers and 8 buses.

32 registers instead of 16, but at the same hardware cost, allows a partitioned vector register file to more easily accommodate larger dependence graphs, which, although unlikely to require more than 8 buses, are likely to require more than 16 registers.

The third factor that favors the partitioned organization is that the cost of implementing 8 buses and 32 registers can be lessened by reducing the vector length. Whereas the cost/performance analysis in Figure 6.2 assumes that the vector length remains constant at 64, Figure 6.13 shows how the increase in cost is affected by the vector length. When the vector length is 32, a partitioned register file with 8 buses and 32 registers is, in fact, *less costly* to implement than a traditional one with 16 buses and 16 registers. A further reduction in vector length to 16 makes the partitioned organization even more attractive. However, as explained at the beginning of this chapter, sustainable performance may be adversely affected if the vector length is shortened too much. As a result, although this cost analysis looks promising, further studies are needed to measure how shorter vector lengths affect performance.

6.5 Related Work

I presented one algorithm for assigning registers and one for assigning buses, whereas most researchers emphasize only register assignment. Two groups of researchers, Eisenbeis, Jalby, and Lichnewsky as well as Mangione-Smith, Abraham, and Davidson, have both presented assignment algorithms for vector registers [33, 81]. Although these two algorithms are optimal in that they always assign the fewest number of vector registers for a given execution order, a polycyclic vector scheduler produces the execution order. In contrast, my assignment algorithm, which is also optimal, uses an execution order produced by a simple vector scheduler. Both the assignment and scheduling algorithms I use are less complex than those developed for these other studies.

Although my algorithm is used for a vector architecture in this chapter, it can also be used for other architectures. The underlying architecture affects when values are live and active, information that is used to construct the interference graphs. Once constructed, however, the assignment of values to registers can be done using the algorithm I described. In general, assignment algorithms for one type of architecture can often be adapted to other architectures. In contrast, algorithms for different types of program fragments cannot be transformed as easily and as a result, algorithms presented in the literature can be categorized by the type of program fragment they operate on [96]. In the following paragraph, I present this categorization of assignment algorithms to contrast my algorithm with others.

One category of algorithms operates on a single expression at a time, where the expression is represented by a tree, which is a dependence graph with no common subexpressions [87, 98, 99, 3]. Another class of algorithms, which are known as *global register assigners*, operates on the entire program, which is represented by a control flow graph whose vertices are basic blocks [9, 18]. The third category of algorithms, which are known as *local register assigners*, operates on a single basic block at a time. This type of program fragment, which can be represented by a dependence graph with common subexpressions, falls between the types for the other two classes of algorithms. Algorithms for local register assignment can be further classified into two sub-categories based on whether the execution order is fixed [100] or not. The algorithm I presented, as well as the two assignment algorithms for polycyclic vector schedules mentioned above, fall into the second sub-category. In this dissertation, for the sake of brevity, I have used the terms *local register assignment* and *register assignment* to refer to this sub-category rather than the whole category.

Because part of my investigation was to determine the minimum numbers of registers to implement in hardware, I designed my algorithm to use the fewest registers for a given execution order. For practical register assignment, however, the number of registers in hardware is fixed and when the minimal register requirement of an execution order exceeds that number, extra memory references must be generated to spill registers. Hence, in addition to assigning values to registers, an algorithm for a production compiler must also choose an appropriate register to spill so as to minimize the number of extra memory references. Not surprisingly, most algorithms for local register assignment handle the problem of register spilling. Although my algorithm for assigning registers does not directly address this problem, it can be easily extended to do so. I discuss the merits of doing this when I describe future studies in Section 7.2 of Chapter 7, *Concluding Remarks*.

As part of my presentation, I gave a proof that my algorithm always assigns the minimum number of registers. Researchers rarely examine this aspect in much detail, focusing instead on minimizing the number of extra memory references. An exception is Freiburghouse, who describes an optimal algorithm that differs from mine in two ways [44]. First, rather than constructing a live interference graph, Freiburghouse computes *usage counts*, which are the number of times each value is referenced. The second and more important difference is how the optimality of the algorithm is proven. Although Freiburghouse did not give a proof, he refers to Gries who independently developed this same algorithm to allocate temporary variables to memory locations on the stack rather than to registers [51, pages 299–304]. The optimality of this algorithm, which is actually attributed to Dantzig and Reynolds [27], relies on a stack to keep track of available registers and does not allow any choice for a register.

In contrast, the proof showing the optimality of my algorithm emphasizes the importance of assigning values in order of their creation times and does not rely on which register to choose when there are several candidates. Because of the similarities between my algorithm and Freiburghouse's, this proof can also be used to prove the optimality of his algorithm, hence removing the necessity for a stack to keep track of available registers. A consequence of this proof is that a register can be chosen for other reasons. For example, this extra degree of freedom allows available registers to be assigned using a first-in, first-out queue rather than a last-in, first-out stack without affecting the optimality of either algorithm. Assigning registers in such a round-robin fashion increases the probability that the same register will be assigned to values whose lifetimes are temporally far apart. This in turn makes the register assignment less sensitive in terms of performance to mismatches between what the compiler thinks the hardware does and how the hardware actually behaves.

As part of my algorithm, I modeled the assignment of registers as a graph coloring problem. This technique has been applied to global register assignment by Ershov *et al.* [35, 36], who uses the term *incompatibility graph* instead of *interference graph*, and by Chaitin *et al.* [18], who coined the term *interference graph*. Assigning registers globally and locally are two different problems. The interference graph for global register assignment can be described as an intersection graph [78], which is an arbitrary graph, whereas the one for local register assignment is an interval graph, which has a special structure. Although other researchers have also recognized the fact that an interval graph arises when assigning registers locally [11], I am unable to find a reference that specifically describes the use of graph coloring for local register assignment.

Unlike the other applications of graph coloring for register assignment, I use not one but two interference graphs at the same time. These two graphs arose from the special organization of the partitioned vector register file. The live interference graph I use is similar to the interference graph constructed for global register assignment in that liveness is the basis for interference. However, the *range* of a live value for global register assignment is slightly different because a value's life can span multiple basic blocks, thus producing an intersection graph rather than an interval graph. The active interference graph I use is a new construction.

The inspiration for the development of this chapter's assignment algorithm is, of course, the partitioned vector register file. In addition to reducing the cost of implementing more vector registers, this configuration is also an inexpensive method for implementing a reconfigurable vector register file, as is the case in the Ardent Titan and the single-chip vector processor by Fujitsu [31, 64]. Although the Fujitsu's FACOM vector processors also provide a reconfigurable register file, descriptions of its implementation do not indicate whether or not it is a partitioned vector register file [85]. Nonetheless, despite the existence of these commercial implementation, I do not know of any publication that describes the details of an algorithm that can effectively assign values to a partitioned register file.

In addition to developing the assignment algorithm, I also examined the performance impact of partitioned register files and concluded that such configurations are not necessarily an impediment to performance. There are relatively few studies that examine the appropriate balance between number of registers and number of buses. The most related ones are those that investigated the performance impact of different organizations of register files in VLIW or superpipelined scalar architectures [105, 15, 37]. These studies compared the performances of monolithic and distributed register files, whereas I compared the performances of various configurations of a partitioned register file, an organization that falls between the other two with respect to accessibility and hardware cost.[5] A conclusion that could be drawn from these studies is that, in order to get the best performance, the number of result buses must be equal to the number of register sets in a distributed register file. However, the amount of parallelism supported by the architectures in these studies is small in comparison to the amount supported by the vector architecture I used. Once enough parallelism is supported by the *entire* architecture of a processor, I showed that the number of buses can be less than the number of register banks without adversely affecting performance.

6.6 Summary

In this chapter, I examined in greater detail the tradeoff between improved performance and increased cost when implementing more vector registers. In a multi-chip implementation, such as the Cray Y-MP processor, the number of chips is a good measure of hardware cost. Using this metric, I showed that doubling the number of registers from 8 to 16 results in a 25% increase in the number of chips when implemented in a straightforward fashion, while providing only a 9% improvement in performance. The reason for this high cost is that the implementation of a vector register file actually consists of register banks, which store data, and interconnections, which link register banks to functional units. Doubling the number of registers in a straightforward fashion requires doubling both these types of components.

Because the size of an interconnection is determined by the number of buses attached to them, an obvious hardware solution that improves the tradeoff between increased cost and performance gain is to have more than one register share a bus. This new configuration, which I call a *partitioned vector register file*, is another example of partitioning a register file to reduce cost at the expense of increased restrictions on accessing registers. Just as a traditional vector register file, which is also partitioned, is less costly to implement than a monolithic one with the same number of ports, a partitioned vector register file is less costly to implement than a traditional one with the same number of vector registers.

Although the restricted access to registers is not a problem in a vector register file, the restricted access to vector registers in a partitioned vector register file does present a challenge. Figure 6.14 presents an overview of the algorithm I developed that circumvents the restricted access of this new configuration. My algorithm has two goals. One goal is to avoid access conflicts, such as WAR or WAW register dependences, which would degrade performance. This is accomplished by always assigning values that are active at the same time to different buses and assigning values that are live at the same time to different

[5] Section 2.2.2 (pages 11 to 16 in Chapter 2, *Fundamentals of Vector Architectures*) explains the organizational differences between monolithic, partitioned, and distributed registers files.

registers. Another goal is to assign as few buses and registers as possible to help determine the minimum number of buses and registers needed in a partitioned register file that is cost-effective. This goal is accomplished not only by re-assigning buses and registers whenever possible but also by assigning values in the order in which they are produced. Although it would seem more judicious to process values based on their number of potential conflicts, I showed that assigning values in order of creation time has some advantages.

Finally, I presented data to demonstrate the effectiveness of the algorithm I developed and to choose a partitioned register file that is cost-effective. Because the Y-MP simulator does not model a partitioned register file, my performance metric is the number of conflict-free assignments produced by my algorithm rather than execution time. Although the data clearly shows that 8 buses are enough, the appropriate number of registers—16 or 32—is less obvious. As a result, the final choice is also influenced by the relative priorities of performance and cost. If cost is more important than performance, then choosing 16 registers is more appropriate and methods for improving performance, such as combining different heuristics in a cascading fashion or using register spilling, should be investigated further. On the other other hand, if performance has higher priority over cost, then 32 registers is the better choice. Cost can be reduced by shortening the vector length from 64 to 32, although further studies are needed to measure the resultant impact on performance.

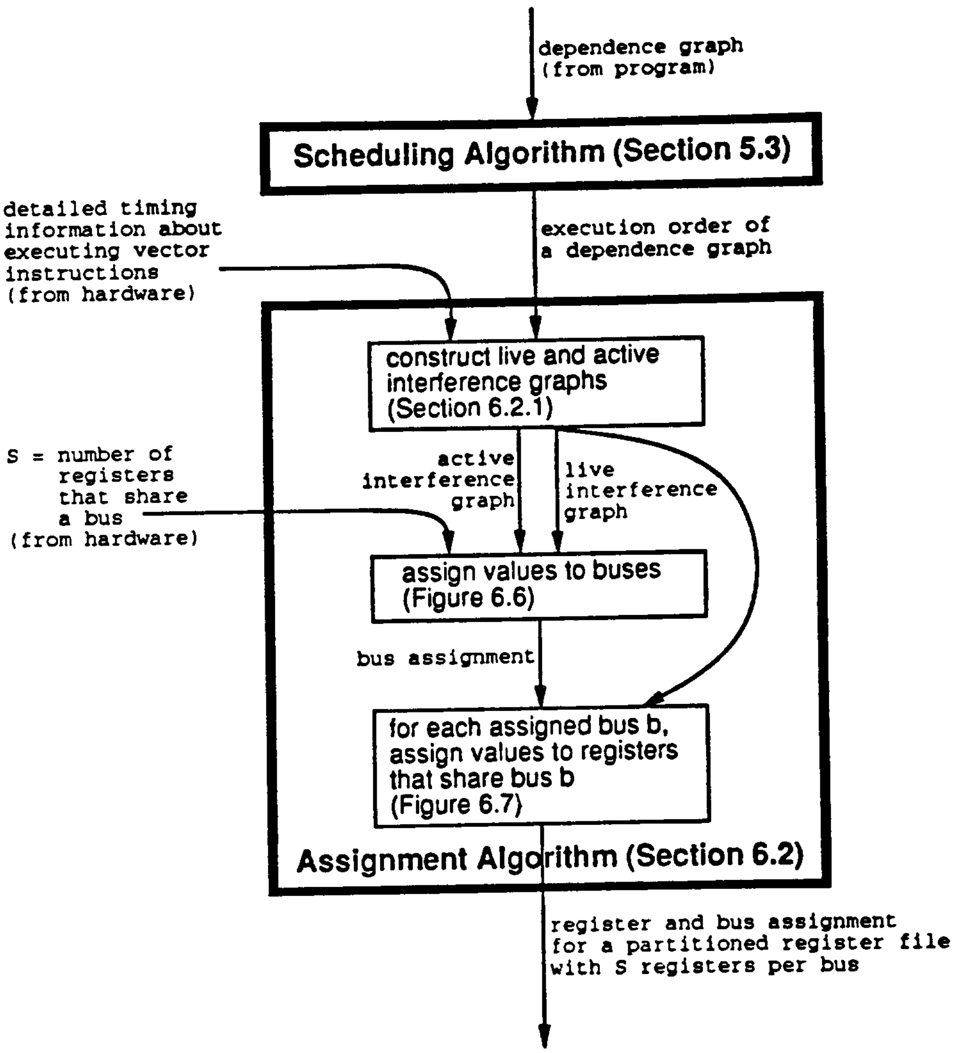


Figure 6.14: Algorithm for Assigning Values to a Partitioned Register File

This figure is an overview of the assignment algorithm I developed for using a partitioned register file where more than one register shares a bus. Details of the individual components are given in the indicated sections and figures. Figure 4.2 (on page 67 in Chapter 4, *Common Experimental Framework*) shows where the functions in the above diagram are performed in the *cft77* compiler.

Chapter 7

Concluding Remarks

In this chapter, I summarize my work by highlighting the contributions in this dissertation. I conclude with a discussion of studies for future work.

7.1 Contributions of Dissertation

Each of the four major chapters in this dissertation contains contributions to the areas of processor design and code optimization. These contributions fall into one of three categories: improvements to previous work, syntheses of published material, and extensions to the state of the art.

7.1.1 Improvements to Previous Work

Chapters 2, 5, and 6 contain improvements to previous work. The first of these improvements is in Section 2.2 of Chapter 2, *Fundamentals of Vector Architectures* (pages 6 to 17), where I used a common framework to compare how different architectural classes support fine-grain parallelism. Jouppi and Wall have done a similar comparison for these architectural classes but focussed mainly on how multiple operations are initiated [70]. My comparison adds to theirs by including techniques for the simultaneous delivery of operands and results. Another improvement is the classification of implementations for a multiported register file in Section 2.2.2 (pages 11 to 16). Other researchers in processor design have classified register files into shared and split ones, which I call monolithic and distributed, respectively [105, 15]. I expanded this classification to include a partitioned register file, of which a vector register file is an example.

A third improvement is in Chapter 5, *Register Usage and Instruction Scheduling*, for which I developed a vector scheduling algorithm that is better able to use more vector registers than the scheduling algorithm used in the version of the *cft77* compiler I used during my work term at Cray Research, Incorporated in the fall of 1990. Described in Figure 5.9 (on page 94), my algorithm is a list scheduling one that has been modified for a vector architecture and is similar to Tang and Davdison's *simple vector scheduler* [107]. List scheduling algorithms have previously been used for VLIW and scalar architectures [34, 48]. An algorithm similar to the one I developed has been implemented in a version of the *cft77* compiler more recent than the one I used for my studies [62].

The fourth and final improvement is in Chapter 6, ***Bus Usage and Register Assignment***, for which I developed an optimal algorithm that assigns values to the minimum number of registers for a given execution order of a dependence graph. Although other optimal algorithms have been published for this problem, the proof of their optimality relies on the use of a stack and does not allow any choice for a register [44, 51, 27]. In contrast, I based the proof for my algorithm, which is presented in Section 6.2.3 (pages 123 to 130), on a necessary but not sufficient condition for optimality that is completely independent of register choice. As a result, my algorithm emphasizes the importance of assigning values in the order of their creation times and does not specify which register to choose when there are several candidates, a choice that can be left as an implementation detail without affecting the optimality of my algorithm.

7.1.2 Syntheses of Published Material

The next set of contributions, which appear in Chapters 2 and 3, provide new observations of already published material. The first contribution that synthesizes known material concerns stripmining, which is the classic technique for executing long loops with vector instructions, and loop unrolling, which is a standard compiler optimization for executing a loop with scalar instructions. In Section 2.3.2 of Chapter 2, *Fundamentals of Vector Architectures* (pages 24 to 29), I showed that using vector instructions in a stripmined loop is, in fact, a compact form of loop unrolling, an observation that was made parenthetically by Jouppi and Wall [70].

Another contribution from this chapter is the presentation of the properties of a vectorizable program fragment. Because it is the responsibility of a compiler to identify such parts, most descriptions of a vectorizable program fragment are given from the perspective of a compiler. However, such presentations also include what cannot be vectorized because of inadequate compilation technology. In contrast, to illustrate the restrictions imposed by vector hardware, I derive the properties of a vectorizable program fragment based on characteristics of vector hardware (in Section 2.3.1, pages 17 to 24).

The third and largest contribution is the synthesis of observations and data in Chapter 3, *A Case for Vector Architectures*. Although some of these have been published before and others are obvious, I transform these individual items into arguments that together advocate the implementation of vector architectures over superscalar ones in CMOS VLSI technology. Following is a list of the more convincing arguments:

- In Section 3.1.3 (pages 38 to 44), I showed how the partitioning of a vector register file provides 8 times as many registers but requires only 1.25 times as much area as a monolithic register file with 64 registers and comparable bandwidth.

- In Section 3.2.1 (pages 46 to 51), I used data from Wall's parallelism study to show that vectorizable program fragments are rich in parallelism and are, furthermore, most likely to be the more time-consuming programs in a workload [115].

- In the subsequent section, Section 3.2.2 (pages 51 to 54), to demonstrate the effectiveness of this type of parallelism, I explained how 25 times as many instructions could be executed if the hardware were to make full use of the intrinsic parallelism in Wall's workload. The use of parallelism to increase the size of a workload rather than reduce execution time was first documented by Gustafson [54].

- Finally, because it would be unwise to completely ignore the effects of Amdahl's Law, I presented data in Section 3.2.3 (pages 54 to 58) showing that a superpipelined architecture, such as the Cray Y-MP scalar processor, can take advantage of the limited parallelism in non-vectorizable program fragments. Moreover, a different interpretation of data published by Weiss and Smith [119] shows that, for vectorizable program fragments, vector hardware in combination with superpipelined hardware provides significant improvement in performance over superpipelined hardware alone. Jouppi and Wall have also argued for the use of superpipelined hardware, albeit without any vector hardware, to support fine-grain parallelism [70, 69].

7.1.3 Extensions to the State of the Art

The last and most significant set of contributions appear in Chapters 5 and 6. These contributions extend the state of the art with a compiler algorithm for a new register organization and with empirical data that strengthen qualitative observations. One of these contributions is in Chapter 6, *Bus Usage and Register Assignment*, for which I developed an assignment algorithm for a vector register file where more than one register shares a bus. In Section 6.1 (pages 116 to 130), I modeled the problem of locally assigning values to such a register file as a problem of coloring two graphs, thus building upon previous work that uses graph coloring to model the problem of globally assigning registers [35, 36, 18]. Based on definitions from graph theory, I also demonstrated how alternative representations for graphs reveal any special structure that these graphs may have [82, 49]. In addition, I extended my algorithm to allow assignments of the form `Vi<-Vi op Vj` and presented data showing that the reuse of registers in such an assignment has minimal impact on overall register usage (Section 6.4.2 (pages 133 to 139)).

In addition to a new algorithm, I also contributed to the state of the art by carrying out experiments that validated hypotheses concerning the effectiveness of the compiler algorithms I developed and the register organizations I studied. The infrastructure for these experiments, which was provided by Cray Research, Incorporated and is described in Chapter 4, *Common Experimental Framework*, consists of the following three items:

1. a development version of a production vectorizing compiler,
2. a simulator that models the Y-MP vector processor, an architecture which has fully flexible chaining capabilities, and
3. a set of 36 vectorizable loops that are extracted from actual applications.

Both Chapters 5 and 6 contain contributions in the form of empirical data.

In Section 5.1 of Chapter 5, *Register Usage and Instruction Scheduling* (pages 72 to 83), I hypothesized that both more than 8 vector registers and a scheduling algorithm different from the one used in the 1990 version of the *cft77* compiler are needed to improve performance. Although two research groups have also hypothesized the need for more registers and a third has carried out experiments to determine how many, their research centers around using *polycyclic vector scheduling* for vector architectures without chaining capabilities [108, 32, 33, 81]. The empirical data I presented in Section 5.4 (pages 95 to 104) not only validated my hypotheses but also sharpened the qualitative descriptions I presented by indicating *how many* more registers are needed to improve performance by *how much* and *how frequently*.

In Section 6.1 of Chapter 6, *Bus Usage and Register Assignment* (pages 111 to 116), I observed that only a subset of simultaneously live values are actually used at any given time and hypothesized that partitioning a vector register file would reduce the cost of implementing one with minimal loss in performance. The data I presented in Section 6.4 (pages 131 to 140) not only indicates the effectiveness of my assignment algorithm but also shows that the subset of simultaneously live values is small enough for the majority of loops in the CRI workload to effectively use a partitioned vector register file. Moreover, this data indicates that my hypothesis is true once enough partitions are provided, thus providing quantitative evidence that partitioning is a cost-effective method for improving performance. I do not know of any other published work that provides data demonstrating the effectiveness of a partitioned vector register file.

7.2 Future Studies

Although my investigations varied the number of registers and buses in a vector register file, the number of elements per vector register remained constant at 64. More simulation studies are needed to determine the effect of longer and shorter vector lengths on performance. One experiment could verify the hypothesis that increasing the number of vector registers improves performance more than does increasing the number of elements per vector register. Another experiment could determine if performance does *not* decline significantly when the vector length is shortened to 32 elements per vector register in order to demonstrate that a register organization with 8 buses and 32 registers provides an excellent tradeoff between increased cost and improved performance.

The assignment algorithm I presented did not include any contingency for when number of registers or buses assigned exceeds what is provided in hardware. A study for the future is to modify the algorithm to handle this case and evaluate the impact on performance of spilling registers in a partitioned vector register file. Of particular interest to such a future study is a register organization with 8 buses and 16 registers, the one for which I was unable to produce strong performance data in Section 6.4 of Chapter 6, *Bus Usage and Register Assignment* (pages 131 to 140).

Although my algorithm for optimally assigning registers does not directly address the problem of spilling registers, it can be easily extended to do so. What is unclear is whether it would remain optimal. Horwitz *et al.* as well as Prabhala and Sethi have developed algorithms that spill registers using the minimal number of memory references for a given number of registers in hardware [60, 95]. These algorithms, however, apply to index registers and stack registers, respectively. For general purpose registers, Hsu, Fischer and Goodman have presented an optimal algorithm for register spilling that is an extension of the algorithm by Horwitz *et al.* [63]. Because the algorithm is based on enumeration, however, it can become computationally impractical for large programs. Because none of these algorithms model the problem of spilling registers as a graph coloring one, a project of theoretical interest is to determine whether my optimal algorithm for locally assigning registers can be extended to spill registers using the minimal number of memory references for a given number of registers in hardware.

In Chapter 5, *Register Usage and Instruction Scheduling*, I showed that, although a list scheduling algorithm outperforms the one used by the 1990 version of the *cft77* compiler, the reverse is true for a few dependence graphs, two of which are shown in Figures 5.12 and 5.13 (on pages 98 and 99). An undertaking for the future is to develop an algorithm that performs as well as the list scheduler but never does worse than the *cft77* compiler. One possibility is an algorithm that schedules operations in an order that is the reverse of the order used by either the *cft77* or list scheduler and that uses a placement strategy similar to the *cft77*'s one. Although such an algorithm produces the same execution order as does the *cft77* scheduler for the dependence graphs shown in in Figures 5.12 and 5.13, more simulation studies are needed to determine whether it performs as well or better for the rest of the CRI workload. Because both the order and the strategy differ in the list and *cft77* schedulers, another study for the future is to determine which — order or strategy — has more impact on performance. Because there are dependence graphs for which either makes a difference, a potential line of investigation is to analyze actual dependence graphs for any special structures.

The investigations in this dissertation explored various register organizations but always used the configuration of functional units from the Cray Y-MP. Another project for the future is to compare the performance of different configurations of functional units, varying both the number and types. An extension to this proposed study and those in this dissertation is to determine the appropriate numbers of buses and registers needed to use different configurations of functional units effectively in an attempt to establish a rule-of-thumb that would provide this ratio analytically. Such a study would also verify my hypothesis that a partitioned vector register file with 4 buses is an inadequate design for the configurations of functional units in the Ardent Titan and Fujitsu VPU. Because binary compatibility is not affected by a change of functional units, an interesting value to quantify is the impact on performance when using an execution order scheduled for a configuration different from what is provided in hardware.

The vector architecture I used implements fully flexible chaining. A final suggestion for future work is to combine this work with work done for vector architectures that implement no chaining [108, 32, 33, 81] in order to examine the interaction among levels of chaining, register organization, and scheduling algorithm. For cost reasons, it may be desirable to reduce the level of chaining. One example is to prevent vector memory instructions from chaining with non-memory ones in order to simplify the memory system. Although less chaining provides less opportunity for parallelism to occur, sophisticated scheduling algorithms, such as *polycyclic vector scheduling*, can be used to increase the amount of parallelism but at the expense of more registers. Interesting values to compare are the numbers of registers and buses needed to compensate for the lack of chaining and the numbers needed when there is chaining.

Bibliography

[1] Thomas L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedulers for parallel processing systems. *Communications of the ACM*, 17(12):685–691, December 1974.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles,, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[3] A.V. Aho and S.C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, August 1976.

[4] Randy Allen and Steve Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the Symposium on Compiler Construction*, pages 241–249, June 1988.

[5] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[6] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica Lam, Onat Menzilcioglu, and Jon A.Webb. The Warp computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[7] Siamak Arya. An optimal instruction-scheduling model for a class of vector processors. *IEEE Transactions on Computers*, C-34(11):981–995, November 1985.

[8] Melvin C. August, Gerald M. Brost, Christopher C. Hsiung, and Alan J. Schiffleger. Cray X-MP: The birth of a supercomputer. In Yale Patt [91], pages 45–52.

[9] J.C. Beatty. Register assignment algorithm for generation of highly optimized object code. *IBM Journal of Research and Development*, 18(1):20–39, January 1974.

[10] David Bernstein, Haran Boral, and Ron Y. Pinter. Optimal chaining in expression trees. *IEEE Transactions on Computers*, C-37(11):1366–1374, November 1988.

[11] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahson, and Ron Y. Pinter. Spill code minimization techniques for optimizating compilers. In *Conference on Programming Language Design and Implemen-*

tation, pages 258–263, June 1989. Special issue of *SIGPLAN Notices*, Vol. 24, No. 7, July 1989.

[12] M. Berry, D. Chen, P. Koss, and D. Kuck. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. CSRD Rpt. No. 827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801, November 1988.

[13] Greg Blanck and Steve Krueger. SuperSPARC: A fully integrated superscalar processor. In *Hot Chips III: A Symposium on High-Performance Chips*. IEEE, August 1991.

[14] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H.T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P.S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339. ACM and IEEE, November 1988.

[15] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. The effect on RISC performance of register set size and structure versus code generation strategy. In *Proceedings of the International Symposium on Computer Architecture*, pages 330–339. ACM and IEEE, May 1991.

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

[17] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52. ACM and IEEE, April 1991.

[18] Gregory J. Chaitin, Marc A. Auuslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[19] Mike Chastain, Gary Gostin, Jim Mankovich, and Steve Wallach. The Convex C240 architecture. In *Proceedings of Supercomputing '88*, pages 321–329. ACM and IEEE, November 1988.

[20] Tzi-cker Chiueh. Multi-threaded vectorization. In *Proceedings of the International Symposium on Computer Architecture*, pages 352–359. ACM and IEEE, 1991.

[21] R.S. Clark and T.L. Wilson. Vector system performance of the IBM 3090. *IBM Systems Journal*, 25(1):63–82, 1986.

[22] Robert P. Colwell, W. Eric Hall, Chandra S. Joshi, David B. Papworth, Paul K. Rodman, and James E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Proceedings of Supercomputing '90*, pages 910–919. ACM and IEEE, November 1990.

[23] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–182. ACM and IEEE, October 1987. Also published in *IEEE Transactions on Computers*, C-37(8):967–979, August 1988.

[24] Cray Research, Incorporated. *Cray Y-MP Computer Systems Functional Description Manual, HR-4001.* Distribution Center, 2360 Pilot Knob Road, Mendota Heights, MN 55120, January 1988.

[25] Cray Research, Incorporated. *Designing for Speed: An Introduction to the Cray World of Computing.* Mendota Heights, MN, 1989. Notes for course of the same title.

[26] Cray Research, Incorporated. *Cft77 Online Manual Page.* Unicos 6.0/Cft77 5.0, 1990.

[27] G.B. Dantzig and G. Reynolds. Optimal assignment of computer storage by chain decomposition of partially ordered sets. Operations Research Center Report ORC-66-6, University of California, Berkeley, March 1966. Cited in [51].

[28] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Architectural support for overlapped loops on the Cydra 5. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, April 1989.

[29] James Demmel. Private communication, 28 January 1992.

[30] DESCR file for *fpppp*. Part of distribution tape for SPEC benchmark suite [106], September 1989.

[31] Tom Diede, Carl F. Hagenmaier, Glen S. Miranker, Jonathan J. Rubinstein, and William S. Worley, Jr. The Titan graphics supercomputer architecture. *IEEE Computer*, 21(9):13–30, September 1988.

[32] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Squeezing more CPU performance out of a Cray-2 by vector block scheduling. In *Proceedings of Supercomputing '88*, pages 237–246. ACM and IEEE, November 1988.

[33] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Compiler techniques for optimizing memory and register usage on the Cray-2. *International Journal of High Speed Computing*, 2(2):193–222, June 1990.

[34] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, 1986. PhD thesis, Yale University, May 1984.

[35] A.P. Ershov. Reducing the problem of memory allocation when compiling programs to one of colouring the vertices of graphs. *Dokl. Akad. Nauk. S.S.S.R.*, 142(4), 1962. Cited in [36].

[36] A.P. Ershov, L.L. Zmiyevskaya, R.D. Mishkovitch, and L.K. Trokhan. Economy and allocation of memory in the ALPHA-translator. In A.P. Ershov, editor, *The Alpha Automatic Programming System*, Chapter 9, pages 161–196. Academic Press, 1971. Translated by J. McWilliam.

[37] Matthew K. Farrens and Andrew R. Pleszkun. Strategies for achieving improved processor throughput. In *Proceedings of the International Symposium on Computer Architecture*, pages 362–369. ACM and IEEE, May 1991.

[38] Sidney Fernbach, editor. *Supercomputers: Class VI Systems, Hardware and Software*. North-Holland, Elsevier Science Publishers, Inc., 1986.

[39] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[40] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the International Symposium on Computer Architecture*, pages 140–150. ACM and IEEE, 1983.

[41] Josh Fisher. Private communication, 14 May 1990.

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

[43] L.R. Ford, Jr. and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[44] R.A. Freiburghouse. Register allocation via usage counts. *Communications of the ACM*, 17(11):638–642, November 1974.

[45] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.

[46] Micahel R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[47] Patrick P. Gelsinger, Paolo A. Gargini, Gerhard H. Parker, and Albert Y.C. Yu. Microprocessors circa 2000. *IEEE Spectrum*, 26(10):43–47, October 1989.

[48] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the Symposium on Compiler Construction*, pages 11–16, July 1986. Special issue of *SIGPLAN Notices*, 21(7).

[49] Michel Gondran and Michel Minoux. *Graphs and Algorithms*. John Wiley and Sons, 1984. Translated by Steven Vajda.

[50] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *International Conference on Supercomputing*, pages 442–452. ACM, July 1988.

[51] David Gries. *Compiler Construction for Digital Computers.* John Wiley and Sons, Inc., 1971.

[52] Thomas R. Gross. *Code Optimization of Pipeline Constraints.* PhD thesis, Stanford University, Computer Systems Laboratory, December 1983. Also published as Technical Report No. 83-255.

[53] John Gustafson, Diane Rover, Stephen Elbert, and Michael Carter. SLALOM benchmarks. *Supercomputing Review,* 4(7):52–59, July 1991. Send e-mail to *netlib@tantalus.al.iastate.edu* for more information.

[54] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM,* 31(5):532–533, May 1988.

[55] James J. Hack. Peak versus sustained performance in highly concurrent vector machines. *IEEE Computer,* 19(9):11–19, September 1986.

[56] Luddy Harrison. Automatic parallelization of symbolic computation. Notes for course at Illinois Summer Institute on Parallelizing Compilers, University of Illinois at Urbana-Champaign, July 1990.

[57] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer,* 24(9):18–29, September 1991. Special 40th anniversary issue on "The Promise of the Next Decade.".

[58] Mark D. Hill and David A. Wood. Guest editor's introduction: Hot Chips II Symposium. *IEEE Micro,* pages 8–9, June 1991. Special issue highlighting presentations from Hot Chips II, 1990.

[59] R.W. Hockney and C.R. Jesshope. *Parallel Computers,* Section 2.4 The FPS AP-120B, pages 126–143. Adam Hilger Ltd., 1981.

[60] L.P. Horwitz, R.M. Karp, R.E. Miller, and S. Winograd. Index register allocation. *Journal of the ACM,* 13(1):43–61, January 1966.

[61] *Hot Chips III: A Symposium on High-Performance Chips,* IEEE, August 1991.

[62] Wei-Chung Hsu. Private communications, 1990–1992.

[63] Wei-Chung Hsu, Charles N. Fischer, and James R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Transactions on Software Engineering,* 15(10):1252–1260, October 1989.

[64] Hideyuki Iino, Hiromasa Takahashi, Takao Sukemura, Masaharu Kimura, Koichi Fujita, and Shosuke Mori. A 289MFLOPS sincle-chip supercomputer. In *IEEE International Solid-State Circuits Conference,* volume 35, pages 112–113, February 1992.

[65] Paul M. Johnson. An introduction to vector processing. *Computer Design,* 17(2):89–97, February 1978.

[66] William M. Johnson. *Super-Scalar Processor Design*. PhD thesis, Stanford University, Computer Systems Laboratory, Stanford, CA, June 1989. Also published as Technical Report No. CSL-TR-89-383.

[67] Richard D. Jolly. A 9-ns, 1.4-gigabyte/s, 17-ported CMOS register file. *IEEE Journal of Solid-State Circuits*, 26(10):1407–1412, October 1991.

[68] Tom Jones. Engineering design of the Convex C2. In Yale Patt [91], pages 36–44.

[69] Norman P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, C-38(12):1645–1658, December 1989.

[70] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282. ACM and IEEE, April 1989.

[71] Earl Killian. MIPS R4000 technical overview: 64 bits/100 MHz or bust. In *Hot Chips III: A Symposium on High-Performance Chips*. IEEE, August 1991.

[72] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the International Symposium on Computer Architecture*, pages 43–53. ACM and IEEE, May 1991.

[73] Leslie Kohn and Sai Wai Fu. A 1 000 000 transistor microprocessor. In *IEEE International Solid-State Circuits Conference*, volume 32, pages 54–55, February 1989. Cited in [93].

[74] David J. Kuck. *The Structure of Computers and Computations, Vol. 1*. John Wiley and sons, Inc., 1978.

[75] Richard l. Sites and Richard T. Witek. Alphad architecture and first implementation. In *Digest of Papers for the* 37th *Annual IEEE International Computer Conference (COMPCON)*, page 214, February 1992. Abstract only.

[76] Monica S. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, 1989. PhD thesis, Carnegie Mellon University, 1987.

[77] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74. ACM and IEEE, April 1991.

[78] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. In *Proceedings of the Symposium on Compiler Construction*, pages 255–263, July 1986. Special issue of *SIGPLAN Notices*, 21(7).

[79] James Richard Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California at Berkeley, Computer Science Division (EECS), May 1982. Also published as Technical Report No. UCB/CSD 89/502.

[80] Neil Lincoln. Demolition of reasonable principles by pretty pathetic practices. Keynote talk at *Symposium on Principles and Practice of Parallel Programming,* March 1990.

[81] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. Vector register design for polycyclic vector scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 154–163. ACM and IEEE, April 1991.

[82] Clifford W. Marshall. *Applied Graph Theory.* John Wiley and Sons, 1971.

[83] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *SIGPLAN Conference on Programming Language Design and Implementation,* pages 1–14. ACM, June 1991. Special issue of *SIGPLAN Notices,* 26(6).

[84] Frank H. McMahon. The Livermore FORTRAN Kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, University of California, Livermore, CA 94550, December 1986.

[85] Kenichi Miura. Fujitsu's supercomputer: FACOM vector processor system. In Fernbach [38], Chapter 7, pages 137–151.

[86] Glenford J. Myers, Albert Y.C. Yu, and David L. House. Microprocessor technology trends. *Proceedings of the IEEE,* 74(12):1605–1622, December 1986.

[87] Ikuo Nakata. On compiling algorithms for arithmetic expressions. *Communications of the ACM,* 10(8):492–494, August 1967.

[88] Toshihiko Odaka, Shigeo Nagashima, and Shun Kawabe. Hitachi supercomputer S-810 array processor system. In Fernbach [38], Chapter 6, pages 113–136.

[89] Richard R. Oehler and Michael W. Blasgen. IBM RISC Systems/6000: Architecture and performance. *IEEE Micro,* pages 14–17,56–62, June 1991. Special issue highlighting presentations from Hot Chips II, 1990.

[90] Fuyuki Okamoto, Yasuhiko Hagihara, Chie Ohkubo, Naoki Nishi, Hachiro Yamada, and Tadayoshi Enomoto. A 200-MFLOPS 100-MHz 64-b BiCMOS vector-pipelined processor (VPP) ULSI. *IEEE Journal of Solid-State Circuits,* 26(12):1885–1893, December 1991.

[91] Yale Patt, guest editor. Special issue on "Real machines: Design choices/engineering trade-offs.". *IEEE Computer,* 22(1), January 1989.

[92] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[93] Tekla S. Perry. Intel's secret is out. *IEEE Spectrum*, 26(4):22–28, April 1989.

[94] Val Popescu, Merle Schultz, John Spracklen, Gary Gibson, Bruce Lightner, and David Isaman. The Metaflow architecture. *IEEE Micro*, pages 10–13,63–73, June 1991. Special issue highlighting presentations from Hot Chips II, 1990.

[95] Bhaskaram Prabhala and Ravi Sethi. Efficient computation of expressions with common subexpressions. *Journal of the ACM*, 27(1):146–163, January 1980.

[96] Vaclav Rajlich and M. Drew Moshier. A survey of algorithms for register allocation in straight-line programs. Technical Report CRL-TR-14-84, The University of Michigan Computing Research Laboratory, February 1984.

[97] B. Ramakrishna Rau, David W.L. Yen, Wei Yen Ross, and A. Towle. The Cydra 5 departmental supercomputer: Design philosphies, decisions and trade-offs. In Yale Patt [91], pages 12–35.

[98] R.R. Redziejowski. On arithmetic expressions and trees. *Communications of the ACM*, 12(2):81–84, February 1969.

[99] R. Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[100] Ravi Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, September 1975.

[101] Richard L. Sites. An analysis of the Cray-1 computer. In *Proceedings of the International Symposium on Computer Architecture*, pages 101–106. ACM and IEEE, April 1978.

[102] Debra L. Slater, David M. Fenwick, D. John Shakshober, and Douglas D. Williams. Vector processing on the VAXvector 6000 Model 400. *Digital Technical Journal*, 2(2):11–26, Spring 1990.

[103] J.E. Smith and W. Taylor. Notes on evaluation of linear recurrences on a vector processor. Unpublished notes, January 1990.

[104] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62. ACM and IEEE, April 1991.

[105] Gurindar S. Sohi and Sriram Vajapeyam. Tradeoffs in instruction format design for horizontal architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–25. ACM and IEEE, April 1989.

[106] *SPEC Benchmark Suite Release 1.0*. National Computer Graphics Association (administrator), 2722 Merrilee Dr., Suite 200, Fairfax, Virginia 22031, September 1989.

[107] Ju-ho Tang and Edward S. Davidson. An evaluation of Cray-1 and Cray X-MP performance on vectorizable Livermore Fortran kernels. In *International Conference on Supercomputing*, pages 510–518. ACM, July 1988.

[108] Ju-ho Tang, Edward S. Davidson, and Johau Tong. Polycyclic vector scheduling vs. chaining on 1-port vector supercomputers. In *Proceedings of Supercomputing '88*, pages 122–129. ACM and IEEE, November 1988.

[109] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, October 1991.

[110] Jack R. Thompson. The CRAY-1, the CRAY X-MP, the CRAY-2 and beyond: The supercomputers of Cray Research. In Fernbach [38], Chapter 4, pages 69–81.

[111] Garold S. Tjaden and Michael J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers*, C-19(10):889–895, October 1970.

[112] R.M. Tomasulo. An efficient algorithm for exploting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[113] S.G. Tucker. The IBM 3090 system: An overview. *IBM Systems Journal*, 25(1):4–19, 1986.

[114] Hideo Wada, Koichi Ishii, Masakazu Fukagawa, Hiroshi Murayama, and Shun Kawabe. High-speed processing schemes for summation type and iteration type vector instructions on HITACHI Supercomputer S-820 system. In *International Conference on Supercomputing*, pages 197–206. ACM, July 1988.

[115] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188. ACM and IEEE, April 1991.

[116] Tadashi Watanabe, Hiroshi Katayama, and Akihiro Iwaya. Introduction of NEC Supercomputer SX system. In Fernbach [38], Chapter 8, pages 153–167.

[117] John Wawrzynek. Private communication, 21 August 1991.

[118] John Wawrzynek and Thorsten von Eicken. MIMIC: A custom VLSI parallel for musical sound synthesis. Technical Report No. UCB/CSD 90/578, University of California, Computer Science Division (EECS), University of California, Berkeley, CA 94720, July 1990.

[119] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–109. ACM and IEEE, October 1987.

[120] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, 1989. Ph.D. Thesis, University of Illinois at Urbana-Champaign, October 1982.